Javari: Adding Reference Immutability to Java

by

Matthew S. Tschantz

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

August 2006

Author .....
Department of Electrical Engineering and Computer Science
August 22, 2006

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

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

Javari: Adding Reference Immutability to Java

by

Matthew S. Tschantz

Submitted to the Department of Electrical Engineering and Computer Science
on August 22, 2006, in partial fulfillment of the
requirements for the degree of
Master of Engineering in Computer Science and Engineering

Abstract

This paper describes a programming language, Javari, that is capable of expressing and enforcing immutability constraints. The specific constraint expressed is that the abstract state of the object to which an immutable reference refers cannot be modified using that reference. The abstract state is (part of) the transitively reachable state: that is, the state of the object and all state reachable from it by following references. The type system permits explicitly excluding fields from the abstract state of an object. For a statically type-safe language, the type system guarantees reference immutability.

The type system is distinguishes the notions of assignability and mutability; integrates with Java's generic types and with multi-dimensional arrays; provides a mutability polymorphism approach to avoiding code duplication; and has type-safe support for reflection and serialization. This paper describes a core calculus including formal type rules for the language.

Additionally, this paper describes a type inference algorithm that can be used convert existing Java programs to Javari. Experimental results from a prototype implementation of the algorithm are presented.

Thesis Supervisor: Michael D. Ernst
Title: Associate Professor

Acknowledgments

I thank Michael Ernst for his guidance, encouragement, and willingness to help think through any problem. Adrian Birka implemented the Javari2004 prototype compiler. Matthew McCutchen helped with the implementation and evaluation of Javarifier including conducting the Jolden case study. Jaime Quinonez assisted with testing Javarifier. Matthew Papi designed and implemented the extended Java annotation system that was used by Javarifier. Finally, we are grateful to Joshua Bloch, John Boyland, Gilad Bracha, Doug Lea, Sandra Loosemore, and Jeff Perkins for their comments on the Javari design. This work was supported in part by NSF grants CCR-0133580 and CCR-0234651, DARPA contract FA8750-04-2-0254, and gifts from IBM and Microsoft.

Contents

List of Figures

Chapter 1

Introduction

The Javari programming language extends Java to allow programmers to specify and enforce reference immutability constraints. An immutable, or read-only, reference cannot be used to modify the object, including its transitive state, to which it refers. A type system enforcing reference immutability has a number of benefits: it can increase expressiveness; it can enhance program understanding and reasoning by providing explicit, machine-checked documentation; it can save time by preventing and detecting errors that would otherwise be very difficult to track down; and it can enable analyses and transformations that depend on compiler-verified properties.

Javari's type system differs from previous proposals (for Java, C++, and other languages) in a number of ways. First, it offers reference, not object, immutability; reference immutability is more flexible, as it provides useful guarantees even about code that manipulates mutable objects. For example, many objects are modified during a construction phase but not thereafter. As another example, an interface can specify that a method that receives an immutable reference as a parameter does not modify the parameter through that reference, or that a caller does not modify a return value. Furthermore, a subsequent analysis can strengthen reference immutability into stronger guarantees, such as object immutability, where desired.

Second, Javari offers guarantees for the entire transitively reachable state of an object — the state of the object and all state reachable by following references through its (non-static) fields. A programmer may use the type system to support reasoning about either the representation state of an object or its abstract state; to support the latter, parts of a class can be marked as not part of its abstract state.

Third, Javari combines static and dynamic checking in a safe and expressive way. Dynamic checking is necessary only for programs that use immutability downcasts, but such downcasts can be convenient for interoperation with legacy code or to express facts that cannot be proved by the type system. Javari also offers parameterization over immutability.

Experience with over 160,000 lines of Javari code, including the Javari compiler itself, indicates that Javari is effective in practice in helping programmers to document their code, reason about it, and prevent and eliminate errors [5]. Despite this success, deficiencies of a previous proposal for the Javari language limit its applicability in practice. Many of the limitations of [5] were addressed in [48], an abbreviated version of this paper. In addition to the contributions of [48], this paper describes a type inference that is capable of converting Java programs to the Javari language and discusses experience with a prototype implementation of the analysis. Furthermore, this paper rectifies a flaw in the core calculus

that was previously presented.

This paper demonstrates that it is possible to implement a usable extension to Java that enforces reference immutability. In particular, for the language to be useful to programmers it must obey the following design goals.

Non-conversion Disallow the conversion, through assignment or unsafe casts, of a read-only reference to a mutable reference. This type-soundness goal allows reference immutability serve as a way to protect the abstract state of objects from mutation.

Transitive Provide transitive (deep) immutability that protects the entire abstract state of an object. This goal enables programmers to use reference immutability as a means to reason abstractly about the mutation of objects.

Flexible Allow programmers to exclude parts of an object's concrete state from the object's abstract state. This goal ensures that the language is usable in real-world programs that cache information, use complex data structures, and use utility and debugging classes such as loggers.

Compatible Be backwards compatible with existing Java programs and the Java Virtual Machine, to allow programmers to easily migrate to Javari.

Usable Naturally extend the Java language in a way that is intuitive to current programmers. This goal is a necessity for the language to be adopted by the mainstream Java programing community.

The rest of this document is organized as follows. Chapter 2 presents examples that motivate the need for reference immutability in a programming language. Chapter 3 describes the Javari language. Chapter 4 formalizes the type rules of Javari. Chapter 5 discusses extensions to the basic Javari language to make the language more useful in practice. Chapter 6 presents a type inference algorithm that adds Javari's immutability constraints to an existing Java program. Chapter 7 discusses related work, including a previous dialect of the Javari language. Chapter 8 concludes with a summary of contributions. Appendix A gives examples of each of Javari's assignability and mutability types. Finally, appendix B explains why Java's annotation mechanism is not expressive enough to encode Javari's constraints.

Chapter 2

Motivation

Reference immutability provides a variety of benefits. Many other papers and books (see chapter 7) have justified the need for immutability constraints. The following simple examples suggest a few uses for immutability constraints and give a flavor of the Javari language.

Consider a voting system containing the following routine:

```
ElectionResults tabulate(Ballots votes) { ... }
```

It is necessary to safeguard the integrity of the ballots. This requires a machine-checked guarantee that the routine does not modify its input `votes`. Using Javari, the specification of `tabulate` could declare that `votes` is read-only:

```
ElectionResults tabulate(readonly Ballots votes) {
  ...  // cannot tamper with the votes
}
```

and the compiler ensures that implementers of `tabulate` do not violate the contract.

Accessor methods often return data that already exists as part of the representation of the module. For example, consider the `Class.getSigners` method, which returns the entities that have digitally signed a particular implementation. In JDK 1.1.1, its implementation is approximately:

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

This is a security hole, because a malicious client can call `getSigners` and then add elements to the `signers` array.

Javari permits the following solution:

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

The `readonly` keyword ensures that the caller of `Class.getSigners` cannot modify the returned array.

An alternate solution to the `getSigners` problem, which was actually implemented in later versions of the JDK, is to return a copy of the array `signers` [6]. This works, but is error-prone and expensive. For example, a file system may allow a client read-only access to its contents:

```
class FileSystem {
  private List<Inode> inodes;
  List<Inode> getInodes() {
    ... // Unrealistic to copy
  }
}
```

Javari allows the programmer to avoid the high cost of copying `inodes` by writing the return type of the method as:

```
readonly List<readonly Inode> getInodes()
```

This return type prevents the `List` or any of its contents from being modified by the client. As with all parameterized classes, the client of `List` specifies the type argument, including whether it is read-only or not, independently of the parameterized type `List`.

A similar form of dangerous, mutation-permitting aliasing can occur when a data structure stores information passed to it (for instance, in a constructor) and a client retains a reference. Use of the `readonly` keyword again ensures that either the client's copy is read-only and cannot be modified, or else the data structure makes a copy, insulating it from changes performed by the client. In other words, the annotations force programmers to copy only when necessary.

As a final example, reference immutability can be used to establish the stronger guarantee of object immutability: a value is never modified if all references to it are immutable (which can be established by a subsequent analysis). For example, there is only one reference when an object is first constructed. As another simple example, some data structures must be treated as mutable when they are being initialized, but as immutable thereafter; an analysis can build upon Javari in order to make both the code and the reasoning simple.

```
Graph g1 = new Graph();
... construct cyclic graph g1 ...
// Suppose no aliases to g1 exist.
readonly Graph g = g1;
g1 = null;
```

Chapter 3

Language Design

In Java, each variable or expression has an *assignability* property, controlled by the `final` keyword, that determines whether it may be the lvalue (left-hand side) of an assignment. In Javari, each (non-primitive) reference additionally has a *mutability* property that determines whether its abstract state may be changed (for example, by setting its fields). Both properties are specified in the source code, checked at compile time, and need no run-time representation. The assignability and mutability properties determine whether various operations, such as reassignment and calling side-effecting methods, are permitted on a reference.

Javari extends Java by providing additional constraints that may be placed on the assignability and mutability of a reference. Javari's keywords are those of Java, plus five more: `assignable` (the complement of `final`); `readonly` and its complement `mutable`; `romaybe`, a syntactic convenience that reduces code duplication; and `? readonly`, which is used to create wildcards that vary over mutability.[1]

For ease of presentation, this chapter introduces the various aspects of the Javari language incrementally then concludes (in section 3.8) with a summary of the language as a whole. While some aspects of the language are explained in great detail, they are merely results of the application of Javari's simple design goals: preventing the conversion of a read-only reference into a mutable reference, providing transitive immutability, allowing programmers the flexibility to exclude parts of an object's concrete state from the object's abstract state, providing backwards compatibility, and naturally extending Java.

The rest of the section explains how Javari's assignability and mutability concepts interact with the constructs of Java. Sections 3.1 and 3.2 introduce the concepts of assignability and mutability, respectively, and discuss their application to local variables. Section 3.3 introduces read-only methods and section 3.4 presents immutable classes. Section 3.5 discusses the assignability and mutability of fields. Section 3.6 applies assignability and mutability to generic classes, and section 3.7, to arrays. Finally, section 3.8 summarizes Javari's syntax.

3.1 Assignability and final references

Assignability determines whether a reference may be reassigned. By default, a local variable is assignable: it may be reassigned. Java's `final` keyword makes a variable unassignable:

[1] For ease of reading, we present Javari's immutability as keywords; however, as discussed in appendix B, Javari can be implemented using extended Java annotations.

it cannot be reassigned. Javari retains the `final` keyword, but provides greater control over the assignability of references via the `assignable` keyword (see section 3.5.3) and the concept of this-assignability (see section 3.5.1). The following is an example of assignable and unassignable references.

```
final          Date d = new Date(); // unassignable
/*assignable*/ Date e = new Date(); // assignable
e = new Date(); // OK
d = new Date(); // error: d cannot be reassigned
```

Above, "`/*assignable*/`" is not a required part of the program's syntax but a comment to remind the reader that Java references are assignable by default. Except in section 5.3.5, all errors noted in code examples are compile-time errors.

Assignability does not affect the type of the reference. Assignability constraints add no new types to the Java type hierarchy, and there is no type `final T` (for an existing type `T`).

3.2 Mutability and read-only references

Mutation is any modification to an object's abstract state. The abstract state is (part of) the transitively reachable state, which is the state of the object and all state reachable from it by following references. It is important to provide transitive (deep) immutability guarantees in order to capture the full abstract state represented by a Java object. Clients of a method or other construct are typically interested in properties of the abstraction, not the concrete representation. (Javari provides ways to exclude selected fields from the abstract state; see section 3.5.3.)

Javari's `readonly` type modifier declares immutability constraints. A reference that is declared to be of a `readonly` type cannot be used to mutate the object to which it refers. For example, suppose a variable, `rodate`, is declared to have the type `readonly Date`. Then `rodate` can only be used to perform actions on the `Date` object that do not modify it:

```
readonly Date rodate = ...; // readonly reference to a Date object
rodate.getMonth();          // OK
rodate.setYear(2005);       // error
```

For every Java reference type `T`, `readonly T` is a valid Javari type and a supertype of `T`; see figure 3-1. A mutable reference may be used where a read-only reference is expected, because it has all the functionality of a read-only reference. A read-only reference may not be used where a mutable reference is expected, because it does not have all the functionality of a mutable reference: it cannot be used to modify the state of the object to which it refers.

The type `readonly T` can be thought of as an interface that contains a subset of the methods of `T` (namely, those that do not mutate the object) and that is a supertype of `T`. However, Java interfaces are less powerful and cannot be used in place of the `readonly T` construct; see section 7.2 for details.

Given the type hierarchy shown in figure 3-1, Java's existing type-checking rules enforce that mutable methods cannot be called on read-only references and that the value referenced by a read-only variable cannot be copied to a non-read-only variable (non-conversion).

```
readonly    Date rodate = new Date(); // read-only Date
/*mutable*/ Date date   = new Date(); // mutable Date
```

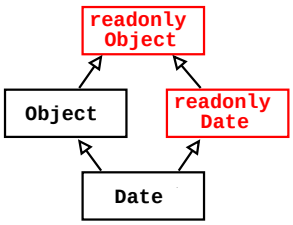

Figure 3-1: A portion of the Javari type hierarchy, which includes read-only and mutable versions of each Java reference type. Arrows connect subtypes to supertypes.

```
rodate = date;                    // OK
rodate = (readonly Date) date;    // OK
date = rodate;                    // error
date = (Date) rodate;             // error: Java cast cannot make a read-only reference
                                  // mutable.  See section 5.3.5 for details.
```

A read-only reference type can be used in a declaration of any variable, field, parameter, or method return type. Local variables, including method parameters and return types, are by default mutable (non-read-only). Primitive types, such as `int`, `float`, and `boolean`, are immutable — they contain no modifiable state. Thus, it is not meaningful to annotate them with `readonly`, and Javari's syntax prohibits it.

Note that `final` and `readonly` are orthogonal notions in a variable declaration. The keyword `final` makes the variable not assignable, but the object it references remains mutable. The keyword `readonly` makes the referenced object non-mutable (through that reference), but the variable may remain assignable. Using both keywords gives a variable whose transitively reachable state cannot be changed, except through a mutable aliasing reference.

3.3 Read-only methods (`this` parameters)

Just as `readonly` may be applied to any explicit formal parameter of a method, it may be applied to the implicit `this` parameter by writing `readonly` immediately following the parameter list. For example, an appropriate declaration for the `StringBuffer.charAt` method in Javari is:

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

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

3.3.1 Overloading

Javari's immutability annotations should not affect the runtime-behavior of programs. This approach allows class files generated by Javari to be backwards compatible and for Javari to be implemented using erasure, as parametric types are implemented in Java 1.5. One consequence of this approach is that two overloaded methods cannot differ only in the mutability of its parameters. For example, the following pair of methods can not overload one another:

```
void foo(/*mutable*/ Date d) { ... }
void foo(readonly    Date d) { ... }
```

This restriction is similar to Java's restriction that overloaded methods may not differ only by type arguments. For example, in Java the following methods may not overload one another:

```
void baz(List<Date>    x) { ... }
void baz(List<Integer> x) { ... }
```

Similarly, in Javari methods cannot be overloaded based on the mutability of the methods' receivers. For example, the following methods may not overload one another:

```
void bar() /*mutable*/  { ... }
void bar() readonly     { ... }
```

3.4 Immutable classes

A class or an interface can be declared to be immutable via the `readonly` modifier in its declaration.[2]

This is a syntactic convenience: the class's non-static fields default to read-only and final, and its non-static methods (and, for inner classes, constructors) default to read-only. As a further syntactic convenience, every reference to an object of immutable type `T` is implicitly read-only. For example, `String` means the same thing as `readonly String` (the latter is forbidden, for syntactic simplicity), and it is impossible to specify the type "`mutable String`".

An immutable class may extend only other immutable classes. Furthermore, subclasses or subinterfaces of immutable classes and interfaces must be immutable.

3.5 Fields

Controlling mutation is a central theme of Javari; therefore, Javari's handling of fields is of key importance to the language design. This section begins by discussing how Javari reaches its goal of providing transitive immutability using fields that default to being this-assignable and this-mutable (section 3.5.1). Next, this section shows how Javari naturally handles field accesses that occur within methods (section 3.5.2). Third, this section shows how Javari reaches its flexibility design goal by providing assignable and mutable fields (section 3.5.3). Finally, this section concludes by demonstrating the details of the Javari type system that ensure fields cannot be used as a mechanism to convert a read-only reference into a mutable reference, thus supporting Javari's non-conversion design goal (section 3.5.4).

3.5.1 This-assignability and this-mutability

One of the design goals of Javari is to provide transitive immutability: a field reached through a read-only reference may not be modified directly nor assigned to a mutable reference. Javari fulfills this goal, by defaulting fields to inherit their assignability and mutability

[2] A "read-only type" is `readonly T`, for some `T`. An "immutable type" is a primitive type, or a class or interface whose definition is marked with `readonly`.

from the mutability of the reference through which the field is reached. These defaults are called *this-assignability* and *this-mutability*, respectively, because the assignability (or mutability) of the field is the same as that of `this`. Transitive immutability is ensured because a this-assignable, this-mutable field read through a read-only reference is unassignable and read-only. The same field read through a mutable reference is assignable and mutable, however. The behavior of this-assignable and this-mutable fields is illustrated below.

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

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

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

 c.d.setYear(2005);   // OK: c.d is mutable
rc.d.setYear(2005);   // error: rc.d is read as read-only
```

This-assignability and this-mutability can only be applied to fields. Fields default to this-assignable, but this can be overridden by declaring a field to be final (section 3.1) or assignable (section 3.5.3). Fields default to this-mutable, but this can be overridden by the declaring a field to be read-only (section 3.2) or mutable (section 3.5.3). This-assignable, final, and assignable are the three kinds of assignabilities in Javari, and this-mutable, read-only, and mutable are the three kinds of mutabilities.

Section 3.5.4 gives additional information about assignable (section 3.5.3), this-mutable fields.

3.5.2 Field accesses within methods

No special rules are needed for handling field accesses within method bodies. Within a read-only method, the declared type of `this` is read-only; therefore, all this-mutable fields are read as read-only and all this-assignable fields are unassignable. Within a mutable method, the declared type of `this` is mutable; therefore, all this-mutable fields are mutable and all this-assignable fields are assignable. These rules are demonstrated below.

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

  /*mutable*/ Date foo() readonly { // this is readonly
    d = new Date();  // error: this.d is unassignable
    d.setYear(2005); // error: this.d is read as readonly
    return d;        // error: this.d is read as readonly
  }

  /*mutable*/ Date bar() /*mutable*/ { // this is mutable
    d = new Date();  // OK: this.d is assignable
    d.setYear(2005); // OK: this.d is mutable
    return d;        // OK: this.d is mutable
  }
}
```

3.5.3 Assignable and mutable: excluding fields from the abstract state

Javari gives programmers the flexibility to exclude fields from the abstract state of an object by using the `assignable` and `mutable` keywords. By default, fields are this-assignable and this-mutable. Under these defaults, all the fields are considered to be a part of the object's abstract state and, therefore, can not be modified through a read-only reference. The `assignable` and `mutable` keywords override these defaults by excluding specific fields from an object's abstract state. `assignable` and `mutable` can also be used when the relationship between the object's abstract state and a field is too complicated to be captured by the Javari type system.

`assignable` fields

Declaring a field `assignable` specifies that the field can always be reassigned, even through a read-only reference. This can be useful for caching or specifying that the identity of a field is not a part of the object's abstract state. For example, `hashCode` is a read-only method: it does not modify the abstract state of the object. In order to cache the hash code, a programmer can use the `assignable` keyword as shown below.

```
class Foo {
  assignable int hc;
  int hashCode() readonly {
    if (hc == 0) {
      hc = ... ;       // OK: hc is assignable
    }
    return hc;
  }
}
```

`mutable` fields

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

```
class Foo {
  /** A Log of the method call history. */
  final mutable List<String> log;

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

If `log` had the default this-mutability, the mutation of `log` would be a compile-time error because within the body of a read-only method `log` would be read as a read-only reference.

3.5.4 Rule for this-mutable lvalues

The Javari language must ensure that a read-only reference cannot be converted to a mutable reference, which would subvert the type system. This section discusses how Javari ensures

that a field is not used to undertake such a conversion. In short, in Javari a read-only reference can not be assigned to a this-mutable field (even if reached through a read-only reference), because the this-mutable field can be read as mutable when reached through a mutable reference.

Suppose, by contrast, that the type of a this-mutable field, reached through a read-only reference, was read-only. Then one could use an assignable, this-mutable field to convert a read-only reference to a mutable reference as shown below. (In the unsound type system, the compile-time error shown below would not occur.)

```
class DateConverter {
  assignable /*this-mutable*/ Date f;
}

readonly   Date rd = ...;                       // Start with a readonly Date.
/*mutable*/ DateConverter mutView = ...;
readonly    DateConverter  roView = mutView; // Create a read-only alias.
roView.f = rd;              // error: but would be allowed if lvalue type was read-only.
/*mutable*/ Date md = mutView.f;      // Created a mutable reference to the Date.
```

Javari prevents the above loop-hole by not allowing read-only references to be written to a this-mutable field.

While a this-mutable field, reached through a read-only reference, is read as read-only, it would be incorrect to say that its type is read-only. As an rvalue (an expression in a value-expecting context [1]), such a reference is read-only — it may be *assigned to* a read-only reference but not a mutable reference. However, as an lvalue (an expression in a location-expecting context, such as on the left side of an assignment [1]), such a reference is mutable — it may be *assigned with* a mutable reference but not a read-only reference. For example,

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

readonly Cell rc;

readonly    Date rd;
/*mutable*/ Date md;

rd = rc.d; // OK: rd and rc.d's rvalue are both readonly     Date
rc.d = md; // OK: rc.d's lvalue and md are both /*mutable*/ Date
md = rc.d; // error
rc.d = rd; // error
```

For a subtyping relationship to exist between two classes, the rvalue types of the fields[3] must be covariantly related while the lvalue types of the fields must be contravariantly related.[4] More concretely, if a field of the mutable version of a type has a mutable lvalue type, then the lvalue type of the corresponding field in the read-only version of the type

[3]Note that this discussion is orthogonal to field shadowing, which involves two *distinct* fields (of the same name) in classes that have a subtyping relationship.

[4]Java normally imposes the even stricter requirement that field types must be invariantly related. Javari loosens this rule only for a field's mutability. Without doing so, Javari would be unable to provide both mutable and read-only references to the same object.

```
class Quadrilateral {
  /*this-assignable*/ /*this-mutable*/ Line side1;
  /*this-assignable*/ /*this-mutable*/ Line side2;
  ...

  // Calculated if needed.
  assignable /*this-mutable*/ Line longestSide;

  readonly Line getLongestSide() readonly {
    if (side1.longer(side2) && side1.longer(side3) ... ) {

      longestSide = side1; // OK: this-mutable fields of the same object

    } else if (...) {
     ...
    }
     ...
  }
}
```

Figure 3-2: An assignable, this-mutable field may be assigned with other this-mutable fields of the same object, even within read-only methods.

must also be mutable. Otherwise, the lvalue types would not be contravariantly related, and the above type system loophole would allow one to convert a read-only reference into a mutable reference.

Note, that this-mutable fields may always be assigned to other this-mutable fields within the *same* object. For an example, see figure 3-2.

3.6 Generic classes

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

Javari uses the same rules, extended in the natural way to account for the fact that Javari types include a mutability specification (figure 3-1). A use of a type parameter within the generic class body has the exact mutability with which the type parameter was instantiated.

The rest of this section demonstrates how parametric types in Javari achieve Javari's design goals. First, section 3.6.1 gives basic examples of Javari's parameterized types and demonstrates the intuitive nature of Javari's type arguments. Section 3.6.2 shows how Javari provides the flexibility to exclude the type arguments of a field from the abstract state of an object. Section 3.6.3 explains why allowing one to modify the mutability of a type variable results in violating the non-conversion design goal. Section 3.6.4 shows how Javari ensures that parametric types can not be used to violate the non-conversion design goal.

3.6.1 Type arguments

As with any local variable's type, type arguments to the type of a local variable may be mutable (by default) or read-only (through use of the `readonly` keyword). Below, four valid local variable, parameterized type declarations of `List` are shown. Note that the mutability of the parameterized type `List` does not affect the mutability of the type argument. Thus the `Date` objects inside of `ld3` are mutable even though `ld3` is `readonly`.

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

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

```
class DateList {
  // 3 read-only lists whose elements have different mutability
  readonly List</*this-mutable*/ Date> lst1;
  readonly List<readonly         Date> lst2;
  readonly List<mutable          Date> lst3;
}
```

As in any other case, the mutability of a type with this-mutability is determined by the mutability of the object in whose declaration it textually appears (not the mutability of the parameterized class in which it might be a type argument[5]). In the case of `DateList` above, the type (`/*this-mutable*/ Date`) of `lst1`'s elements is written inside the `DateList` declaration, so the mutability of `lst1`'s elements is determined by the mutability of the reference to `DateList`, not by the mutability of `lst1` itself. The following example illustrates this behavior.

```
/*mutable*/ DateList mutDateList;
readonly    DateList roDateList;

// The reference through which lst is accessed determines
// the mutability of the elements.

/*mutable*/ Date d;
d = mutDateList.lst1.get(0); // OK: elements are mutable
d = roDateList.lst1.get(0);  // error: elements are read as readonly
```

The advantage of this choice is that all all occurrences of `Date` within a given class have the same meaning, namely `/*this-mutable*/ Date`. (We speculate that it would be confusing to have different meanings for different occurrences of lexically identical text.) It can be viewed as first resolving the meaning of the type arguments (including any `/*this-mutable*/`, and *then* substituting the type arguments into the generic class.

[5]There is no need for a modifier that specifies that the type parameter's mutability should be inherited from the mutability of the parameterized class. The declaration of a reference to a parameterized class specifies both the parameterized class's mutability, and the mutability for the type argument. For example, a `List` where the mutability of the type parameter matches the mutability of the `List`'s `this` type can be declared as: `List<Date>` or as `readonly List<readonly Date>`.

3.6.2 `mutable` type arguments

Javari permits a type argument to be excluded from a class's abstract state. A type argument to a field's (parametric) type may be declared mutable with the `mutable` keyword as shown below.

```
class MutDateList {
  readonly List<mutable Date> lst; // read-only list of mutable items
}
```

In this case, the mutability of the reference to the `MutDateList` object has no effect on the mutability of the elements of `lst`, because they are explicitly declared mutable. (As described in section 3.6.1, the mutability of `lst` has no effect on the mutability of its elements.) The case that the elements of `lst` are declared `readonly` is symmetric. The following example illustrates this behavior.

```
class DateList {
  readonly List<mutable  Date> lstOfMut; // read-only list of mutable items
  readonly List<readonly Date> lstOfRo;  // read-only list of readonly items
}

/*mutable*/ Date d;

/*mutable*/ DateList mutDateList;
readonly    DateList  roDateList;

// The mutability of the elements of lstOfMut is mutable regardless of
// whether the field is reached through a read-only or mutable reference.
d = mutDateList.lstOfMut.get(0); // OK
d =  roDateList.lstOfMut.get(0); // OK

// The mutability of the elements of lstOfRo is read-only regardless of
// whether the field is reached through a read-only or mutable reference.
d = mutDateList.lstOfRo.get(0); // error
d =  roDateList.lstOfRo.get(0); // error
```

3.6.3 Mutability modifiers on type variables

It may be tempting for some programmers to modify type variables with mutability annotations. For example, one may wish to write `mutable X`, where `X` is a type variable. Javari does not allow such types because (1) it is inconsistent with Java's notation of type variables and (2) such types can result in read-only references being converted into mutable references.

In Java, a client of a generic class controls the type that a type variable is instantiated with. However, the generic class's author can place bounds on what types a type variable may be instantiated with. To maintain a intuitive integration with Java, Javari exactly follows these principles. Since the client of a generic class controls the type that a type variable is instantiated with, the author of the generic class should not be allowed to change that type by modifying it with mutability modifiers. The author of the generic class can still place bounds on the mutability of a type variable, however. For example, in the class declaration below, the author of the class allows `X` to range over all `Dates` while `Y` can only range over mutable `Dates`.

```
class Foo<X extends readonly Date, Y extends mutable Date> { ... }
```

Additionally, allowing type variables to be modified with `mutable` allows one to convert read-only references to mutable references. For the demonstration of this loophole, assume that one could write `mutable X`, where `X` is a type variable. If `X` was instantiated with a read-only type, an illegal downcast from a read-only type to a mutable type would occur. The type loophole is demonstrated below:

```
class Convert<X extends readonly Object> {
   X f;

   /*mutable*/ Object convertF() {
     mutable T val = f;
     return val;
   }
}
```

3.6.4 `? readonly`

Java wildcards permit abstraction over the type parameter of a generic class. For instance, `List<?>` represents the type of all Lists, regardless of their element type; `List<? extends Number>` represents the type of all Lists whose element type is `Number` or a subtype thereof; and `List<? super Integer>` represents the type of all Lists whose element type is `Integer` or a supertype (namely, `Number`, `Object`, `Serializable`, or `Comparable<Integer>`).

Likewise, Javari contains a wildcard mechanism that abstracts over the mutability of a type parameter. Javari's wildcard mechanism permits methods to be written that can handle generics that have either `readonly` or `mutable` elements. In particular, `? readonly C` is shorthand for `? extends readonly C super mutable C`, which is a wildcard type with both an upper and a lower bound. Wildcards can only appear at type arguments, and may not be used in any other location.

This section discusses how we prevent parametric types from being used to convert read-only references into mutable references. Similar to the case of a top-level type (section 3.5.4), a this-mutable type argument reached through a read-only reference has different rvalue and lvalue types. Such a type argument's rvalue (how it is read) must be read-only to provide the transitive immutability provision. However, the type argument's lvalue (how it is written to) must be mutable. If the lvalue was read-only a type loophole similar to the one shown earlier in section 3.5.4 would exist. For an example of the different reading and writing behaviors of a this-mutable type argument, see lines 16 through 20 of figure 3-3.

A type that is written as mutable and read as read-only can be represented using wildcards that are bounded from above (`extends`) and below (`super`). For example, the type of `roBar.c` (figure 3-3) can be written as:

```
readonly Cell<? extends readonly Date super mutable Date>
```

The upper bound on the wildcard is `readonly Date` while the lower bound is `mutable Date`. Note that the bounds on the wildcard only differs in terms of mutability; Javari does not need arbitrary two-sided wildcards.

Since Java does not allow a wildcard to have both an upper bound and a lower bound, we use the `? readonly` syntax to represent two-sided wildcards that differ (only) in terms of mutability.[6] We will also use `? readonly` types as meta-syntax to represent top-level types

[6] `? readonly T` can be thought of as the bounded existential type, $\exists$ `X extends readonly T super mutable T`. This interpretation is expanded on in the core calculus; see chapter 4.

```
1   class Bar {
2     assignable Cell</*this-mutable*/ Date> c;
3   }
4
5   class Cell<T extends readonly Object> {
6     assignable T val;
7   }
8
9   // roBar.c's rvalue type is readonly Cell<? readonly Date>
10  // roBar.c's lvalue type is mutable Cell<mutable Date>
11  readonly Bar roBar;
12
13  readonly    Date rd;
14  /*mutable*/ Date md;
15
16  rd = roBar.c.val; // OK: roBar.c.val is read as read-only
17  md = roBar.c.val; // error: roBar.c.val is read as read-only
18
19  roBar.c.val = rd; // error: elems are written to as mutable
20  roBar.c.val = md; // OK: elems are written to as mutable
21
22
23  // cell.val's rvalue type is readonly Date
24  // cell.val's lvalue type is mutable Date
25  readonly Cell<? readonly Date> cell = roBar.c;
26
27  rd = cell.val; // OK: elems are read as readonly
28  md = cell.val; // error: elems are read as readonly
29
30  cell.val = rd; // error: elems are written to as mutable
31  cell.val = md; // OK: elems are written to as mutable
32
33
34  mutable Cell<mutable Date> newCell;
35
36  // roBar.c lvalue type is mutable Cell<mutable Date>
37  roBar.c = newCell; // OK
38  roBar.c = cell;    // error
```

Figure 3-3: Demonstration of a this-mutable field reached through a `readonly` reference. `Bar.c` and `Cell.val` are assignable so that the fields' lvalue types can be demonstrated.

that are read-only when read and mutable when written to. However, as with all Java wildcards, `? readonly` cannot appear as a *top-level* type in source code. For example, from figure 3-3, the type of `roBar.c` can be written as `? readonly List<? readonly Date>`.

Lines 23–31 of figure 3-3 give an example of a `? readonly` type argument. `roBar.c` is shown being used as an rvalue on lines 27–31 of the example and `roBar.c` is used as an lvalue on lines 36–38.

Following the normal wildcard typing rules for Java, `List<? readonly Date>` is a common supertype of `List<readonly Date>` and `List<mutable Date>`. Figure 3-4 shows a portion of the Javari type hierarchy for lists of Dates.

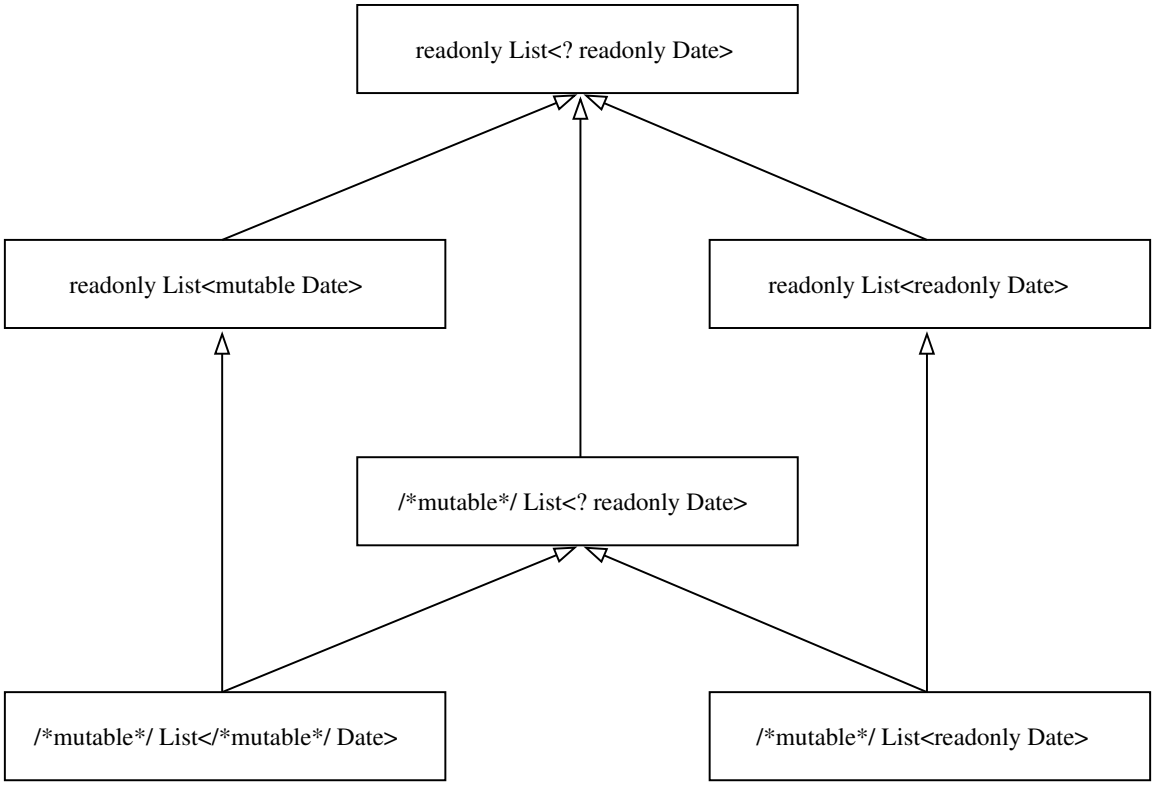

Figure 3-4: The Javari type hierarchy for Lists of Dates. Arrows connect subtypes to supertypes.

```
class Baz {
  /*this-mutable*/ List</*this-mutable*/ Set</*this-mutable*/ Date>> f;
}

readonly Baz rb;
readonly List<? readonly Set<? readonly Date>> lst = rb.f;
```

Figure 3-5: An example of nested this-mutable types.

`? readonly` can also be used in the case of nested parametric types. For example, in figure 3-5, the type

```
readonly List<? readonly Set<? readonly Date>>
```

desugars to

```
readonly List<? extends readonly Set<? extends readonly Date
                                     super /*mutable*/ Date>
              super /*mutable*/ Set</*mutable*/ Date>>
```

From the desugaring, one can see that the list may only have completely mutable types, `/*mutable*/ Set</*mutable*/ Date>>`, added as elements. However, the list's elements are read as completely immutable: `readonly Set<? extends readonly Date super /*mutable*/ Date>>`. The `Set`'s elements are declared to have type `? extends readonly Date super`

`/*mutable*/ Date` to prevent a `readonly Date` from being inserted into the set. A complete desugaring of `? readonly` into two-sided wildcards is given in figure 4-13.

3.7 Arrays

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

```
                  Date [] ad1; // add/remove, mutate
         (readonly Date)[] ad2; // add/remove
readonly          Date [] ad3; // mutate
readonly (readonly Date)[] ad4; // neither
```

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

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

```
(/*mutable*/ Date)[] ad1;
(readonly    Date)[] ad2;

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

As in the case with parametric container classes, arrays may use the `? readonly` syntax. For example,

```
class Foo {
  /*this-mutable*/ (/*this-mutable*/ Date)[] dates;
}

readonly Foo rf;
readonly (? readonly Date)[] x = rf.dates;
```

3.8 Summary

Javari enables a programmer to independently declare the assignability and mutability of a reference. Figure 3-6 summarizes the assignability and mutability keywords of Javari:

`final` declares a reference to be unassignable.
`assignable` declares a reference always to be assignable even if accessed through a read-only reference. Redundant for references other than instance fields.
`readonly` declares a reference to be read-only. Redundant for immutable types.

Construct	Assignability			Mutability		
	assignable	unassign-able	this-assign.	mutable	read-only	this-mutable
Instance fields	`assignable`	`final`	(default)	`mutable`	`readonly`	(default)
Static fields	(default)	`final`	N/A	(default)	`readonly`	N/A
Local variables	(default)	`final`	N/A	(default)	`readonly`	N/A
Formal param.s	(default)	`final`	N/A	(default)	`readonly`	N/A
Return values	N/A	N/A	N/A	(default)	`readonly`	N/A
`this`	N/A	(default)	N/A	(default)	`readonly`	N/A

Figure 3-6: Javari's keywords. Each cell contains the keyword that achieves a particular assignability or mutability for a given variety of reference. "N/A" denotes assignabilities or mutabilities that are not valid for a given construct. "(default)" denotes that a given assignability or mutability is the default for that construct; no keyword is required, and redundant use of keywords is prohibited (a compile-time error), in order to reduce confusion. This-assignable and this-mutable can only be applied to instance fields because other references do not have a notion of `this` to inherit from. The mutability of `this` is declared after the parameter list.

Resolved assignability of `a.b`

Declared assignability of `b`	Resolved mutability of `a`	
	mutable	read-only
assignable	assignable	assignable
unassignable	unassignable	unassignable
this-assignable	assignable	unassignable

Resolved mutability of `a.b`

Declared mutability of `b`	Resolved mutability of `a`	
	mutable	read-only
mutable	mutable	mutable
read-only	read-only	read-only
this-mutable	assignable	`? readonly`

Figure 3-7: Semantics of Javari's keywords: resolved type of the expression `a.b`, given the resolved type of `a` and the declared type of field `b`. Also see figure 4-12, which presents the same information in a different form.

`mutable` declares a reference always to be mutable even if accessed through a read-only reference. Redundant for references other than instance fields. Cannot be applied to type parameters (see section 3.6.3).

`? readonly` declares a type argument to have a read-only upper bound and a mutable lower bound.

Figure 3-7 briefly gives the semantics of the keywords. Appendix A gives examples of each of the 9 combinations of assignability and mutability.

Javari is backward compatible with Java: any Java program that uses none of Javari's keywords is a valid Javari program, with the same semantics. Javari's defaults have been chosen to ensure this property.

Chapter 4

Type Rules

This chapter presents the key type rules for Javari in the context of a core calculus, Lightweight Javari, that captures the essential features of Javari. The core language allows one to examine the underlying structure of Javari's type system without being overwhelmed with the complexity of the full language. Additionally, the core language can be used to prove properties about the language.

Lightweight Javari builds upon Wild FJ (WFJ) [47], a core calculus for Java including generic types. WFJ extends Featherweight Generic Java (FGJ) [23], with wildcard types. WFJ is a functional language: it has no notion of assignment. Therefore, Section 4.1 first introduces Lightweight Java (LJ), an extension of WFJ that builds on ClassicJava [18] to support field assignment and the `final` keyword. Then, Section 4.2 extends LJ to Lightweight Javari (LJR), which adds support for reference immutability.

Before presenting our core languages, we explain, below, how WFJ models capture conversion and wildcard capture, two of the most complex aspects of Java's type system.

Java's capture conversion makes wildcards more useful by replacing top-level type argument wildcards with globally fresh type variables. To understand the utility of this technique, consider the method definition and invocation shown in figure 4-1. It appears that the method `copy` can not be used on list `x` because `x`'s type, `List<?>`, is not a subtype of `List<T>`. An explanation for why a list with a wildcard type card type argument, e.g., `List<?>`, is not subtype of a list with an invariant type argument, e.g., `List<T>` is given in figure 4-2. Using capture conversion, one can convert `x`'s type from `List<?>` to `List<X>` where `X` is a fresh type variable. Since `List<X>` is a subtype of `List<T>` (when `T` is inferred to be `X`), the invocation of `copy(x)` succeeds. Note that capture conversion does not allow the method invocation shown in figure 4-2 because `String` is not a subtype of the type variable created by capture converting `List<?>`.

```
/** Returns a shallow copy of lst */
static <T extends Object> List<T> copy(List<T> lst) { ... }

List<?> x;
copy(x);    // Disallowed without capture conversion.
```

Figure 4-1: Without capture conversion, the type-safe method invocation of `copy(x)` would be disallowed because, without conversion, `List<?>` is not a subtype of `List<T>` (as explained in figure 4-2).

```
 /** Adds val to lst. */
static <T extends Object> void insert(List<T> lst, T val) {
  lst.add(val);
}

List<Date> lstOfDates;
List<?>    lstOfUnknowns = lstOfDates; // OK
insert(lstOfUnknowns, ``hi'');         // error: Would be legal if List<?> was
                                       // a subtype of List<T>
```

Figure 4-2: A demonstration of why it would be unsound for `List<?>` to be a subtype of `List<T>` where `T` is any type variable. In the code above, allowing `lstOfUnknown`, which has type `List<?>`, to be passed as a `List<T>` to method `insert` results a `String` being inserted into a list of dates, `lstOfDates`.

While implementing capture conversion by creating globally fresh type variables is a realist strategy for a compiler, it conflicts with the compositionality of the type rules. Therefore, WFJ models capture conversion using existential types. An existential type, written as $\exists\Delta.\mathtt{K}$, consists of (1) a type environment, Δ, which contains a mapping from the the fresh type variables to their bounds of, and (2) a type, `K`, which is free of any top-level wildcards, `K`. The typing of an expression (section 4.1.3) returns a existential type where the type environment contains any fresh type variables that were created during capture conversion, and the top-level wildcard free type represents the type of the expression. Existential types only serve as meta-syntax and does not appear in the source language. WFJ's implementation of capture conversion is given in section 4.1.2.

A concept related but distinct from capture conversion is wildcard capture. In the example given in figure 4-1, we did not discuss how the method invocation inferred that `T`, the method's type parameter, should be instantiated with `X`, the type argument of `x`'s of capture-converted type. This process is called wildcard capture. Normally, in WFJ, a method invocation must supply a type argument for each of the method's type parameters. For example, in the method invocation shown below, the type variable, `T`, of `copy` is explicitly instantiated with the type `Integer`.

```
// Same method as from figure 4-1.
static <T extends Object> List<T> copy(List<T> lst) { ... }

List<Integer> integers = ... ;
List<Integer> myCopy = <Integer>copy(integers);
```

However, since the fresh type variables created by capture conversion can not be referred to by the source code of a program, it is impossible to supply a correct type argument. For example, below, it is impossible for the method invocation to supply the correct type argument.[1]

```
List<?> lstOfUnknowns = ... ;
List<?> myCopy = </*Unknown fresh type variable*/>copy(lstOfUnknowns);
```

To solve this dilemma, a programmer is allowed to pass the special symbol $\star$ as the type argument to such a method invocation. $\star$ serves as a request for wildcard capture.

[1]Note that is it incorrect to use `?` as the type argument to `copy` because `?` is not a type.

Wildcard capture allow capture converted types to serve as type arguments to methods by inferring the correct type arguments to a method invocation. The type inference is implemented by examining the types of actual arguments to a method to determine the type arguments of the method invocation. Since the types of the method's actual arguments have already been capture converted, wildcard capture is capable of inferring the fresh type variables that were created by capture conversion. WFJ's implementation of wildcard capture is given in section 4.1.2.

4.1 Lightweight Java

Lightweight Java (LJ) extends WFJ to include field assignment, expressed via the "`set`" construct. Fields may be declared `final`; to make LJ's syntax fully explicit, non-final fields must be declared `assignable`. LJ does not permit parameters to be reassigned: such an extension is straightforward and does not demonstrate any interesting aspects of the Javari type system. LJ omits casts because they are not important for demonstrating the assignability and mutability rules of Javari.

4.1.1 Syntax

The syntax of LJ is largely identical to that of WFJ, with the exception of the new `set` expression and the assignability modifiers, `final` and `assignable`. The syntax of LJ is shown in figure 4-3.

The metavariable `C` ranges over (unparameterized) class names; `X`, `Y`, and `Z` over type variables; `m` over method names; `f` over field names; and `x` over formal parameter names. `Q` ranges over class declarations; `M` over method declarations; and `e` over expressions. `AF` (assignability for fields) ranges over assignability modifiers.

`S`, `T`, `U`, and `V` range over types, which may be either a parameterized type, `C<`$\overline{\texttt{A}}$`>`, or a type variable, `X`. `A` ranges over type arguments to parametric types, which may be either a type, `T`, or a wildcard, ? `B` or ? `readonly T`. `B` ranges over type bounds and includes both an upper bound, $\texttt{B}_{\triangleleft}$, and a lower bound $\texttt{B}_{\triangleright}$. An upper (lower) bound is either an extends (super) cause, $\triangleleft$ `T` ($\triangleright$ `T`), or is absent, $\bullet$. The notation $\triangleleft$ stands for "extends" (LJ has no interfaces), and $\triangleright$ stands for "super". Parametric method invocations must be provided an explicit type argument, `P`, which is either a type, `T`, or a special marker, $\star$, that is resolved by wildcard capture. `N` ranges over class types, which are nonvariable types which do not have wildcards as a top-level type arguments. `K` ranges over the union on of class types, `N`, and type variables, `X`.

$\overline{\texttt{x}}$ serves as shorthand for the (possibly empty) sequence $\texttt{x}_1 \ldots \texttt{x}_n$ with the appropriate syntax separating the elements. In cases such as $\overline{\texttt{T}}$ $\overline{\texttt{f}}$, the items are grouped in pairs: $\texttt{T}_1$ $\texttt{f}_1 \ldots \texttt{T}_n$ $\texttt{f}_n$. The empty sequence is written as $\bullet$, and sequences are concatenated with a comma.

[a/b]c denotes the result of replacing `b` by `a` in `c`.

`this` is considered a special variable implicitly bound to the receiver of a method invocation.

A class is considered to have all the fields that are declared in its body and its superclasses' bodies. The field names of a class must be distinct from the field names of its superclasses — there is no field shadowing. Additionally, there is no method overloading.

WFJ does not model constructors because they do not demonstrate any interesting properties of the type system.

T, S, U, V	::=	C<A> | X	*types*
A	::=	T | ? B| ? readonly T	*type arguments*
P	::=	T | ⋆	*method type parameters*
N	::=	C<$\overline{\text{T}}$>	*class types*
K	::=	N | X	*class types and var.s*
B	::=	$\text{B}_\triangleleft$ $\text{B}_\triangleright$	*bounds*
$\text{B}_\triangleleft$	::=	$\triangleleft$ T | •	*upper bounds*
$\text{B}_\triangleright$	::=	$\triangleright$ T | •	*lower bounds*
Q	::=	class C<$\overline{\text{X}}$ $\overline{\text{B}}$> $\triangleleft$ N {$\overline{\text{AF}}$ $\overline{\text{T}}$ $\overline{\text{f}}$; $\overline{\text{M}}$}	*class declaration*
M	::=	<$\overline{\text{X}}$ $\overline{\text{B}}$> T m($\overline{\text{T}}$ $\overline{\text{x}}$){ return e; }	*method declarations*
e	::=	x	*expressions*
	|	e.f	
	|	e.m<$\overline{\text{P}}$>($\overline{\text{e}}$)	
	|	new N($\overline{\text{e}}$)	
	|	set e.f = e then e	
AF	::=	final | assignable	*assignabilities*
E, F	::=	$\exists\Delta$.K	*existential types*
Δ	::=	$\emptyset$ | Δ, X $\in$ B	*type environments*
C		*class names*	
X, Y, Z		*type variables*	
m		*method names*	
f		*field names*	
x		*variables*	

Figure 4-3: Syntax of Lightweight Java (LJ). Changes from WFJ are shown in boxes.

LJ introduces the `set` construct to FGJ. "`set` e_0`.f =` e_v `then` e_b" reassigns the field `f` of the object to which e_0 evaluates. The field's new value is the value to which e_v evaluates. The `set` expression then evaluates the body expression e_b. The `set` expression's value is the value to which e_b evaluates. The `set` syntax was chosen to avoid the complications of allowing multiple expressions within a method. Method arguments and assignment expressions are evaluated left-to-right (see figure 4-10).

4.1.2 Auxiliary functions

Bound calculation

The subtyping rules use the following auxiliary functions to calculate the bounds of type variables. $bound^1_\Delta(\text{X})$ calculates the upper bound of X in type environment Δ. If the upper bound is absent, `Object` is returned. $lbound^1_\Delta(\text{X})$, returns the lower bound of X in Δ. *lbound* is undefined if the lower bound is absent. $bound_\Delta(\text{T})$ when T is a type variable, X, recursively applies $bound^1$ until a nonvariable upper bound of X is found. In the case that T is the nonvariable type C<$\overline{\text{A}}$>, *bound* returns that type, C<$\overline{\text{A}}$>. See figure 4-4 for the definitions of the bound calculation methods. For convenience, *bound* is also defined on existential types, E; see below.

bound^1

$$\frac{\Delta(\mathtt{X}) = \bullet\ \mathtt{B}_{\triangleright}}{bound^1_\Delta(\mathtt{X}) = \mathtt{Object}} \qquad \frac{\Delta(\mathtt{X}) = \triangleleft\ \mathtt{T}\ \mathtt{B}_{\triangleright}}{bound_\Delta(\mathtt{X}) = \mathtt{T}}$$

lbound

$$\frac{\Delta(\mathtt{X}) = \mathtt{B}_{\triangleleft}\ \triangleright\ \mathtt{T}}{lbound^1_\Delta(\mathtt{X}) = \mathtt{T}}$$

bound

$$\frac{bound^1_\Delta(\mathtt{X}) = \mathtt{C<}\overline{\mathtt{T}}\mathtt{>}}{bound_\Delta(\mathtt{X}) = \mathtt{C<}\overline{\mathtt{T}}\mathtt{>}} \qquad \frac{bound^1_\Delta(\mathtt{X}) = \mathtt{Y} \qquad bound_\Delta(\mathtt{Y}) = \mathtt{C<}\overline{\mathtt{T}}\mathtt{>}}{bound_\Delta(\mathtt{X}) = \mathtt{C<}\overline{\mathtt{T}}\mathtt{>}} \qquad bound_\Delta(\mathtt{C<}\overline{\mathtt{T}}\mathtt{>}) = \mathtt{C<}\overline{\mathtt{T}}\mathtt{>}$$

$$\frac{bound_{\Delta\Delta'}(\mathtt{K}') = \mathtt{C<}\overline{\mathtt{T}}\mathtt{>} \qquad snap(\mathtt{C<}\overline{\mathtt{T}}\mathtt{>}) = \exists\Delta''.\mathtt{N}}{bound_\Delta(\exists\Delta'.\mathtt{K}') = \exists\Delta'\Delta''.\mathtt{N}}$$

Figure 4-4: Bound calculation functions. These functions are unchanged from those of WFJ.

Existential type creation: Snap

The auxiliary functions of figure 4-5 are used to create existential types. Existential types are used to model the capture conversion of wildcards (see the introduction of this chapter). Through the use of existential types the subtyping rules are simplified and do not need to directly deal with wildcards. *snap*(T) creates an existential type, $\exists\Delta.\mathtt{K}$, from a type, T. *snap* replaces each top-level wildcard, ? B, with a fresh type variable, which is added to the type environment of the existential type. For example, $snap(\mathtt{C<?\ B}_{\triangleleft}\ \mathtt{B}_{\triangleright}\mathtt{>}) = \exists\mathtt{X} \in \mathtt{B}_{\triangleleft}\ \mathtt{B}_{\triangleright}.\mathtt{C<X>}$ where X is fresh. *snap* uses $fix(\overline{\mathtt{A}}, \overline{\mathtt{B}})$ to produce the new type variables, including bounds, for each wildcard type. *merge* is used by *snap* to combine the wildcard's bounds with the declared bound of the relevant type parameter.

Wildcard capture

Wildcard capture is used to allow capture converted types to be passed as type arguments to method invocations (see the introduction of this chapter). Wildcard capture enables this action by inferring the correct type arguments of a method invocation when those types cannot be expressed in the source language. The function $capture_\Delta(\mathtt{P}, \mathtt{X}, \overline{\mathtt{T}}, \overline{\mathtt{K}})$ of figure 4-6 is used to perform wildcard capture on a method invocation where P is the actual type parameter to a method. In the case that P is a type, T, then wildcard capture is not needed and P is returned unchanged. However, if P is a type capture marker, $\star$, then wildcard capture is performed. X is the type parameter from the signature of the method being invoked. $\overline{\mathtt{T}}$ is the parameter types from the signature of the method. $\overline{\mathtt{K}}$ is the types of the actual arguments to the method. The definition of *capture* is given in figure 4-6.

Snap:

$$snap(\mathtt{X}) = \exists\emptyset.\mathtt{X}$$

$$\frac{\mathtt{class\ C<}\overline{\mathtt{Y}}\ \overline{\mathtt{B_0}}\mathtt{>} \lhd \mathtt{N}\ \{\ldots\} \qquad fix(\overline{\mathtt{A}}, \overline{\mathtt{B_0}}) = (\overline{\mathtt{T}}, \overline{\mathtt{X}}, \overline{\mathtt{B}}) \qquad \Delta = \overline{\mathtt{X}} \in [\overline{\mathtt{T}}/\overline{\mathtt{Y}}]\overline{\mathtt{B}}}{snap(\mathtt{C<}\overline{\mathtt{A}}\mathtt{>}) = \exists\Delta.\mathtt{C<}\overline{\mathtt{T}}\mathtt{>}}$$

Fix:

$$\frac{fix(\overline{\mathtt{A}}, \overline{\mathtt{B_0}}) = (\overline{\mathtt{T}}, \overline{\mathtt{X}}, \overline{\mathtt{B}})}{fix(\mathtt{T} :: \overline{\mathtt{A}}, \mathtt{B_0} :: \overline{\mathtt{B_0}}) = (\mathtt{T} :: \overline{\mathtt{T}}, \overline{\mathtt{X}}, \overline{\mathtt{B}})}$$

$$\frac{fix(\overline{\mathtt{A}}, \overline{\mathtt{B_0}}) = (\overline{\mathtt{T}}, \overline{\mathtt{X}}, \overline{\mathtt{B}}) \qquad \mathtt{X\ fresh}}{fix(?\ \mathtt{B} :: \overline{\mathtt{A}}, \mathtt{B_0} :: \overline{\mathtt{B_0}}) = (\mathtt{X} :: \overline{\mathtt{T}}, \mathtt{X} :: \overline{\mathtt{X}}, merge(\mathtt{B}, \mathtt{B_0}) :: \overline{\mathtt{B}})}$$

$$fix(\bullet, \bullet) = (\bullet, \bullet, \bullet)$$

Merge:

$$merge(\bullet\ \bullet, \mathtt{B}_\lhd\ \mathtt{B}_\rhd) = \mathtt{B}_\lhd\ \mathtt{B}_\rhd \qquad merge(\lhd\ \mathtt{T}\ \bullet, \mathtt{B}_\lhd\ \mathtt{B}_\rhd) = \lhd\ \mathtt{T}\ \mathtt{B}_\rhd$$

$$merge(\bullet\ \rhd\ \mathtt{S}, \mathtt{B}_\lhd\ \mathtt{B}_\rhd) = \mathtt{B}_\lhd\ \rhd\ \mathtt{S} \qquad merge(\lhd\ \mathtt{T}\ \rhd\ \mathtt{S}, \mathtt{B}_\lhd\ \mathtt{B}_\rhd) = \lhd\ \mathtt{T}\ \rhd\ \mathtt{S}$$

Figure 4-5: Auxiliary functions for existential type creation. These functions are unchanged from those of WFJ.

Class member lookup

The typing and reduction rules use the following auxiliary functions. The function *fields*(C<$\overline{\mathtt{T}}$>) returns a sequence of triplets, $\overline{\mathtt{AF}}\ \overline{\mathtt{T}}\ \overline{\mathtt{f}}$, with the assignability modifier, type, and name of each of the fields of the nonvariable type C<$\overline{\mathtt{T}}$>.

The function *mtype*(m, C<$\overline{\mathtt{T}}$>) (figure 4-7) returns the type of method m of the nonvariable type C<$\overline{\mathtt{T}}$>. The type of a method is written as <$\overline{\mathtt{X}}$ $\overline{\mathtt{B}}$>$\overline{\mathtt{U}}$ → U where $\overline{\mathtt{X}}$, with the bounds $\overline{\mathtt{B}}$, are the type parameters of the method, $\overline{\mathtt{U}}$ are the types of the method's parameters, and U is the return type of the method. Because there is no overloading in LJ, *mtype* does not need to know the types of the arguments to m.

The function *mbody*(<$\overline{\mathtt{V}}$>m, C<$\overline{\mathtt{T}}$>) returns the pair $\overline{\mathtt{x}}$.e where $\overline{\mathtt{x}}$ are the formal parameters to m in C<$\overline{\mathtt{T}}$> and e is the body of the method.

Finally, *override*(m, N, <$\overline{\mathtt{Y}}$ $\overline{\mathtt{B}}$>$\overline{\mathtt{T}}$ → T) (figure 4-7) declares that method m with type <$\overline{\mathtt{Y}}$ $\overline{\mathtt{B}}$>$\overline{\mathtt{T}}$ → T) correctly overrides any methods with the same name possessed by the nonvariable type N.

Details on these auxiliary functions can be found in [47].

Capture:

$$capture_{\Delta}(\mathtt{U}, \mathtt{X}, \overline{\mathtt{T}}, \overline{\mathtt{K}}) = \mathtt{U}$$

$$\frac{\mathtt{T}_i = \mathtt{C<}\overline{\mathtt{A}}\mathtt{>} \qquad \mathtt{A}_j = \mathtt{X} \qquad \Delta \vdash \mathtt{K}_i <: \mathtt{C<}\overline{\mathtt{A'}}\mathtt{>} \qquad \mathtt{A}'_j = \mathtt{V}}{capture_{\Delta}(\star, \mathtt{X}, \overline{\mathtt{T}}, \overline{\mathtt{K}}) = \mathtt{V}}$$

Figure 4-6: Wildcard capture auxiliary function. These functions are unchanged from those of WFJ.

Method type lookup:

$$\frac{\mathtt{class\ C<}\overline{\mathtt{X}}\ \overline{\mathtt{N}}\mathtt{>} \lhd \mathtt{B}\{\overline{\mathtt{AF}}\ \overline{\mathtt{T}}\ \overline{\mathtt{f}};\ \overline{\mathtt{M}}\} \qquad \mathtt{<}\overline{\mathtt{Y}}\ \overline{\mathtt{B'}}\mathtt{>\ U\ m(}\overline{\mathtt{U}}\ \overline{\mathtt{x}}\mathtt{)\ ML\ \{\ return\ e;\ \}} \in \overline{\mathtt{M}}}{mtype(\mathtt{m}, \mathtt{C<}\overline{\mathtt{T}}\mathtt{>}) = [\overline{\mathtt{T}}/\overline{\mathtt{X}}](\mathtt{<}\overline{\mathtt{Y}}\ \overline{\mathtt{B'}}\mathtt{>}\overline{\mathtt{U}}\ \rightarrow \mathtt{U})}$$

$$\frac{\mathtt{class\ C<}\overline{\mathtt{X}}\ \overline{\mathtt{N}}\mathtt{>} \lhd \mathtt{B}\{\overline{\mathtt{AF}}\ \overline{\mathtt{T}}\ \overline{\mathtt{f}};\ \overline{\mathtt{M}}\} \qquad \mathtt{m} \notin \overline{\mathtt{M}}}{mtype(\mathtt{m}, \mathtt{C<}\overline{\mathtt{T}}\mathtt{>}) = mtype(\mathtt{m}, [\overline{\mathtt{T}}/\overline{\mathtt{X}}]\mathtt{N})}$$

Valid method overriding:

$$\frac{\begin{array}{c} mtype(\mathtt{m}, \mathtt{N}) = \mathtt{<}\overline{\mathtt{Y'}}\ \overline{\mathtt{B'}}\mathtt{>}\overline{\mathtt{T'}}\ \mathtt{ML'} \rightarrow \mathtt{T'} \text{ implies} \\ \overline{\mathtt{B}} = [\overline{\mathtt{Y}}/\overline{\mathtt{Y'}}]\overline{\mathtt{B'}} \text{ and } \overline{\mathtt{T}} = [\overline{\mathtt{Y}}/\overline{\mathtt{Y'}}]\overline{\mathtt{T'}} \text{ and } \overline{\mathtt{Y}} \in \overline{\mathtt{B}} \vdash \mathtt{T} <: [\overline{\mathtt{Y}}/\overline{\mathtt{Y'}}]\mathtt{T'} \end{array}}{override(\mathtt{m},\ \mathtt{N},\ \mathtt{<}\overline{\mathtt{Y}} \lhd \overline{\mathtt{B}}\mathtt{>}\overline{\mathtt{T}}\ \mathtt{ML} \rightarrow \mathtt{T})}$$

Figure 4-7: Lightweight Java (LJ) auxiliary functions. These functions are unchanged from those of WFJ.

4.1.3 Static semantics

Subtyping

WFJ subtyping and typing rules operate over existential types, $\mathtt{E} ::= \exists\Delta.\mathtt{K}$. Δ is the type environment, a mapping from type variables, X, to bounds, B. Although used in the static semantics, existential types cannot appear in an WFJ program's source. The use of existential types is needed for modeling wildcard capture conversion.

The (reflexive, transitive) subtyping relationship is denoted by "<:". The subtyping rules of LJ are unchanged from WFJ and are shown in figure 4-8. The subtyping rules operate on existential types, E; however, WS-Syntactic is provided as a convenient short hand for applying the subtyping rules to types, T.

Typing judgements

LJ's typing rules calculate existential types for expressions. Γ is defined as the environment, a mapping from variables to types, T. The typing rules are shown in figure 4-9.

The judgment $\Delta \vdash \mathtt{A\ ok}$ declares that the type or wildcard A is well-formed under context Δ. A type is well-formed if its type parameters respect the bounds placed on them in the class's declaration. A wildcard is well-formed if its bounds are well formed. The judgment M OK IN C declares that method declaration M is sound in the context of class C.

Subtyping:

$$\frac{\Delta \ \vdash \ \mathtt{E} <: \mathtt{E}'' \qquad \Delta \ \vdash \ \mathtt{E}'' <: \mathtt{E}'}{\Delta \ \vdash \ \mathtt{E} <: \mathtt{E}'} \text{ (WS-TRANS)}$$

$$\Delta \ \vdash \ \exists\Delta'.\mathtt{X} <: \exists\Delta'.\mathit{bound}^1_{\Delta\Delta'}(\mathtt{X}) \text{ (WS-VAR)}$$

$$\Delta \ \vdash \ \exists\Delta'.\mathit{lbound}^1_{\Delta\Delta'}(\mathtt{X}) <: \exists\Delta'.\mathtt{X} \text{ (S-LVAR)}$$

$$\frac{\texttt{class C<}\overline{\mathtt{X}}\ \overline{\mathtt{B}}\texttt{>} \lhd \mathtt{N}\ \{\ldots\}}{\Delta \ \vdash \ \exists\Delta'.\mathtt{C}\texttt{<}\overline{\mathtt{T}}\texttt{>} <: \exists\Delta'.[\overline{\mathtt{T}}/\overline{\mathtt{X}}]\mathtt{N}} \text{ (WS-SUBCLASS)}$$

$$\frac{\Delta\Delta' \vdash \overline{\mathtt{U}} \in [\overline{\mathtt{U}}/\overline{\mathtt{X}}]\overline{\mathtt{B}}}{\Delta \vdash \exists\Delta'.[\overline{\mathtt{U}}/\overline{\mathtt{X}}]\mathtt{K} <: \exists\overline{\mathtt{X}} \in \overline{\mathtt{B}}.\mathtt{K}} \text{ (WS-ENV)}$$

$$\frac{\Delta \vdash \mathit{snap}(\mathtt{S}) <: \mathit{snap}(\mathtt{T})}{\Delta \vdash \mathtt{S} <: \mathtt{T}} \text{ (WS-SYNTACTIC)}$$

Bound Containment:

$$\Delta \vdash \mathtt{T} \in \bullet\ \bullet \text{ (WB-NONE)}$$

$$\frac{\Delta \vdash \mathtt{S} <: \mathtt{T}}{\Delta \vdash \mathtt{T} \in \bullet\ \rhd \mathtt{S}} \text{ (WB-LOWER)}$$

$$\frac{\Delta \vdash \mathtt{T} <: \mathtt{S} \qquad \Delta \vdash \mathtt{T} \in \bullet\ \mathtt{B}_\rhd}{\Delta \vdash \mathtt{T} \in \lhd \mathtt{S}\ \mathtt{B}_\rhd} \text{ (WB-UPPER)}$$

Figure 4-8: LJ's subtyping rules. The rules are the same as WFJ's.

The judgment `C OK` declares the class declaration of `C` to be sound. These judgments are unchanged from WFJ and can be found in [47].

4.1.4 Operational semantics

To support the assignment of fields, we introduce a store, $\mathcal{S}$, to WFJ's reduction rules. As in ClassicJava [18], the store is a mapping from an object to a pair containing the class type, `N`, of the object and a field record. A field record, $\mathcal{F}$, is a mapping from field names to values.

The reduction rules are shown in figure 4-10. Each reduction rule is a relationship, $\langle \mathtt{e},\ \mathcal{S}\rangle \longrightarrow \langle \mathtt{e}',\ \mathcal{S}'\rangle$, where `e` with store $\mathcal{S}$ reduces to `e'` with store $\mathcal{S}'$ in one step.

The addition of the assignment statement requires new reduction and congruence rules to be added to those of WFJ. The reduction rule for `set` binds the field to a new value, then evaluates the "`then`" part of the expression.

Expression typing:

$$\Delta; \Gamma \vdash \mathtt{x} : \mathit{snap}(\Gamma(\mathtt{x})) \qquad \text{(T-Var)}$$

$$\frac{\begin{array}{c}\Delta; \Gamma \vdash \mathtt{e}_0 : \mathtt{E}_0 \\ \mathit{bound}_\Delta(\mathtt{E}_0) = \exists \Delta_0.\mathtt{N}_0 \qquad \mathit{fields}(\mathtt{N}_0) = \boxed{\overline{\mathtt{AF}}}\ \overline{\mathtt{T}}\ \overline{\mathtt{f}} \qquad \mathit{snap}(\mathtt{T}_i) = \exists \Delta'.\mathtt{K}\end{array}}{\Delta; \Gamma \vdash \mathtt{e}_0.\mathtt{f}_i : \exists \Delta_0 \Delta'.\mathtt{K}} \text{ (T-Get)}$$

$$\frac{\begin{array}{c}\Delta \vdash \overline{\mathtt{P}}\ \mathtt{ok} \qquad \Delta; \Gamma \vdash \mathtt{e}_0 : \mathtt{E}_0 \qquad \mathit{bound}_\Delta(\mathtt{E}_0) = \exists \Delta_0.\mathtt{N}_0 \qquad \Delta; \Gamma \vdash \overline{\mathtt{e}} : \exists \overline{\Delta}.\overline{\mathtt{K}} \\ \Delta_1 = \Delta \Delta_0 \overline{\Delta} \qquad \mathit{mtype}(\mathtt{m}, \mathtt{N}_0) = \mathtt{<}\overline{\mathtt{Y}}\ \overline{\mathtt{B}}\mathtt{>}\overline{\mathtt{U}} \to \mathtt{U} \qquad \overline{\mathtt{V}} = \mathit{capture}_{\Delta_1}(\overline{\mathtt{P}}, \overline{\mathtt{Y}}, \overline{\mathtt{U}}, \overline{\mathtt{K}}) \\ \Delta_1 \vdash \overline{\mathtt{V}} \in [\overline{\mathtt{V}}/\overline{\mathtt{Y}}]\mathtt{B} \qquad \Delta_1 \vdash \overline{\mathtt{K}} <: [\overline{\mathtt{V}}/\overline{\mathtt{Y}}]\overline{\mathtt{U}} \qquad \mathit{snap}([\overline{\mathtt{V}}/\overline{\mathtt{Y}}]\mathtt{U}) = \exists \Delta'.\mathtt{K}\end{array}}{\Delta; \Gamma \vdash \mathtt{e}_0.\mathtt{<}\overline{\mathtt{P}}\mathtt{>m}(\overline{\mathtt{e}}) : \exists \Delta_0 \overline{\Delta} \Delta'.\mathtt{K}} \text{ (T-Invk)}$$

$$\frac{\Delta \vdash \mathtt{N}\ \mathtt{ok} \qquad \mathit{fields}(\mathtt{N}) = \boxed{\overline{\mathtt{AF}}}\ \overline{\mathtt{T}}\ \overline{\mathtt{f}} \qquad \Delta; \Gamma \vdash \overline{\mathtt{e}} : \overline{\mathtt{E}} \qquad \Delta \vdash \overline{\mathtt{E}} <: \mathit{snap}(\overline{\mathtt{T}})}{\Delta; \Gamma \vdash \mathtt{new}\ \mathtt{N}(\overline{\mathtt{e}}) : \exists \emptyset.\mathtt{N}} \text{ (T-New)}$$

$$\boxed{\frac{\begin{array}{c}\Delta; \Gamma \vdash \mathtt{e}_0 : \mathtt{E}_0 \\ \mathit{bound}_\Delta(\mathtt{E}_0) = \exists \Delta_0.\mathtt{N}_0 \qquad \mathit{fields}(\mathtt{N}_0) = \boxed{\overline{\mathtt{AF}}}\ \overline{\mathtt{T}}\ \overline{\mathtt{f}} \qquad \boxed{\mathtt{AF}_i = \mathtt{assignable}} \\ \mathit{snap}(\mathtt{T}_i) = \exists \Delta_i.\mathtt{K} \qquad \Delta; \Gamma \vdash \mathtt{e}_v : \mathtt{E} \qquad \Delta \Delta_0 \vdash \mathtt{E} <: \exists \Delta_i.\mathtt{K} \qquad \mathtt{e}_b : \mathtt{E}'\end{array}}{\Delta; \Gamma \vdash \mathtt{set}\ \mathtt{e}_0.\mathtt{f}_i = \mathtt{e}_v\ \mathtt{then}\ \mathtt{e}_b : \mathtt{E}'} \text{ (T-Set)}}$$

Method typing:

$$\frac{\begin{array}{c}\Delta \vdash \overline{\mathtt{B}'},\ \mathtt{T},\ \overline{\mathtt{T}}\ \mathtt{ok} \qquad \Delta = \overline{\mathtt{Y}} \in \overline{\mathtt{B}'},\ \overline{\mathtt{X}} \in \overline{\mathtt{B}} \qquad \Delta; \overline{\mathtt{x}} : \overline{\mathtt{T}}, \mathtt{this} : \mathtt{C}\mathtt{<}\overline{\mathtt{X}}\mathtt{>} \vdash \mathtt{e}_0 : \mathtt{E} \\ \Delta \vdash \mathtt{E} <: \mathit{snap}(\mathtt{T}) \qquad \mathtt{class}\ \mathtt{C}\mathtt{<}\overline{\mathtt{X}}\ \overline{\mathtt{B}}\mathtt{>} \lhd \mathtt{N}\ \{\ldots\} \qquad \mathit{override}(\mathtt{m}, \mathtt{N}, \mathtt{<}\overline{\mathtt{Y}}\ \overline{\mathtt{B}'}\mathtt{>}\overline{\mathtt{T}} \to \mathtt{T})\end{array}}{\mathtt{<}\overline{\mathtt{Y}}\ \overline{\mathtt{B}'}\mathtt{>}\ \mathtt{T}\ \mathtt{m}(\overline{\mathtt{T}}\ \overline{\mathtt{x}})\{\ \mathtt{return}\ \mathtt{e}_0;\ \}\ \mathtt{OK\ IN}\ \mathtt{C}\mathtt{<}\overline{\mathtt{X}}\ \overline{\mathtt{B}}\mathtt{>}} \text{ (T-Method)}$$

Class typing:

$$\frac{\overline{\mathtt{X}} \in \overline{\mathtt{B}} \vdash \overline{\mathtt{B}},\ \mathtt{N},\ \overline{\mathtt{T}}\ \mathtt{ok} \qquad \overline{\mathtt{M}}\ \mathtt{OK\ IN}\ \mathtt{C}\mathtt{<}\overline{\mathtt{X}}\ \overline{\mathtt{B}}\mathtt{>}}{\mathtt{class}\ \mathtt{C}\mathtt{<}\overline{\mathtt{X}}\ \overline{\mathtt{B}}\mathtt{>} \lhd \mathtt{N}\ \{\boxed{\overline{\mathtt{AF}}}\ \overline{\mathtt{T}}\ \overline{\mathtt{f}};\ \overline{\mathtt{M}}\}\ \mathtt{OK}} \text{ (T-Class)}$$

Figure 4-9: Lightweight Java (LJ) typing rules. Changes from WFJ are indicated by boxes.

4.1.5 Properties

First we state by a type soundness theorem for LJ. If a term is well typed and reduces to a normal form (an expression that cannot reduce any further and, therefore, is an object, v) then it is a value of a subtype of the original term's type. Put differently, an evaluation cannot go wrong, which in our model means getting stuck.

Theorem 1 (LJ Type Soundness). *If* $\emptyset; \emptyset \vdash \mathtt{e} : \mathtt{E}$ *and* $\langle \mathtt{e},\ \mathcal{S} \rangle \to^* \langle \mathtt{e}',\ \mathcal{S}' \rangle$ *with* $\mathtt{e}'$ *a normal form, then* $\mathtt{e}'$ *is a value,* v*, such that* $\mathcal{S}'(v) = \langle \mathtt{N},\ \mathcal{F} \rangle$ *and* $\emptyset \vdash \exists \emptyset.\mathtt{N} <: \mathtt{E}$*.*

The soundness of LJ can be proved using the standard technique of subject reduction and progress theorems. A sketch of the proof is given below, a full proof is not currently available for WFJ or LJ.

v ::= an object
Computation:

$$\frac{\mathcal{S}(v_1) = \langle \mathtt{N},\ \mathcal{F}\rangle \qquad \mathcal{F}(\mathtt{f}_i) = v_2}{\langle v_1.\mathtt{f}_i,\ \mathcal{S}\rangle \longrightarrow \langle v_2,\ \mathcal{S}\rangle}\ \text{(R-Field)}$$

$$\frac{\begin{array}{c}\mathcal{S}(v) = \langle \mathtt{N},\ \mathcal{F}\rangle \qquad \mathcal{S}(\overline{v}) = \langle \overline{\mathtt{N}},\ \overline{\mathcal{F}}\rangle \qquad \mathit{mtype}(\mathtt{m}, \mathtt{N}) = \mathtt{<}\overline{\mathtt{Y}}\ \overline{\mathtt{B}}\mathtt{>}\overline{\mathtt{U}} \to \mathtt{U} \\ \overline{\mathtt{V}} = \mathit{capture}_{\emptyset}(\overline{\mathtt{P}}, \overline{\mathtt{Y}}, \overline{\mathtt{U}}, \overline{\mathtt{N}}) \qquad \mathit{mbody}(\mathtt{<}\overline{\mathtt{V}}\mathtt{>m},\ \mathtt{N}) = \overline{\mathtt{x}}.\mathtt{e}_0\end{array}}{\langle v.\mathtt{<}\overline{\mathtt{P}}\mathtt{>m}(\overline{v}),\ \mathcal{S}\rangle \longrightarrow \langle [\overline{v}/\overline{\mathtt{x}},\ v/\mathtt{this}]\mathtt{e}_0,\ \mathcal{S}\rangle}\ \text{(R-Invk)}$$

$$\frac{v \notin \mathtt{dom}(\mathcal{S}) \qquad \mathcal{F} = [\overline{\mathtt{f}} \mapsto \overline{v}]}{\langle \mathtt{new\ N}(\overline{v}),\ \mathcal{S}\rangle \longrightarrow \langle v,\ \mathcal{S}[v \mapsto \langle \mathtt{N},\ \mathcal{F}\rangle]\rangle}\ \text{(R-New)}$$

$$\frac{\mathcal{S}(v_1) = \langle \mathtt{N},\ \mathcal{F}\rangle}{\langle \mathtt{set}\ v_1.\mathtt{f}_i = v_2\ \mathtt{then}\ \mathtt{e}_b,\ \mathcal{S}\rangle \longrightarrow \langle \mathtt{e}_b,\ \mathcal{S}[v_1 \mapsto \langle \mathtt{N},\ \mathcal{F}[\mathtt{f}_i \mapsto v_2]\rangle]\rangle}\ \text{(R-Set)}$$

Congruence:

$$\frac{\langle \mathtt{e}_0,\ \mathcal{S}\rangle \longrightarrow \langle \mathtt{e}_0',\ \mathcal{S}\rangle}{\langle \mathtt{e}_0.\mathtt{f},\ \mathcal{S}\rangle \longrightarrow \langle \mathtt{e}_0'.\mathtt{f},\ \mathcal{S}\rangle}\ \text{(RC-Field)}$$

$$\frac{\langle \mathtt{e}_0,\ \mathcal{S}\rangle \longrightarrow \langle \mathtt{e}_0',\ \mathcal{S}\rangle}{\langle \mathtt{e}_0.\mathtt{<}\overline{\mathtt{P}}\mathtt{>m}(\overline{\mathtt{e}}),\ \mathcal{S}\rangle \longrightarrow \langle \mathtt{e}_0'.\mathtt{<}\overline{\mathtt{P}}\mathtt{>m}(\overline{\mathtt{e}}),\ \mathcal{S}\rangle}\ \text{(RC-Invk-Recv)}$$

$$\frac{\langle \mathtt{e}_i,\ \mathcal{S}\rangle \longrightarrow \langle \mathtt{e}_i',\ \mathcal{S}\rangle}{\langle v.\mathtt{<}\overline{\mathtt{P}}\mathtt{>m}(\overline{v}, \mathtt{e}_i, \overline{\mathtt{e}}),\ \mathcal{S}\rangle \longrightarrow \quad \langle v.\mathtt{<}\overline{\mathtt{P}}\mathtt{>m}(\overline{v}, \mathtt{e}_i', \overline{\mathtt{e}}),\ \mathcal{S}\rangle}\ \text{(RC-Invk-Arg)}$$

$$\frac{\langle \mathtt{e}_i,\ \mathcal{S}\rangle \longrightarrow \langle \mathtt{e}_i',\ \mathcal{S}\rangle}{\langle \mathtt{new\ N}(\overline{v}, \mathtt{e}_i, \overline{\mathtt{e}}),\ \mathcal{S}\rangle \longrightarrow \quad \langle \mathtt{new\ N}(\overline{v}, \mathtt{e}_i', \overline{\mathtt{e}}),\ \mathcal{S}\rangle}\ \text{(RC-New-Arg)}$$

$$\frac{\langle \mathtt{e}_0,\ \mathcal{S}\rangle \longrightarrow \langle \mathtt{e}_0',\ \mathcal{S}\rangle}{\langle \mathtt{set}\ \mathtt{e}_0.f = \mathtt{e}_v\ \mathtt{then}\ \mathtt{e}_b,\ \mathcal{S}\rangle \longrightarrow \quad \langle \mathtt{set}\ \mathtt{e}_0'.f = \mathtt{e}_v\ \mathtt{then}\ \mathtt{e}_b,\ \mathcal{S}\rangle}\ \text{(RC-Set-LHS)}$$

$$\frac{\langle \mathtt{e}_v,\ \mathcal{S}\rangle \longrightarrow \langle \mathtt{e}_v',\ \mathcal{S}\rangle}{\langle \mathtt{set}\ v.f = \mathtt{e}_v\ \mathtt{then}\ \mathtt{e}_b,\ \mathcal{S}\rangle \longrightarrow \quad \langle \mathtt{set}\ v.f = \mathtt{e}_v'\ \mathtt{then}\ \mathtt{e}_b,\ \mathcal{S}\rangle}\ \text{(RC-Set-RHS)}$$

Figure 4-10: Lightweight Java (LJ) reduction rules. Unlike WFJ, LJ's reduction rules contain a store.

The reduction theorem states that each step taken in the evaluation preserves the type correctness of the expression-store pair. During each step of the reduction, the environment and type environment must be consistent, $\vdash_\sigma$, with the store:

$$
\begin{array}{ll}
& \Delta;\Gamma \vdash_\sigma \mathcal{S} \iff \\
& \quad \mathcal{S}(v) = \langle \mathtt{N},\ \mathcal{F} \rangle \Rightarrow \\
\Sigma_1: & \qquad \Delta \vdash \mathtt{N} <: \Gamma(v) \\
\Sigma_2: & \qquad \text{and } \mathrm{dom}(\mathcal{F}) = \{\mathtt{f} \mid \mathtt{AF\ T\ f}\ \in \mathit{fields}(\mathtt{N})\} \\
\Sigma_3: & \qquad \text{and } \mathrm{rng}(\mathcal{F}) \subseteq \mathrm{dom}(\mathcal{S}) \\
\Sigma_4: & \qquad \text{and } (\mathcal{F}(\mathtt{f}) = v' \text{and } \mathtt{AF\ T\ f} \in \mathit{fields}(\mathtt{N}) \\
& \qquad\qquad \Rightarrow ((\mathcal{S}(v') = \langle \mathtt{N}',\ \mathcal{F}' \rangle) \Rightarrow \Delta \vdash \mathtt{N}'\ <: \mathtt{T})) \\
\Sigma_5: & \text{and } v \in \mathrm{dom}(\Gamma) \Rightarrow v \in \mathrm{dom}(\mathcal{S}) \\
\Sigma_6: & \text{and } \mathrm{dom}(\mathcal{S}) \subseteq \mathrm{dom}(\Gamma)
\end{array}
$$

Theorem 2 (LJ Subject Reduction). *If* $\Delta;\Gamma \vdash \mathtt{e} : \mathtt{E}$, $\Delta;\Gamma \vdash_\sigma \mathcal{S}$, *and* $\langle \mathtt{e},\ \mathcal{S} \rangle \longrightarrow \langle \mathtt{e}',\ \mathcal{S}' \rangle$, *then there exists a* Γ' *and* Δ' *such that* $\Delta';\Gamma' \vdash \mathtt{e}' : \mathtt{E}$ *and* $\Delta';\Gamma' \vdash_\sigma \mathcal{S}'$.

Proof. The proof can be done by induction on the derivation of $\langle \mathtt{e},\ \mathcal{S} \rangle \longrightarrow \langle \mathtt{e}',\ \mathcal{S}' \rangle$ with a case analysis on the reduction rule used. For each case, we construct the new environment Γ' and a new type environment Δ' and show that (1) $\Delta';\Gamma' \vdash \mathtt{e}' : \mathtt{E}$ and (2) $\Delta';\Gamma' \vdash_\sigma \mathcal{S}$. □

The progress theorem states that a well-typed program can not get stuck. That is, if a expression is well-typed and not a value, there exists some progress rule that may be applied to the expression.

Theorem 3 (LJ Progress). *If* $\Delta;\Gamma \vdash \mathtt{e} : \mathtt{E}$ *and* $\Delta;\Gamma \vdash_\sigma \mathcal{S}$, *then either* `e` *is a value or there exists an* $\langle \mathtt{e}',\ \mathcal{S}' \rangle$ *such that* $\langle \mathtt{e},\ \mathcal{S} \rangle \longrightarrow \langle \mathtt{e}',\ \mathcal{S}' \rangle$.

Proof. The proof can be done by an analysis of the possible cases for the current redex in `e` (in the case that `e` is not a value). □

The LJ Type Soundness theorem is immediate from the LJ Subject Reduction and LJ Progress theorems.

In addition to type soundness we would like a guarantee that LJ obeys its assignability rules. We state this property as the LJ Assignment Soundness theorem.

Theorem 4 (LJ Assignment Soundness). *Outside of an object's constructor, assignments may only be made to* `assignable` *fields.*

Proof. Immediate from T-Set. □

4.2 Lightweight Javari

We add Javari's concept of reference immutability to LJ to create the language Lightweight Javari (LJR). In LJR, every type is modified by one of the mutability modifiers: `readonly`, `mutable`, or (for field types only) `this-mutable`. In addition to `final` and `assignable`, LJR also allows fields be marked as this-assignable with the `this-assignable` keyword.

$\texttt{T}, \texttt{S}, \texttt{U}, \texttt{V}$	::=	$\boxed{\texttt{ML}}\ \texttt{C<A>} \mid \texttt{X}$	*types*
$\texttt{A}$	::=	$\texttt{T} \mid \texttt{?}\ \texttt{B}$	*type arguments*
$\texttt{P}$	::=	$\texttt{T} \mid \star$	*method type parameters*
$\texttt{N}$	::=	$\texttt{C<}\overline{\texttt{T}}\texttt{>}$	*class types*
$\texttt{K}$	::=	$\boxed{\texttt{ML}}\ \texttt{N} \mid \texttt{X}$	*class types and var.s*
$\texttt{B}$	::=	$\texttt{B}_\triangleleft\ \texttt{B}_\triangleright$	*bounds*
$\texttt{B}_\triangleleft$	::=	$\triangleleft\ \texttt{T} \mid \bullet$	*upper bounds*
$\texttt{B}_\triangleright$	::=	$\triangleright\ \texttt{T} \mid \bullet$	*lower bounds*
$\boxed{\texttt{F}}$	::=	$\texttt{MF C<}\overline{\texttt{FA}}\texttt{>} \mid \texttt{X}$	*fieldtypes*
$\boxed{\texttt{FA}}$	::=	$\texttt{F} \mid \texttt{?}\ \texttt{B}$	*field type arguments*
$\texttt{Q}$	::=	$\texttt{class C<}\overline{\texttt{X}}\ \overline{\texttt{B}}\texttt{>} \triangleleft \texttt{C}'\texttt{<}\overline{\texttt{T}}\texttt{>}\{\overline{\texttt{AF}}\ \boxed{\overline{\texttt{F}}}\ \overline{\texttt{f}}\texttt{;}\ \overline{\texttt{M}}\}$	*class declaration*
$\texttt{M}$	::=	$\texttt{<}\overline{\texttt{X}}\ \overline{\texttt{B}}\texttt{>}\ \texttt{T m}(\overline{\texttt{T}}\ \overline{\texttt{x}})\boxed{\texttt{ML}}\{\ \texttt{return e;}\ \}$	*method declarations*
$\texttt{e}$	::=	$\texttt{x}$	*expressions*
	$\mid$	$\texttt{e.f}$	
	$\mid$	$\texttt{e.m<}\overline{\texttt{T}}\texttt{>}(\overline{\texttt{e}})$	
	$\mid$	$\texttt{new N}(\overline{\texttt{e}})$	
	$\mid$	$\texttt{set e.f} = \texttt{e then e}$	
$\texttt{AF}$	::=	$\texttt{final} \mid \texttt{assignable} \mid \texttt{this-assignable}$	*assignabilities*
$\boxed{\texttt{ML}}$	::=	$\texttt{readonly} \mid \texttt{mutable}$	*local mutabilities*
$\boxed{\texttt{MF}}$	::=	$\texttt{this-mutable} \mid \texttt{ML}$	*field mutabilities*
$\texttt{E}$	::=	$\exists\Delta.\texttt{K}$	*existential types*
Δ	::=	$\emptyset \mid \Delta, \texttt{X} \in \texttt{B}$	*type environments*
$\texttt{f}$		*field names*	
$\texttt{C}, \texttt{D}$		*class names*	
$\texttt{X}, \texttt{Y}, \texttt{Z}$		*type variables*	
$\texttt{x}$		*variables*	
$\texttt{m}$		*method names*	

Figure 4-11: Syntax of Lightweight Javari (LJR). Changes from Lightweight Java (LJ, figure 4-3) are indicated by boxes.

4.2.1 Syntax

The syntax of LJR is shown in figure 4-11. `ML` and `MF` are introduced to range over the mutability modifiers that may occur in a program. `ML` ranges over the mutabilities that may appear in local variables' types: `readonly` and `mutable`. `MF`, mutability for fields, additionally ranges over `this-mutable`. `C` continues to range over class names. `T` now ranges over both read-only and mutable types. Thus, a type is a pair, `ML C<`$\overline{\texttt{A}}$`>`, consisting of a mutability and a "base" type.

`F` ranges over field type declarations, including those with this-mutability. A type argument to a field may have this-mutability. We denote such type arguments as `FA`. In addition to field types, `FA` ranges over wildcards. The definition of `FA` does not allow the bound of a wildcard type to have this-mutability as such an extension is cumbersome to the syntax. All type meta-variables other than `F` may not have this-mutability.

A mutability modifier, `ML` or `MF`, may not be applied to a type that is a type variable

Assignability resolving:

$$
\begin{array}{lcl}
\textit{assignability}(\texttt{this-assignable},\ \texttt{mutable}) & = & \texttt{assignable} \\
\textit{assignability}(\texttt{this-assignable},\ \texttt{readonly}) & = & \texttt{final} \\
\textit{assignability}(\texttt{final},\ \texttt{ML}) & = & \texttt{final} \\
\textit{assignability}(\texttt{assignable},\ \texttt{ML}) & = & \texttt{assignable}
\end{array}
$$

Mutability resolving:

$$\textit{mutability}(\texttt{F},\ \texttt{mutable}) = [\texttt{mutable}/\texttt{this-mutable}]\texttt{F}$$

$$\textit{mutability}(\texttt{F},\ \texttt{readonly}) = [?\ \texttt{readonly}/\texttt{this-mutable}]\texttt{F}$$

Figure 4-12: Lightweight Javari (LJR): Resolving assignability and mutability. Also see figure 3-7, which presents the same information in a different form. For *assignability*, the first parameter is the keyword with which the field is declared and the second parameter is the mutability of the reference through which the field is accessed. The first parameter to *mutability* is the type of a field. The second parameter is the mutability of the reference through which the field is accessed.

(see section 3.6).

`AL` still ranges over assignable and final, modifiers applicable to method parameters (local variables) and `AF` ranges over the assignabilities that a field may be declared to have: `assignable`, `final`, and `this-assignable`.

4.2.2 Auxiliary functions

Resolving this-assignability and this-mutability

The assignability/mutability of a this-assignable/this-mutable field is determined by the mutability of the reference used to reach the field (section 3.5).

The function *assignability*(`AF`, `ML`) (shown in figure 4-12) is used to resolve the assignability of a field. A this-assignable field resolves to assignable, if reached through a mutable reference, and to final, if reached through a read-only reference. Fields that are declared `final` or `assignable` are trivially resolved to final and assignable, respectively.

The function *mutability* (shown in figure 4-12) is used to resolve the mutability of a this-mutable type. If reached through a mutable reference, *mutability* replaces all instances of `this-mutable` in the type with `mutable`. If reached through a readonly reference, *mutability* replaces all instances of `this-mutable` in the type with `readonly`. Unlike *assignability*, *mutability* must resolve the mutabilities of the type arguments in addition to the top-level type.

Existential type creation: Snap

We extend LJ's *snap* to handle `? readonly` types. `? readonly` types denote the resolved (see previous section) type of this-mutable type (top-level or type arguments) reached through a read-only reference. To ensure transitive immutability, a `? readonly` type can only be read (as determined by the type's upper bound) as read-only. However, a this-mutable type may be read as a mutable when reached through a mutable reference; therefore, only mutable

Snap:

$$snap(\texttt{X}) = \exists\emptyset.\texttt{X}$$

$$\frac{\texttt{class C<}\overline{\texttt{Y}}\ \overline{\texttt{B}_0}\texttt{>} \lhd \texttt{N}\ \{\ldots\} \qquad fix(\overline{\texttt{A}}, \overline{\texttt{B}_0}) = (\overline{\texttt{T}}, \overline{\texttt{X}}, \overline{\texttt{B}}) \qquad \Delta = \overline{\texttt{X}} \in [\overline{\texttt{T}}/\overline{\texttt{Y}}]\overline{\texttt{B}}}{snap(\texttt{C<}\overline{\texttt{A}}\texttt{>}) = \exists\Delta.\texttt{C<}\overline{\texttt{T}}\texttt{>}}$$

$$\boxed{\begin{array}{c}\texttt{class C<}\overline{\texttt{Y}}\ \overline{\texttt{B}_0}\texttt{>} \lhd \texttt{N}\ \{\ldots\} \\ fix(\overline{\texttt{A}}, \overline{\texttt{B}_0}) = (\overline{\texttt{T}}, \overline{\texttt{X}}, \overline{\texttt{B}}) \qquad \texttt{B}_\lhd = \lhd\ \texttt{readonly C<}\overline{\texttt{T}}\texttt{>} \\ fix([\texttt{mutable}/?\ \texttt{readonly}]\overline{\texttt{A}}, \overline{\texttt{X}}, \overline{\texttt{B}}) = (\overline{\texttt{T}'}, \overline{\texttt{X}'}, \overline{\texttt{B}'}) \qquad \texttt{B}_\rhd = \rhd\ \texttt{mutable C<}\overline{\texttt{T}'}\texttt{>} \\ \texttt{Z fresh} \qquad \Delta = \texttt{Z} :: \overline{\texttt{X}}\overline{\texttt{X}'} \in [\overline{\texttt{T}}/\overline{\texttt{Y}}]\texttt{B}_\lhd\ \texttt{B}_\rhd :: \overline{\texttt{B}}\overline{\texttt{B}'} \\ \hline snap(?\ \texttt{readonly C<}\overline{\texttt{A}}\texttt{>}) = \exists\Delta.\texttt{Z}\end{array}}$$

Fix:

$$\frac{fix(\overline{\texttt{A}}, \overline{\texttt{B}_0}) = (\overline{\texttt{T}}, \overline{\texttt{X}}, \overline{\texttt{B}})}{fix(\texttt{T} :: \overline{\texttt{A}}, \texttt{B}_0 :: \overline{\texttt{B}_0}) = (\texttt{T} :: \overline{\texttt{T}}, \overline{\texttt{X}}, \overline{\texttt{B}})}$$

$$\frac{fix(\overline{\texttt{A}}, \overline{\texttt{B}_0}) = (\overline{\texttt{T}}, \overline{\texttt{X}}, \overline{\texttt{B}}) \qquad \texttt{X fresh}}{fix(?\ \texttt{B} :: \overline{\texttt{A}}, \texttt{B}_0 :: \overline{\texttt{B}_0}) = (\texttt{X} :: \overline{\texttt{T}}, \texttt{X} :: \overline{\texttt{X}}, merge(\texttt{B}, \texttt{B}_0) :: \overline{\texttt{B}})}$$

$$\boxed{\frac{fix(\overline{\texttt{A}}, \overline{\texttt{B}_0}) = (\overline{\texttt{T}}, \overline{\texttt{X}}, \overline{\texttt{B}}) \qquad \texttt{B}'_\lhd = \lhd\ \texttt{readonly C<}\overline{\texttt{A}'}\texttt{>} \qquad \texttt{B}'_\rhd = \rhd\ \texttt{mutable}\ [\texttt{mutable}/?\ \texttt{readonly}]\texttt{C<}\overline{\texttt{A}'}\texttt{>}}{fix(?\ \texttt{readonly C<}\overline{\texttt{A}'}\texttt{>} :: \overline{\texttt{A}}, \texttt{B}_0 :: \overline{\texttt{B}_0}) = (\texttt{X} :: \overline{\texttt{T}}, \texttt{X}, :: \overline{\texttt{X}}, \texttt{B}'_\lhd\ \texttt{B}'_\rhd :: \overline{\texttt{B}})}\ \texttt{X fresh}}$$

$$fix(\bullet, \bullet) = (\bullet, \bullet, \bullet)$$

Figure 4-13: LJR's methods for creating existential types. Changes from LJ's methods (figure 4-5) are indicated by boxes. The *merge* function is unchanged and not shown.

types may be assigned (as determined by the type's lower bound) to a `? readonly` type.[2] This intuition can be seen in the new *snap* rule (shown, boxed, in figure 4-13). In the rule, a fresh type variable is created with a readonly upper bound and a mutable lower bound. In *snap*, the upper bound's type arguments are set to the non-wildcard types calculated by *fix*. In the case of the lower bound, all instances of `? readonly` are replaced with `mutable` before *snap*() is applied to the type arguments to ensure that only fully mutable types are contained in the lower bound. The new *fix* rule (also shown, boxed, in figure 4-13) is similar but simpler because it does not need to recurse into the type arguments—*snap*() is only required remove wildcards from the top-level type.

Method type lookup:

$$\frac{\texttt{class C<}\overline{\texttt{X}}\ \overline{\texttt{N}}\texttt{>} \lhd \texttt{B}\{\overline{\texttt{AF}}\ \boxed{\overline{\texttt{F}}}\ \overline{\texttt{f}};\ \overline{\texttt{M}}\} \qquad \texttt{<}\overline{\texttt{Y}}\ \overline{\texttt{B}'}\texttt{>}\ \texttt{U m(}\overline{\texttt{U}}\ \overline{\texttt{x}}\texttt{)}\ \boxed{\texttt{ML}}\ \{\ \texttt{return e;}\ \} \in \overline{\texttt{M}}}{\mathit{mtype}(\texttt{m}, \texttt{C<}\overline{\texttt{T}}\texttt{>}) = [\overline{\texttt{T}}/\overline{\texttt{X}}](\texttt{<}\overline{\texttt{Y}}\ \overline{\texttt{B}'}\texttt{>}\overline{\texttt{U}}\ \boxed{\texttt{ML}} \to \texttt{U})}$$

$$\frac{\texttt{class C<}\overline{\texttt{X}}\ \overline{\texttt{N}}\texttt{>} \lhd \texttt{B}\{\overline{\texttt{AF}}\ \boxed{\overline{\texttt{F}}}\ \overline{\texttt{f}};\ \overline{\texttt{M}}\} \qquad \texttt{m} \notin \overline{\texttt{M}}}{\mathit{mtype}(\texttt{m}, \texttt{C<}\overline{\texttt{T}}\texttt{>}) = \mathit{mtype}(\texttt{m}, [\overline{\texttt{T}}/\overline{\texttt{X}}]\texttt{N})}$$

Valid method overriding:

$$\frac{\begin{array}{c}\mathit{mtype}(\texttt{m}, \texttt{N}) = \texttt{<}\overline{\texttt{Y}'}\ \overline{\texttt{B}'}\texttt{>}\overline{\texttt{T}'}\ \boxed{\texttt{ML}'} \to \texttt{T}' \text{ implies} \\ \overline{\texttt{B}} = [\overline{\texttt{Y}}/\overline{\texttt{Y}'}]\overline{\texttt{B}'} \text{ and } \overline{\texttt{T}} = [\overline{\texttt{Y}}/\overline{\texttt{Y}'}]\overline{\texttt{T}'} \text{ and } \overline{\texttt{Y}} \in \overline{\texttt{B}} \vdash \texttt{T} <: [\overline{\texttt{Y}}/\overline{\texttt{Y}'}]\texttt{T}' \text{ and } \boxed{\texttt{ML}' = \texttt{ML}}\end{array}}{\mathit{override}(\texttt{m},\ \texttt{N},\ \texttt{<}\overline{\texttt{Y}} \lhd \overline{\texttt{B}}\texttt{>}\overline{\texttt{T}}\ \boxed{\texttt{ML}} \to \texttt{T})}$$

Figure 4-14: Lightweight Javari (LJR) auxiliary functions. Changes from Lightweight Java (LJ) (figure 4-7) are indicated by boxes.

$$\frac{\texttt{class C<}\overline{\texttt{X}}\ \overline{\texttt{B}}\texttt{>} \lhd \texttt{N}\ \{\ldots\}}{\Delta\ \vdash\ \exists\Delta'.\texttt{ML C<}\overline{\texttt{T}}\texttt{>} <: \exists\Delta'.\texttt{ML}\ [\overline{\texttt{T}}/\overline{\texttt{X}}]\texttt{N}}\ \text{(WS-Subclass)}$$

$$\Delta \vdash \texttt{mutable C<}\overline{\texttt{T}}\texttt{>} <: \texttt{readonly C<}\overline{\texttt{T}}\texttt{>}\ \text{(WS-Mutability)}$$

Figure 4-15: Lightweight Javari (LJR) subtyping rules. Only changes from LJ's subtyping rules (figure 4-8) are shown.

Class member lookup

The other auxiliary functions that are changed from LJ's are shown in figure 4-14. The functions for method type lookup and valid method overriding must be modified because the mutability of the receiver is part of a method's signature. In the case of method lookup, the function now returns the mutability of the receiver as a part of the method's type. Because LJ and LJR do not allow method overloading, there cannot be a read-only and a mutable version of a method; thus, the mutability of the receiver does not need to be passed to *mtype*. The rule for valid method overriding now checks that the mutability of the receiver in the overriding method matches the mutability of the receiver in the overridden method.

4.2.3 Static semantics

Subtyping

The subtyping rules of LJ (figure 4-8) must be modified to include mutability. The `this-mutable` construct is not a part of the type hierarchy and, thus, is not shown in the subtyping rules. The mutability of a field declared `this-mutable` must be resolved before subtyping relations may be calculated.

[2]Otherwise, a type loop would exist that could be used to convert read-only types to mutable types; see section 3.

As with LJ the subtyping rules are applied to existential types.

LJR's subtyping rules (figure 4-15) are little changed from those of LJ. WS-Subclass is modified to take the mutabilities of the subtype and supertype into account. WS-Mutability is added to enforce the fact that a mutable version of a type is a subtype of the readonly version of the type. The other LJ subtyping rules are also subtyping rules of LJR with the exception that they now operate over LJR types instead of LJ types.

Typing judgements

The type rules of LJR are shown in figure 4-16. The rules of LJ are augmented to check that the mutability of an expression is correct for its context.

The RT-Get rule is changed to calculate the mutability of the field using the mutability of the reference, $\texttt{e}_0$, used to reach the field. RT-Invk is augmented to check that the mutability of the receiver is legal for the declared mutability of the method. The rule for `new` expressions require that the object created is mutable because the object may be assigned to a mutable reference; therefore, when calculating the mutability of this-mutable fields, the mutability of the reference through which the fields are accessed is given as `mutable`. The changes to RT-Set are the same as to RT-Get. The RT-Method rule assigns the receiver's mutability to be equal to the mutability with which the method is declared.

4.2.4 Operational semantics

Javari is only a change to Java's type system[3]; therefore, Javari should not affect the runtime behavior of a program. As expected, the reduction rules for LJR are unchanged[4] from those of LJ (see figure 4-10).

4.2.5 Properties

LJR has a similar Type Soundness theorem as LJ:

Theorem 5 (LJR Type Soundness). *If* $\emptyset; \emptyset \vdash \texttt{e} : \texttt{E}$ *and* $\langle \texttt{e},\ \mathcal{S} \rangle \rightarrow^* \langle \texttt{e}',\ \mathcal{S}' \rangle$ *with* $\texttt{e}'$ *a normal form, then* $\texttt{e}'$ *is a value,* v*, such that* $\mathcal{S}'(v) = \langle \texttt{N},\ \mathcal{F} \rangle$ *and* $\emptyset \vdash \exists\emptyset.\texttt{mutable N} <: \texttt{E}$.

In the final the subtyping judgement of the type soundness theorem, the class type `N` is augmented with `mutable`. This step is needed so that the subtyping judgement may be applied and is safe because every object is in fact mutable at runtime.

The changes to store consistency are similar: class types, `N`, are converted to types by adding the fact that runtime objects are mutable. Additionally, field types, `F`, are converted to types using *mutability* assuming that the reference that reaches the field is mutable. This is a requirement to ensure that an object assigned to a field is saved to be later read when the field is reached through a mutable reference.

[3] An exception to this is the optional down casts, which are not modeled in LJR. See section 5.3.5.

[4] Technically, the call to *capture* is malformed because the last argument is a class type, which does not contain a top-level mutability, instead of a normal LJR type. This could be fixed by changing *capture* to accept class types for its last argument and replacing the predicate's subtyping judgement with a contains judgement on the class type's type arguments. We do not make this (solely syntactic) change to maintain similarity with WFJ's presentation of *capture*.

Expression typing:

$$\Delta; \Gamma \vdash \mathtt{x} : \mathit{snap}(\Gamma(\mathtt{x})) \qquad \text{(RT-Var)}$$

$$\frac{\Delta; \Gamma \vdash \mathtt{e}_0 : \mathtt{E}_0 \qquad \mathit{bound}_\Delta(\mathtt{E}_0) = \exists \Delta_0.\boxed{\mathtt{ML}_0}\ \mathtt{N}_0 \qquad \mathit{fields}(\mathtt{N}_0) = \overline{\mathtt{AF}}\ \boxed{\overline{\mathtt{F}}}\ \overline{\mathtt{f}} \qquad \boxed{\mathit{mutability}(\mathtt{F}_i,\ \mathtt{ML}_0) = \mathtt{A}} \qquad \mathit{snap}(\mathtt{A}) = \exists \Delta'.\mathtt{K}}{\Delta; \Gamma \vdash \mathtt{e}_0.\mathtt{f}_i : \exists \Delta_0 \Delta'.\mathtt{K}}\ \text{(RT-Get)}$$

$$\frac{\Delta \vdash \overline{\mathtt{P}}\ \mathtt{ok} \qquad \Delta; \Gamma \vdash \mathtt{e}_0 : \mathtt{E}_0 \qquad \mathit{bound}_\Delta(\mathtt{E}_0) = \exists \Delta_0.\boxed{\mathtt{ML}}\ \mathtt{N}_0 \qquad \Delta; \Gamma \vdash \overline{\mathtt{e}} : \exists \overline{\Delta}.\overline{\mathtt{K}} \qquad \Delta_1 = \Delta \Delta_0 \overline{\Delta} \qquad \mathit{mtype}(\mathtt{m}, \mathtt{N}_0) = \mathtt{<}\overline{\mathtt{Y}}\ \overline{\mathtt{B}}\mathtt{>}\overline{\mathtt{U}}\ \boxed{\mathtt{ML}'}\ \to \mathtt{U} \qquad \boxed{\mathtt{ML}\ \mathtt{N}_0 <: \mathtt{ML}'\ \mathtt{N}_0} \qquad \overline{\mathtt{V}} = \mathit{capture}_{\Delta_1}(\overline{\mathtt{P}}, \overline{\mathtt{Y}}, \overline{\mathtt{U}}, \overline{\mathtt{K}}) \qquad \Delta_1 \vdash \overline{\mathtt{V}} \in [\overline{\mathtt{V}}/\overline{\mathtt{Y}}]\mathtt{B} \qquad \Delta_1 \vdash \overline{\mathtt{K}} <: [\overline{\mathtt{V}}/\overline{\mathtt{Y}}]\overline{\mathtt{U}} \qquad \mathit{snap}([\overline{\mathtt{V}}/\overline{\mathtt{Y}}]\mathtt{U}) = \exists \Delta'.\mathtt{K}}{\Delta; \Gamma \vdash \mathtt{e}_0.\mathtt{<}\overline{\mathtt{P}}\mathtt{>}\mathtt{m}(\overline{\mathtt{e}}) : \exists \Delta_0 \overline{\Delta} \Delta'.\mathtt{K}}\ \text{(RT-Invk)}$$

$$\frac{\Delta \vdash \mathtt{N}\ \mathtt{ok} \qquad \mathit{fields}(\mathtt{N}) = \overline{\mathtt{AF}}\ \boxed{\overline{\mathtt{F}}}\ \overline{\mathtt{f}} \qquad \boxed{\mathit{mutability}(\overline{\mathtt{F}},\ \mathtt{mutable}) = \overline{\mathtt{A}}} \qquad \Delta; \Gamma \vdash \overline{\mathtt{e}} : \overline{\mathtt{E}} \qquad \Delta \vdash \overline{\mathtt{E}} <: \mathit{snap}(\overline{\mathtt{A}})}{\Delta; \Gamma \vdash \mathtt{new}\ \mathtt{N}(\overline{\mathtt{e}}) : \exists \emptyset.\mathtt{mutable}\ \mathtt{N}}\ \text{(RT-New)}$$

$$\frac{\Delta; \Gamma \vdash \mathtt{e}_0 : \mathtt{E}_0 \qquad \mathit{bound}_\Delta(\mathtt{E}_0) = \exists \Delta_0.\boxed{\mathtt{ML}_0}\ \mathtt{N}_0 \qquad \mathit{fields}(\mathtt{N}_0) = \overline{\mathtt{AF}}\ \boxed{\overline{\mathtt{F}}}\ \overline{\mathtt{f}} \qquad \boxed{\mathit{assignability}(\mathtt{AF}_i,\ \mathtt{ML}_0)} = \mathtt{assignable} \qquad \boxed{\mathit{mutability}(\mathtt{F}_i,\ \mathtt{ML}_0) = \mathtt{A}} \qquad \mathit{snap}(\mathtt{A}) = \exists \Delta_i.\mathtt{K} \qquad \Delta; \Gamma \vdash \mathtt{e}_v : \mathtt{E} \qquad \Delta \Delta_0 \vdash \mathtt{E} <: \exists \Delta_i.\mathtt{K} \qquad \mathtt{e}_b : \mathtt{E}'}{\Delta; \Gamma \vdash \mathtt{set}\ \mathtt{e}_0.\mathtt{f}_i = \mathtt{e}_v\ \mathtt{then}\ \mathtt{e}_b : \mathtt{E}'}\ \text{(RT-Set)}$$

Method typing:

$$\frac{\Delta \vdash \overline{\mathtt{B}'},\ \mathtt{T},\ \overline{\mathtt{T}}\ \mathtt{ok} \qquad \Delta = \overline{\mathtt{Y}} \in \overline{\mathtt{B}'},\ \overline{\mathtt{X}} \in \overline{\mathtt{B}} \qquad \Delta; \overline{\mathtt{x}} : \overline{\mathtt{T}}, \mathtt{this} : \boxed{\mathtt{ML}}\ \mathtt{C}\mathtt{<}\overline{\mathtt{X}}\mathtt{>}\ \vdash \mathtt{e}_0 : \mathtt{E} \qquad \Delta \vdash \mathtt{E} <: \mathit{snap}(\mathtt{T}) \qquad \mathtt{class}\ \mathtt{C}\mathtt{<}\overline{\mathtt{X}}\ \overline{\mathtt{B}}\mathtt{>} \lhd \mathtt{N}\ \{\ldots\} \qquad \mathit{override}(\mathtt{m}, \mathtt{N}, \mathtt{<}\overline{\mathtt{Y}}\ \overline{\mathtt{B}'}\mathtt{>}\overline{\mathtt{T}}\ \boxed{\mathtt{ML}}\ \to \mathtt{T})}{\mathtt{<}\overline{\mathtt{Y}}\ \overline{\mathtt{B}'}\mathtt{>}\ \mathtt{T}\ \mathtt{m}(\overline{\mathtt{T}}\ \overline{\mathtt{x}})\boxed{\mathtt{ML}}\ \{\ \mathtt{return}\ \mathtt{e}_0;\ \}\ \mathtt{OK\ IN}\ \mathtt{C}\mathtt{<}\overline{\mathtt{X}}\ \overline{\mathtt{B}}\mathtt{>}}\ \text{(RT-Method)}$$

Class typing:

$$\frac{\overline{\mathtt{X}} \in \overline{\mathtt{B}} \vdash \overline{\mathtt{B}},\ \mathtt{N},\ \overline{\mathtt{T}}\ \mathtt{ok} \qquad \overline{\mathtt{M}}\ \mathtt{OK\ IN}\ \mathtt{C}\mathtt{<}\overline{\mathtt{X}}\ \overline{\mathtt{B}}\mathtt{>}}{\mathtt{class}\ \mathtt{C}\mathtt{<}\overline{\mathtt{X}}\ \overline{\mathtt{B}}\mathtt{>} \lhd \mathtt{N}\ \{\overline{\mathtt{AF}}\ \boxed{\overline{\mathtt{F}}}\ \overline{\mathtt{f}};\ \overline{\mathtt{M}}\}\ \mathtt{OK}}\ \text{(RT-Class)}$$

Figure 4-16: Lightweight Javari (LJR) typing rules. Changes from Lightweight Java (LJ) (figure 4-9) are indicated by boxes.

$$
\begin{array}{ll}
 & \Delta;\Gamma \vdash_\sigma \mathcal{S} \iff \\
 & \quad \mathcal{S}(v) = \langle \texttt{mutable N},\ \mathcal{F} \rangle \Rightarrow \\
\Sigma_1: & \qquad \Delta \vdash \texttt{N} <: \Gamma(v) \\
\Sigma_2: & \qquad \text{and } \mathrm{dom}(\mathcal{F}) = \{\texttt{f} \mid \texttt{AF F f} \in \mathit{fields}(\texttt{N})\} \\
\Sigma_3: & \qquad \text{and } \mathrm{rng}(\mathcal{F}) \subseteq \mathrm{dom}(\mathcal{S}) \\
\Sigma_4: & \qquad \text{and } (\mathcal{F}(\texttt{f}) = v' \text{and } \texttt{AF F f} \in \mathit{fields}(\texttt{N}) \\
 & \qquad\quad \Rightarrow ((\mathcal{S}(v') = \langle \texttt{N}',\ \mathcal{F}' \rangle) \Rightarrow \Delta \vdash \texttt{mutable N}' <: \mathit{mutability}(\texttt{F},\ \texttt{mutable}))) \\
\Sigma_5: & \text{and } v \in \mathrm{dom}(\Gamma) \Rightarrow v \in \mathrm{dom}(\mathcal{S}) \\
\Sigma_6: & \text{and } \mathrm{dom}(\mathcal{S}) \subseteq \mathrm{dom}(\Gamma)
\end{array}
$$

LJR's subject reduction and progress theorems are unchanged from LJ's.

Chapter 5

Other Language Features

This chapter first discusses a simple template mechanism over mutability used to avoid code duplication. Next, section 5.2 presents Javari's mechanisms for dealing with reflection and serialization. Finally, section 5.3 briefly discusses language features adopted from Javari2004 [5], a previous dialect of Javari (see section 7.1). These features include full interoperability with Java, type-based analyses that build on reference immutability, handling of exceptions, and dynamic casts, an optional feature that substitutes run-time for compile-time checking at specific places in the program.

5.1 Templatizing methods over mutability to avoid code duplication

In Java, each (non-generic) `class` definition defines exactly one type. By contrast, in Javari, `class C { ... }` creates *two* types: `/*mutable*/ C` and `readonly C`. The type `/*mutable*/ C` contains some methods that are absent from `readonly C`, and a given method may have different signatures in the two classes (even though the method's implementation is otherwise identical).

Javari permits a programmer to specify the two distinct types using a single `class` definition, without any duplication of methods. (The alternative, code duplication, is unacceptable.)

Javari provides the keyword `romaybe` to declare that a method should be templated over the mutability of one or more formal parameters — that is, the method should have different signatures in the mutable and read-only types. If the type modifier `romaybe` is applied to any formal parameter (including `this`), then the type checker conceptually duplicates the method, creating two versions of it. (It is not necessary for two versions of a class or method to exist in the classfile or at run time.) In the first version of the method, all instances of `romaybe` are replaced by `readonly` (or `? readonly` in the case of type arguments). In the second version, all instances of `romaybe` are removed, corresponding to the mutability default case: mutable. For example, the following code defines a `DateCell` class:

```
class DateCell {
  Date value;
  romaybe Date getValue() romaybe { return value; }
}
```

Its effect is the same as writing the following (syntactically illegal[1]) code:

```
class DateCell {
  Date value;
  readonly    Date getValue() readonly    { return value; }
  /*mutable*/ Date getValue() /*mutable*/ { return value; }
}
```

As demonstrated above, the primary use of the `romaybe` keyword is to avoid declaring two duplicate field accessor methods, one readonly and one mutable. However, `romaybe` can also be used for static methods:

```
static romaybe Date cellDate(romaybe DateCell c) { return c.getValue(); }
```

`cellDate` returns a mutable `Date` if invoked on a mutable `DateCell` and returns a read-only `Date` if invoked on a read-only DateCell.

5.1.1 Parametric types

When used to modify a type argument, `romaybe` expands to `? readonly` instead of `readonly`. This behavior ensures that `romaybe` is useful for accessor methods of fields with parametric types, as shown below.

```
class Wheel {
  /*this-mutable*/ List</*this-mutable*/ Spoke> spokes;

  romaybe List<romaybe Spoke> getSpokes() romaybe { return spokes; }
}
```

The `getSpokes` method declaration is equivalent to the following two (syntactically illegal) method declarations, which are the correct type signatures for the accessors of `spokes`.

```
readonly List<? readonly Spoke> getSpokes() readonly    {
  return spokes;
}

/*mutable*/ List<mutable Spoke> getSpokes() /*mutable*/ {
  return spokes;
}
```

If the `romaybe` expanded to `readonly` in the case of the type argument, the read-only version of the method above would be illegally typed (see section 3). If this was the case, two versions of the method would need to be written, resulting in code duplication.

5.1.2 Discussion

In the majority of cases only one mutability type parameter (`romaybe`) is needed. Within a templated method, the mutability of the method parameters, including `this`, cannot affect the behavior of the method because the method must be valid for both read-only and mutable types. However, templated parameters can affect the types of the method's references,

[1]Method overloading based on parameter's mutability is illegal. If given different names, these methods would be legal.

including the method's return type. Hence, the only reason for a `romaybe` template is to ensure that the method has the most specific return type possible. In the examples above, a method's return type's mutability (including type arguments) depend on only the mutability of a single parameter, so only a single mutability type parameter is needed. Below is a contrived case that breaks this rule.

```
static Pair<romaybe1 Date, romaybe2 Date>
convert(romaybe1 DateCell c1, romaybe2 DateCell c2) {
  Pair<romaybe1 Date, romaybe2 Date> ret = ... ;
  ret.first  = c1.getDate();
  ret.second = c2.getDate();
  return ret;
}
```

The return type's mutability independently depends on both the mutabilities of `c1` and `c2`. If the types of the arguments to `convert` were `Date`s instead of `DateCell`s, this method would not require multiple mutability type parameters because it could be written using type parameters. We feel that rare cases, as the one above, do not warrant a more complicated mutability templating mechanism.

5.1.3 Template inference

As an alternative to explicitly specifying method templates, Javari could instead use type inference to create a version of a method with a read-only return type. Many languages, such as ML [30], use type inference to permit programmers to write few or no type annotations in their programs; this is especially important when the types are complicated to write. Javari could similarly infer method templates, reducing the number of template annotations (`romaybe`) in the code.

Lack of explicit immutability constraints would eliminate the documentation benefits of Javari, or would cause method signatures to describe what a method does rather than what it is intended to do. Furthermore, programming environments would need to re-implement the inference, in order to present the inferred types to users. Finally, inference is more difficult in object-oriented languages like Java then in simple functional languages like ML.

Despite these problems, there are countervailing advantages to inference. For example, although we have not found it onerous in our experience so far, it is a concern that many methods and arguments would be marked as read-only, cluttering the code. (Were backward compatibility not an issue, we would have chosen different defaults for our keywords.) In future work, we plan to implement such an inference and determine whether users find it helpful.

5.2 Code outside the type system

Certain Java constructs, such as reflection and serialization, create objects in a way that is not checked by the Java type-checker, but must be verified at run time. We discuss how to integrate checking of mutability with these mechanisms, even though mutability has no run-time representation.

5.2.1 Reflection

Reflection enables calling a method whose return type (including mutability) is unknown at compile time. This prevents checking immutability constraints at compile time. We desire to maintain type soundness (reflective calls that should return a `readonly` reference must do so) and flexibility (reflective calls that should return a mutable reference can do so).

In particular, consider the `invoke` method:

```
package java.lang.reflect;
class Method {
  // existing method in Java (and Javari):
  /*mutable*/ Object invoke(java.lang.Object, java.lang.Object...);
  // new method in Javari:
  readonly Object invokeReadonly(java.lang.Object, java.lang.Object...);
}
```

The three dots at the end of the parameter lists are not an ellipsis indicating elided code, but the Java syntax for variable-argument routines.

Javari requires programmers to rewrite some uses of `invoke` into `invokeReadonly`, where `invokeReadonly` returns a `readonly Object` rather than an `Object` as `invoke` does. For example:

```
Method m1 = ...;
Method m2 = ...;
/*mutable*/ Object o1 = m1.invoke(...);
readonly    Object o2 = m2.invokeReadonly(...);
```

`invokeReadonly` returns a read-only reference and does no special run-time checking. `invoke` returns a mutable reference, but performs a run-time check to ensure that the return type of the method being called is mutable. Note that this is a check of the invoked method's signature, *not* a check of the object or reference being returned. This checking is local and fast[2]. To enable this check, the JVM can record the mutability of the return type of each method as the method is loaded.

This proposal takes advantage of the fact that the type-checker knows the compile-time types of the arguments to `invoke`. That is, in a call `foo(d)`, it knows whether the declared type of `d` is read-only. That information is necessary for (1) ensuring that a readonly reference is not passed to a mutable formal parameter and (2) resolving overloading: determining whether `foo(d)` is a call to `foo(Date)` or to `foo(readonly Date)` when both exist.

5.2.2 Serialization

Like reflection, serialization creates objects in a way that is outside the purview of the Java type system. However, deserialization (`readObject`) is guaranteed to return a new object. Therefore, serializing a read-only object then deserializing the object, does not violate immutability constraints. Hence, serialization demands no special treatment by Javari.

[2]Unlike the immutability checking that is required to support general downcasts; section 5.3.5.

5.3 Retained language features

This section briefly explains several Javari features that have not changed from the Javari2004 dialect described in a previous paper and technical report [5, 4]. Full details can be found in those references.

5.3.1 Interoperability with Java

Javari is interoperable with Java and existing JVMs. The language treats any Java method as a Javari method with no immutability specification in the parameters (including `this`) or return type (and similarly for constructors, fields, and classes). Since the Javari type system does not know what a Java method can modify, it assumes that the method may modify anything.

While all Java methods can be called from Javari, Java code can only call Javari methods that does not use `readonly` in their return types. An implementation could enforce this by using standard Java names for mutable types, methods, and classes, and by "mangling" (at class loading time) the names of read-only types, methods, and classes into ones that cannot be referenced by legal Java code.

5.3.2 Type-based analyses

Javari enforces reference immutability — a read-only reference is never used to side-effect any object reachable from it. Reference immutability itself has many benefits. However, other guarantees may be desirable in certain situations. Four of these guarantees are object immutability (an object cannot be modified), thread non-interference (other threads cannot modify an object), parameter non-mutation (an object that is passed as a `readonly` parameter is not modified), and return value non-mutation (an object returned as a `readonly` result is not modified). One advantage of reference immutability is that a subsequent type-based analysis (which assumes that the program type checks [35]) can often establish these other properties from it, but the converse is not true.

Extending reference immutability to stronger guarantees requires escape analysis or partial information about aliasing. Determining complete, accurate alias information remains beyond the state of the art; fortunately, the analyses do not require full alias analysis. Obtaining alias information about a particular reference can be easier and more precise than the general problem [2]. Programmers can use application knowledge about aliasing, new analyses as they become available, or other mechanisms for controlling or analyzing aliasing, such as ownership types [12, 3, 7], alias types [3], linear types [49, 17], or checkers of pointer properties [15, 20].

5.3.3 Inner classes and read-only constructors

Javari protects a read-only enclosing instance from being mutated through an inner class. Placing `readonly` immediately following the parameter list of a method of an inner class declares the receiver *and all enclosing instances* of the receiver to be read-only.

Inner class constructors have no receiver, but placing the keyword `readonly` immediately following the parameter list of an inner class constructor declares all enclosing instances to be read-only. Such a constructor may be called a read-only constructor, by analogy with "read-only method". It is important to note that the "read-only" in "read-only constructor"

refers to the enclosing instance. Read-only constructors do not constrain the constructor's effects on the object being constructed, nor how the client uses the newly-constructed object.

It is a type error for a read-only method or constructor to change the state of the enclosing instance, which is read-only. Furthermore, a non-read-only method or constructor cannot be called through a read-only reference.

5.3.4 Exceptions

Javari prohibits read-only exceptions from being thrown. This restriction, which has so far caused no difficulty in practice, is caused by our desire for interoperability with the existing Java Virtual Machine, in which (mutable) `Throwable` is a supertype of every other `Throwable`. It is possible to modify Javari to lift the restriction on throwing read-only exceptions, but the result is complicated, introduces possibilities for error in the type system and the implementation, and provides little practical benefit.

5.3.5 Downcasts

Every non-trivial type system rejects some programs that are safe — they never perform an erroneous operation at run time — but whose safety proof is beyond the capabilities of the type system. Like Java itself, Javari allows such programs, but requires specific programmer annotations (downcasts); those annotations trigger Javari to insert run-time checks at modification points to guarantee that no unsafe operation is executed. Among other benefits, programmers need not code around the type system's constraints when they know their code to be correct, and interoperation with legacy libraries is eased. The alternatives — prohibiting all programs that cannot be proved safe, or running such programs without any safety guarantee — are unsatisfactory, and are also not in the spirit of Java.

If a program is written in the type-safe subset of Javari, then static type-checking suffices. For our purposes, the unsafe operation is the downcast, which converts a reference to a superclass into a reference to a subclass. (In Java but not in Javari (section 3.7), these can also appear implicitly in certain uses of arrays of references, for which Java's covariant array types prevent sound static type-checking.) Java inserts checks at each down-cast (and array store), and throws an exception if the down-cast fails.

Javari provides the following syntax for downcasting from a read-only type to a mutable type: "`(mutable)`*expression*". Regular Java-style casts may not be used to convert from read-only to mutable types:

```
readonly    Date rd;
/*mutable*/ Date md;
md = (Date) rd; // error: A regular Java cast may not cast away read-only-ness.
```

Special downcast syntax highlights that the cast is not an ordinary Java one, and makes it easy to find such casts in the source code.

Downcasting from a read-only to a mutable type triggers the insertion of run-time checks, wherever a modification (an assignment) may be applied to a reference that has had `readonly` cast away. (In the worst case, every assignment in the program, including libraries, must be checked.) The run-time checks guarantee that even if a read-only reference flows into a mutable reference, it is impossible for modifications to occur through the mutable reference. Thus, Javari soundly maintains its guarantee that a read-only reference cannot be used, directly or indirectly, to modify its referent.

A previous paper [5] describes an efficient technique for checking these casts at run time. It associates a "readonly" Boolean with each reference (*not* with each object). The readonly Boolean is true for each non-read-only reference derived from a `readonly` reference as a result of a downcast. The readonly Boolean is set when `readonly` is cast away, is propagated by assignments, and is checked whenever a modification (i.e., a field update) is performed on a non-read-only reference.

The following example illustrates the behavior of run-time casts.

```
class Foo {
  Date d;
  void setD() /*mutable*/ {
    this.d = new Date();
  }
}

          Foo f1 = new Foo();
readonly Foo rf = f1;
          Foo f2 = (mutable) rf;

f1.d = new Date();  // OK
f2.d = new Date();  // run-time error
f1.setD();          // OK
f2.setD();      // run-time error: at the first line of setD
```

In a previous experiment, there was approximately one cast per 1000 lines [5]. However, that experiment annotated existing code (without improving its design), did not annotate all the libraries that were used, and used an earlier dialect of the Javari language that lacked many features that make casts less necessary. Also lessening the need for casts, section 6 presents a technique for automatically converting existing Java libraries to Javari. Just as most new Java 5 code contains few Java casts, we believe that well-written new Javari code will contain few or no mutability downcasts.

Chapter 6

Type inference for Javari

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

We have created an algorithm that soundly calculates all the references (including local variables, method parameters, and static and instance fields) that may have Javari's `readonly`, `romaybe`, or `? readonly` keywords added to their declarations. Using this algorithm, programmers can convert Java programs, including libraries, to Javari, or can refine incompletely-annotated Javari programs.

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

We have implemented our algorithm in a tool called Javarifier. Javarifier' input is a Java (or partially annotated Javari) program in classfile format, because programmers may wish convert library code that is only available in classfile format. As a pre-pass, Javarifier heuristically recommends fields to declare `assignable` or `mutable`, and the user accepts some or all of the recommendations. Javarifier calculates and indicates to the user the references that may be declared `readonly`, `romaybe`, or `? readonly`. An example Java program and the corresponding inferred Javari program is shown in figure 6-1. Note that `hc` is heuristically recommended to — and in this case approved by — the user to be declared `assignable`. Otherwise, `hashCode` would be inferred, undesirably, to have a mutable receiver.

The rest of this chapter is organized as follows. Section 6.1 explains the algorithm that soundly infers read-only references. Section 6.2 extends the algorithm to arrays and

```
// Java
class Event {

  Date date;

  Date getDate() {
    return Date;
  }

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

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

}


// Javari
class Event {

  /*this-mutable*/ Date date;

  romaybe Date getDate() romaybe {
    return Date;
  }

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

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

}
```

Figure 6-1: A Java program (above) and the corresponding inferred Javari program (below).

generic types. Section 6.3 discusses how our algorithm is soundly extended to infer `romaybe` references. Section 6.4 provides heuristics that can be used to infer `assignable` and `mutable` fields. Section 6.5 describes the implementation of our algorithm within the Javarifier tool. Finally, section 6.6 reports our experience using Javarifier.

6.1 Inferring read-only references

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

The read-only inference algorithm declares all other fields not inferred to be read-only to be this-mutable. The read-only inference algorithm does not determine which fields are not a part of the object's abstract state and should be declared `mutable` or `assignable`. However, it is capable of inferring in the presence of fields that have been declared `mutable` or `assignable` by an outside source (see section 6.4).

For simplicity, we begin by describing our core algorithm (in section 6.1.1), i.e., the algorithm in the absence of unseen code, subtyping, user-provided reference immutability annotations (including assignable and mutable fields), arrays, and parametric types. We then extend the algorithm to allowing subtyping (section 6.1.2), and user-provided constraints and unseen code (section 6.1.3).

6.1.1 Core algorithm

Our algorithm generates, then solves, a set of mutability constraints for a program. A mutability constraint states when a given reference must be declared mutable. The algorithm uses two types of constraints: unguarded and guarded. Unguarded constraints state that a given reference is unconditionally mutable, e.g., "`x` is mutable." Guarded constraints state that a given reference is conditionally mutable if another reference is mutable, e.g., "if `y` is mutable then `x` is mutable." We use *constraint variables* to refer to references in these constraints—`x` and `y` in the previous examples. Unguarded constraints are represented by the constraint variable of the reference it is constraining, e.g., "`x`". Guarded constraints are represented as implications with the guard reference's constraint variable as the predicate and the dependent reference's constraint variable as the consequence, e.g., "`y` $\rightarrow$ `x`". (Section 6.3 introduces a third variety of constraints, *doubly-guarded constraints*.)

After generating the constraints, the algorithm solves the constraints yielding a simplified constraint set. The simplified constraint set is the set of all the constraint variables known to be true. If the guarded constraints are viewed as graph edges, then the core algorithm can be viewed as graph reachability starting at the unguarded constraints. A constraint variable is known to be true if it is present as an unguarded constraint or if it is consequence of a guarded constraint that is satisfied either directly by an unguarded constraint or indirectly though the consequence of a different satisfied guarded constraint. The simplified constraint set contains the set of references that must be declared mutable (or this-mutable in the case of instance fields); all other references may safely be declared read-only.

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

Figure 6-2: Grammar for core language used during constraint generation.

Constraint generation

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

We present constraint generation using a core language. The core language is a simple three-address programming language.[1] The grammar of the language is shown in figure 6-2. We use `Q` and `M` to refer to class and method definitions, respectively. `m` ranges over method names, `f` ranges over field names, and `s` ranges over allowed statements. `p`, `x`, `y`, and `z` range over variables (method parameters and locals). We use $\overline{\texttt{x}}$ as the shorthand for the (possibly empty) sequence $\texttt{x}_1 ... \texttt{x}_n$. We also include the special variable $\texttt{this}_\texttt{m}$, which refers to the receiver of method `m`. $\texttt{this}_\texttt{m}$ is treated as a normal variable, except that any program that attempts to reassign $\texttt{this}_\texttt{m}$ is malformed. Without loss of generality and for ease of presentation, we assume that all references and methods are given globally-unique names.

Control flow constructs are not modeled, because our 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 $\texttt{this}_\texttt{m}$. Static members are omitted because they do not demonstrate any interesting properties.

Each of the statements from figure 6-2 has a constraint generation rule, as shown in figure 6-3. The rules make use of the following auxiliary functions. `this(m)` and `params(m)` return the receiver reference ($\texttt{this}_\texttt{m}$) and parameters of method `m`, respectively. `retVal(m)` returns the constraint variable, $\texttt{ret}_\texttt{m}$, that represents the reference to `m`'s return value.

The constraint generation rules of figure 6-3 are as follows:

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

Invk The assignment, to `x`, of the return value of the invocation of method `m`, on `y`, with arguments $\overline{\texttt{y}}$, generates three kinds of constraints. These 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 $\texttt{this}_\texttt{m}$, each actual argument is assigned to the method's corresponding formal parameter, and the return value, $\texttt{ret}_\texttt{m}$, is assigned to `x`.

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

$$\mathtt{x} = \mathtt{y} : \{\mathtt{x} \rightarrow \mathtt{y}\} \text{ (ASSIGN)}$$

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

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

$$\mathtt{x} = \mathtt{y.f} : \{\mathtt{x} \rightarrow \mathtt{f},\ \mathtt{x} \rightarrow \mathtt{y}\} \text{ (REF)}$$

$$\mathtt{x.f} = \mathtt{y} : \{\mathtt{x},\ \mathtt{f} \rightarrow \mathtt{y}\} \text{ (SET)}$$

Figure 6-3: Constraint generation rules.

In more detail, the guarded constraint $\mathtt{this_m} \rightarrow \mathtt{y}$ is generated because the actual receiver must be mutable if `m` requires a mutable receiver. Similarly, the constraints $\overline{\mathtt{p}} \rightarrow \overline{\mathtt{y}}$ are generated because the i^{th} actual argument must be mutable if `m` requires the i^{th} formal parameter to be mutable. Finally, the constraint $\mathtt{x} \rightarrow \mathtt{ret_m}$ is generated to enforce that `m` must have a mutable return type if the value it returns is assign to a mutable reference.

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

Ref The assignment of `y.f` to `x` generates two constraints. The first, $\mathtt{x} \rightarrow \mathtt{f}$, is required because, if `x` is mutable, then the field `f` cannot be read-only. The second, $\mathtt{x} \rightarrow \mathtt{y}$, 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 non-read-only fields are this-mutable.)

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 $\mathtt{f} \rightarrow \mathtt{y}$ is added because if `f` is found to be this-mutable, then a mutable reference must be assigned to it.

The constraint set for a program is the union of the constraints generated for each line of the program. Figure 6-4 shows constraints generated for a sample program.

Constraint solving

The second phase of the algorithm solves the constraints by simplifying the constraint set. Constraint set simplification checks if any of the unguarded constraints satisfies (i.e., matches) the guard of a guarded constraint. If so, the guarded constraint is "fired" by removing it from the constraint set and adding its consequence to the constraint set as an unguarded constraint. The new unguarded constraint can then be used to fire other guarded constraints. Once no more constraints can be fired, constraint simplification terminates.

The unguarded constraints in the simplified constraint set are exactly the references that cannot be declared read-only. They must be declared this-mutable for instance fields,

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

Program constraints:

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

Figure 6-4: Example of constraint generation. After each line of code, the constraints generated and the constraint generation rule used for is shown.

```
class {
  readonly f;
  readonly foo(p) {
    /*mutable*/ x = p;
    readonly    y = x.f;
    readonly    z = x.foo(y);
    this.f = y;
    return y;
  }
}
```

Figure 6-5: The result applying our algorithm's output to the program in figure 6-4.

and mutable in the case of all other types of references. All other references may safely be declared read-only.

The constraints generated from the example program in figure 6-4 will be simplified as shown below. (For clarity, the unguarded constraints are listed first.)

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

At the end of simplification, the unguarded constraints $\texttt{this}_{\texttt{foo}}$, x and p are present in the simplified constraint set. These references may not be declared read-only. All the other references, y, z, $\texttt{ret}_{\texttt{foo}}$, and f, can be declared read-only. Figure 6-5 shows the result of applying our algorithm to the example program.

6.1.2 Subtyping

Java and Javari allow subtyping polymorphism, which enables multiple implementations of a method to be specified through overriding.[2] Javari requires that all implementations of an overridden method have identical signatures, including the mutability of parameters. Therefore, our algorithm must infer the same mutabilities for the parameters of all implementations of overridden methods. Overridden methods are also restricted to have covariant return types.

To avoid inferring different signatures for polymorphic methods, our algorithm places additional constraints in the constraint set. For every parameter of a overloaded method, we add the constraint that it is equal in mutability to every other implementation of the method. Type equality of two variables `a` and `b` is represented by two guarded constraints: `a` $\rightarrow$ `b` and `b` $\rightarrow$ `a`. For example, below, the method `toString` is overloaded.

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

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

Thus, the mutability of the `this` parameter of `Event`'s and `Birthday`'s `toString` methods must be the same. This requirement generates the constraints:

$$\texttt{this}_{\texttt{Event.toString}} \rightarrow \texttt{this}_{\texttt{Birthday.toString}}$$
$$\texttt{this}_{\texttt{Birthday.toString}} \rightarrow \texttt{this}_{\texttt{Event.toString}}$$

To preserve overloading, both methods must be declared to have mutable `this` parameters, even though only `Birthday`'s `toString` method actually mutates its receiver. This inconsistency might indicate an error (if the specification prohibits or requires the modification), or might indicate that only some implementations perform a permitted side effect. Our algorithm could issue a warning in such cases or make both receivers read-only using the technique of declaring select fields to be mutable or assignable given in section 6.4.1. For example, the technique would state that `Birthday`'s `string` field should be declared assignable, and doing so would allow `Birthday`'s `toString` along with `Event`'s to be read-only, the expected result.

[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 *abstract methods*.

6.1.3 User-provided annotations and unanalyzed code

Our read-only inference algorithm can be extended to incorporate user-provided annotations. This capability is required because the read-only inference algorithm is not capable of directly inferring the `assignable` and `mutable` keywords. Additionally, user-provided annotations are needed for native method invocations, on which our algorithm cannot operate. Finally, a user may, using knowledge of the program, wish to override the annotations inferred by our algorithm.

A user (or another analysis) may specify that instance fields are this-mutable, read-only, or mutable, and that other references (static fields, local variables and parameters) are read-only or mutable.

A user declaring a reference to be read-only causes the algorithm, upon finishing, to check if it is safe to declare the given reference as `readonly`. If not, the algorithm issues an error, stating the conflict with the user annotation, or recommend ways to change the code, discussed in section 6.4.1, such that reference would be safe to declare be `readonly`. It is expected that this type of declaration will be particularly useful for overridden methods (see section 6.1.2) because declaring a supertype's method to be read-only would propagate to all the subtypes. For example, a user could annotate `Object`'s `toString`, `hashCode`, and `equals` methods as having read-only `this` parameters. The user would then be notified if any class's implementation of the above methods are not read-only. Such cases may arise from errors or the usage of a cache fields that should be declared `assignable`.

A user declaring a non-field reference mutable or a field this-mutable results in the system adding an unguarded constraint that the reference cannot be read-only. This specification is useful when a programmer knows that a certain reference is not currently used to modify its referent, but that it may be used to do so in the future.

The core algorithm makes the closed-world assumption: it assumes that all code is seen and, therefore, it is safe to change types of returned/escaped values such as public method return types and types of public fields. The closed world assumption yields more precise results when the algorithm processes the whole program. By contrast, the open-world assumption is required when analyzing partial programs or library classes with unknown clients, because an unseen client may rely on the mutability of a field or return value. In open-world mode, our algorithm marks as mutable (i.e., adds an unguarded constraint for) every non-private field and non-private method return value. 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.

Assignable and mutable fields

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

To handle `assignable` fields, the SET rule is divided into two rules, SET-A and SET-N, which depend on the assignability of the field. If the field is `assignable`, SET-A does not add the unguarded constraint that the reference used to reach the field must be mutable:

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

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

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

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

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

Figure 6-6: Modified constraint generation rules for handling assignable and mutable fields. The original Set rule (shown in figure 6-3) is replaced by Set-A and Set-N. The original Ref rule is replaced by Ref-M and Ref-N. Finally, Mutable is added to the remaining rules from figure 6-3.

an `assignable` field may be assigned through either a read-only or mutable reference. If the field is not `assignable`, Set-N proceeds as normal.

To handle `mutable` fields we add the constraint generation rule Mutable, which adds an unguarded constraint for each `mutable` field. The Ref rule is again divided into two rules: Ref-M and Ref-N depending on the mutability of the field. If the field is `mutable`, then Ref-M does not add any constraints because, when compared to the original Ref rule, (1) the consequence of the first constraint, $\texttt{x} \rightarrow \texttt{f}$, has already been added to the constraint set via the Mutable rule, and (2) the second constraint, $\texttt{x} \rightarrow \texttt{y}$, is eliminated because a `mutable` field is mutable regardless of how it is reached. If the field is not `mutable`, then Ref-N proceeds as normal.

6.2 `? readonly`

This section discusses how to infer types for arrays and generic classes. The key difficulty is inferring the `? readonly` type, which requires inferring two types (an upper and a lower bound) for each array/generic class. If the bounds are different, then the resulting Javari type is `? readonly`.

6.2.1 Arrays

We now extend our algorithm to handle arrays. First, we extend our core language grammar to allow storing to and reading from arrays. The extended grammar is shown in figure 6-7.

Javari allows array elements to have two-sided bounded types (section 3.7). For example, the array `(? readonly Date)[]` has elements with upper bound `readonly Date` and lower bound `mutable Date`. All array element types can be written in the form of having an upper bound and a lower bound. For example, `(readonly Date)[]` has elements with identical

Grammar:

```
s  ::=  ...
     |  x[x] = x
     |  x = x[x]
```

Figure 6-7: Core language grammar extended for arrays. The original grammar is shown in figure 6-2.

```
T, S  ::=  A | C          types
A, B  ::=  T[]        array types
C, D                  class names
```

Figure 6-8: Type meta-variables.

upper bound and lower bound `readonly Date`. Our algorithm infers an upper bound and a lower bound for the type of each array's elements.

In the core algorithm, a reference could only have a single read-only-ness annotation; therefore, using a single constraint variable for each reference sufficed. The introduction of arrays, however, enables multiple read-only-ness annotations to be placed on a reference's type: the array's elements' type can be annotated in addition to the array type. Therefore, our analysis must constrain every part of a type. By "part", we refer to a reference's top-level array type; the upper and lower bounds of the elements of the top-level array type; if the elements of the top-level array are array themselves, then 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; $\texttt{Date[]}_{\triangleleft}$, the upper bound of the element type, and $\texttt{Date[]}_{\triangleright}$, the lower bound of the element type; and then `Date` types corresponding to the upper bound of the upper bound, the upper bound of the lower bound, the lower bound of the upper bound, and the lower bound of the lower bound. To constrain each part of the type, we will use each type part as a constraint variable. To distinguish type parts of an upper bound from the corresponding lower bound, we subscript upper bounds with $_{\triangleleft}$ and lower bounds with $_{\triangleright}$. Additionally, we assume that within a program, textually different instances of the same type are distinguishable.

As in section 4.1, `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 meta-variables are shown in figure 6-8.

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

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 6-9[3].

We modify the constraint generation rules to use types as constraint variables and to enforce the subtyping relationship across assignments including the implicit assignments that occur during method invocation. The extended rules are shown in figure 6-10.

[3]In Java, arrays are covariant; however, in Javari, array are invariant in respect to mutability (see section 3.7), therefore, we use the contains relationship as Java's parametric types do.

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

Figure 6-9: Simplified subtyping (<:) rules for mutability in Javari. The simplified rules only check the mutabilities of the types, because we assume the program being converted type checks under Java. An array element's type, T, is said to be contained by another array element's type, S, written T ⊂: S, if the set of types denoted by T is a subset of the types denoted by S.

$$\texttt{x} = \texttt{y} : \{\mathit{type}(\texttt{y}) <: \mathit{type}(\texttt{x})\} \text{ (ASSIGN)}$$

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

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

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

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

$$\texttt{x} = \texttt{y[z]} : \{\mathit{type}(\texttt{y[z]}) <: \mathit{type}(\texttt{x})\} \text{ (ARRAY-REF)}$$

$$\texttt{x[z]} = \texttt{y} : \{\mathit{type}(\texttt{x}),\ \mathit{type}(\texttt{y}) <: \mathit{type}(\texttt{x[z]})\} \text{ (ARRAY-SET)}$$

Figure 6-10: Constraint generation rules extended for arrays.

Type well-formedness constraints

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

Constraint solving

Before the constraint set can be simplified as before, subtyping and contains constraints must be reduced to guarded constraints. To do so, each subtyping or contains constraint is replaced by the corresponding rule's predicates (see figure 6-9). This step is repeated until only guarded and unguarded constraints remain in the constraint set. For example, the statement x = y, where x and y have the types T[] and S[], respectively, would generate and reduce constraints as follows:

$$
\begin{array}{lcll}
\texttt{T}, \texttt{S} & ::= & \texttt{C<}\overline{\texttt{T}}\texttt{>} \mid \texttt{X} & \textit{types} \\
\texttt{C}, \texttt{D} & & \textit{class names} & \\
\texttt{X}, \texttt{Y} & & \textit{type variables} &
\end{array}
$$

Figure 6-11: Type meta-variables.

$$
\begin{array}{rcl}
\texttt{x} = \texttt{y} & : & \{\textit{type}(\texttt{y}) <: \textit{type}(\texttt{x})\} \\
& : & \{\texttt{S[]} <: \texttt{T[]}\} \\
& : & \{\texttt{T[]} \rightarrow \texttt{S[]},\ \texttt{S} \subset: \texttt{T}\} \\
& : & \{\texttt{T[]} \rightarrow \texttt{S[]},\ \texttt{S}_{\triangleleft} <: \texttt{T}_{\triangleleft},\ \texttt{T}_{\triangleright} <: \texttt{S}_{\triangleright}\} \\
& : & \{\texttt{T[]} \rightarrow \texttt{S[]},\ \texttt{T}_{\triangleleft} \rightarrow \texttt{S}_{\triangleleft},\ \texttt{S}_{\triangleright} \rightarrow \texttt{T}_{\triangleright}\}
\end{array}
$$

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

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

Applying results

Finally, we must apply the results back to the initial Java program. The results are applied to top-level types, the same way they were before. However, for element types, we must map the constraints on the upper bound and type lower bound to a single Javari type. If the upper bound and lower bound are both present in the constraint set, the element's type is annotated as mutable. If the upper bound and lower bound are both not present, then the element's type is annotated as read-only. If the lower bound is present, but the upper bound is not, then the element's type is annotated as `? readonly`. Finally, the case that the upper bound is present and the lower bound is not, cannot occur due to the well-formedness constraints.

6.2.2 Parametric types

Parametric types are handled in a similar fashion as arrays. For a parametric type, constraint variables must be made for the upper and lower bound of each type argument to a parametric class. As with arrays, we use type parts as constraint variables.

We will use the following meta-syntax to represent parametric types. As in section 4.1, `T`, `S`, `U`, and `V` will range over types, `X` and `Y` over type variables, and `C` and `D` over class names. We do not model wildcard types other than `? readonly` in this section. The type meta-variable definitions are shown in figure 6-11.

As with arrays, we use $_{\triangleleft}$ to denote type arguments' upper bounds and $_{\triangleright}$ to denote their lower bounds.

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

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

Figure 6-12: *asType* converts a type (first argument) to one of its superclasses (second argument).

Auxiliary functions

Our type generation rules use the auxiliary function, $bound_{\Delta}$, as previously presented in section 4.1. $bound_{\Delta}(\texttt{T})$ returns the declared upper bound of `T` if `T` is a type variable; if `T` is not a type variable, `T` is returned unchanged. In our formulation, we assume access to a global type environment, Δ, that maps type variables to their declared bounds. *bound* ignores any upper bound or lower bound subscripts present on the type.

As with arrays, our 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 6-12) to return `C`'s super type of class `D`[4]. Parametric types require this function because `C` could be a subtype of any particular parametric instantiation of `D`. Furthermore, the instantiation of `D` can be depended on the instantiation of `C` as in the following case: `C<T> extends D<T>`.

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* is used to convert the value's type to have the same class as the reference. For example,

```
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* is used to convert `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 (in this case they must be mutable) affect the type that class `Foo` is declared to extend instead of the type of `f`'s elements. The two possibilities are shown below:

```
class Foo<T> extends List</*mutable*/ Date> // mutable applied to correct type.

Foo</*mutable*/ Integer> f; // mutable applied to incorrect type
```

Constraint generation

As with arrays, the constraint generation rules (shown in figure 6-13) use subtyping constraints. However, the subtyping rules (shown in figure 6-14) are extended to deal with type variables. A type variable is not allowed to be annotated as mutable (see section 3.6.2);

[4]We use type to describe the first argument because the type arguments to the type are present. We use class to describe the second argument because type arguments are not provided.

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

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

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

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

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

Figure 6-13: Constraint generation rules in presence of parametric types.

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

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

Figure 6-14: Simplified subtyping rules for mutability in the presence of parametric types.

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, the 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 6-15.

Type well-formedness constraints

As with arrays, in addition to the constraints from the constraint generation rules, well-formedness constraints are added to the constraint set. As before, a constraint is added that a type argument's lower bound must be a subtype of the type argument's upper bound. Parametric types, additionally, introduce the well-formedness constraint that a type argument's upper bound (and, therefore, by transitivity, lower bound) is a subtype of the corresponding type variable's declared bound.

Constraint simplification and applying results

As with arrays, subtyping (and contains) constraints are simplified into guarded constraints by removing 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.

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

Figure 6-15: The results of applying our 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`. Note that one may not apply the mutable annotation directly to `f`'s type because a type parameter cannot be annotated as `mutable` (section 3.6.3).

$$
\begin{array}{rcl}
\texttt{M} & ::= & \texttt{<}\overline{\texttt{X}} \triangleleft \overline{\texttt{T}}\texttt{>}\ \texttt{m}(\overline{\texttt{T}}\ \overline{\texttt{x}})\{\overline{\texttt{s;}}\} \\
\texttt{s} & ::= & \texttt{s} \\
 & | & \texttt{x} = \texttt{x.m}\texttt{<}\overline{\texttt{T}}\texttt{>}(\overline{\texttt{x}})
\end{array}
$$

Figure 6-16: Core grammar extended for parameterized methods.

Javari does not allow raw types, and our analysis is incapable of operating on code that contains raw types.

6.2.3 Parametric methods

In addition to classes, Java allows methods to be parameterized. Our type inference algorithm assumes that method invocation type arguments are explicitly recorded. Later in this section we discusses loosening this restriction.

To represent parameterized methods, we extend our core grammar as shown in figure 6-16. The constraint generation rule's method invocation rule is extended as shown in figure 6-17. The rule is extended to substitute the method invocation's type arguments for the method's type parameters in generated constraints. Additionally, the rule enforces the well-formedness constraint that the actual type argument must be a subtype of the type parameter's declared bound (last constraint listed).

Type inference

The formalization above assumes that type arguments to method invocations are explicitly given. In practice, this limitation is not practical because (1) Java allows method invocation type arguments to be implicit within source code, (2) the Java classfile format does not record method invocation type parameters even when explicitly given in the source code, and (3) due to capture conversion (see section 4.1.2), it is impossible to explicitly represent all method invocation type arguments using source or classfile expressible types. Thus, type inference for method invocation type arguments must be performed as a pre-step to applying the constraint generation rules. In this section we present how such a type inference could be conducted. However, this type inference should not be considered directly a part of the read-only-ness analysis.

The type inference algorithm presented assumes types are known for method signatures and fields but not local variables or method invocation type arguments. These assumptions

$$\frac{\texttt{this(m)} = \texttt{this}_\texttt{m} \qquad \texttt{params(m)} = \overline{\texttt{p}} \qquad \texttt{retVal(m)} = \texttt{ret}_\texttt{m} \qquad \texttt{typeParams(m)} = \overline{\texttt{X}}}{\begin{array}{l}\texttt{x} = \texttt{y.m<}\overline{\texttt{T}}\texttt{>}(\overline{\texttt{y}}) : \{\mathit{type}(\texttt{y}) <: \mathit{type}(\texttt{this}_\texttt{m}),\ \mathit{type}(\overline{\texttt{y}}) <: [\overline{\texttt{T}}/\overline{\texttt{X}}]\mathit{type}(\overline{\texttt{p}}), \\ \qquad\qquad [\overline{\texttt{T}}/\overline{\texttt{X}}]\mathit{type}(\texttt{ret}_\texttt{m}) <: \mathit{type}(\texttt{x}),\ \overline{\texttt{T}} <: [\overline{\texttt{T}}/\overline{\texttt{X}}]\mathit{bound}_\Delta(\overline{\texttt{X}})\}\end{array}} \text{ (INVK)}$$

Figure 6-17: The core language grammar and method invocation constraint generation rule extended to handle parametric methods with explicate type arguments.

$$\begin{array}{rcll}
\texttt{Q} & ::= & \texttt{class}\ \{\overline{\texttt{f}}\ \overline{\texttt{M}}\} & \\
\texttt{M} & ::= & \texttt{<}\overline{\texttt{X}} \triangleleft \overline{\texttt{T}}\texttt{>}\ \texttt{m}(\overline{\texttt{T}}\ \overline{\texttt{x}})\{\overline{\texttt{s;}}\} & \\
\texttt{s} & ::= & \texttt{x} = \texttt{x} & \\
 & | & \texttt{x.m<}\overline{\texttt{T}}\texttt{>}(\overline{\texttt{x}}) & \\
 & | & \texttt{return x} & \\
 & | & \texttt{x} = \texttt{x.f} & \\
 & | & \texttt{x.f} = \texttt{x} &
\end{array}$$

$$\begin{array}{rcll}
\texttt{T,S} & ::= & \texttt{C<}\overline{\texttt{T}}\texttt{>} \mid \texttt{X} \mid \texttt{A} & \textit{types} \\
\texttt{A,B} & ::= & \texttt{T[]} & \textit{array types} \\
\texttt{C,D} & & \textit{class names} & \\
\texttt{X,Y} & & \textit{type variables} &
\end{array}$$

Figure 6-18: Core grammar for method invocation type argument inference.

are consistent with the information present in Java classfiles.[5] As with the constraint generation rules, we present our analysis using using our core grammar, which is reviewed in figure 6-18.

Similar to our read-only-ness analysis, the type inference applies a type inference rule to each statement of the program. However, instead of generating constraints to be solved, the type inference rules directly add types to local variables and method invocation type arguments. No constraints are generated or solved. Unlike the read-only-ness analysis, the type inference is applied repeatedly until a fixed-point is found.

Our type inference algorithm is not similar to the Hindley-Milner type inference algorithm. Since we start with the types of fields and the signatures of methods, our job is much simpler. We simply propagate types from typed references (fields, method parameters, and return values) to untyped references (local variables and method invocation type arguments).

The type inference rules are shown in figure 6-19. Our formalization uses the expression $\texttt{x} \Leftarrow: \texttt{y}$ to represent assigning x to have the type of y if and only if x's type is unknown and y's type is known. With the exception of the method invocation rule, the rules are straightforward.

The method invocation rule uses the auxiliary function *inferTypeArgs* (figure 6-20) to infer types of the method invocation's type arguments. We use $\overline{\texttt{?}}$ to represent the unknown types of the method invocation type arguments.

The *inferTypeArgs* auxiliary function calculates the method invocation's type arguments

[5] Additional precision can be obtain by using a classfile's "localVariableTypeTable," which is present if the source file was compiled using Javac's debug flag.

$$\texttt{x} = \texttt{y} : \{\texttt{x} \Leftarrow: \texttt{y},\ \texttt{y} \Leftarrow: \texttt{x}\}$$

$$\texttt{x.f} = \texttt{y} : \{\texttt{y} \Leftarrow: \texttt{f}\}$$

$$\texttt{x} = \texttt{y.f} : \{\texttt{x} \Leftarrow: \texttt{f}\}$$

$$\frac{\mathit{inferTypeArgs}(\texttt{typeParams(m)},\ \mathit{type}(\texttt{params(m)}),\ \mathit{type}(\overline{\texttt{y}}),\ \mathit{type}(\texttt{retVal(m)}),\ \mathit{type}(\texttt{x})) = \overline{\texttt{T}}}{\texttt{x} = \texttt{y.m<}\overline{\texttt{?}}\texttt{>}(\overline{\texttt{y}}) : \{\overline{\texttt{?}} \Leftarrow: \overline{\texttt{T}},\ \texttt{x} \Leftarrow: [\overline{\texttt{T}}/\texttt{typeParams(m)}]\texttt{retVal(m)}\}}$$

$$\texttt{return x} : \{\texttt{x} \Leftarrow: \texttt{retVal(m)}\}$$

Figure 6-19: Type inference rules.

by examining the types of the method invocation's arguments and return value, and the bounds placed on the method's type parameters. We build intuition for our algorithm by presenting a series of example method invocations and their inferred type arguments. We use static methods because the receivers of instance methods do not affect the inference; furthermore, for simplicity, we use only a single type parameter per method.

The simplest case is when the method has a single formal parameter that has the method's type parameter as its type. In this case, the type of the actual argument is the method invocation type argument as shown below:

```
static <T> void foo(T t);
Integer i;
foo(i); // Type argument inferred to be Integer.
```

A slightly more complicated case occurs when the formal parameter's type uses the type parameter as a type argument to another type. In this case, care must be taken to match the type parameter to the correct part of the actual argument's type:

```
static <T> void bar(List<T> lst);
List<Float> f;
bar(f); // Type argument inferred to be Float.
```

Our formalization uses the *matchTypes* auxiliary function, which is implemented using *location* and *getPart* auxiliary functions, to perform the task of matching the type parameter to the correct part of actual argument's type. Note that in some cases, a type parameter can appear in multiple locations within a type. Thus, *matchTypes* can return a set of types:

```
static <T> void printGraph(Map<T, Set<T>> nodeToChildren);
Map<Integer, Set<Integer>> g;
printGraph(g); // Type argument inferred to be Integer.
```

In this particular case, both types were the same, `Integer`. However, in the presence of wildcards, the types may differ. In the case that the types differ, the common super type the set of the types is taken (see next example).

matchTypes must be careful in the case of subtyping to match the type parameters correctly because subclasses can have different type parameters from their superclass. Take the following example:

$$\frac{\mathit{inferFromArgs}(\mathtt{X},\ \overline{\mathtt{T}},\ \overline{\mathtt{S}}) = \mathtt{T}'}{\mathit{inferTypeArgs}(\mathtt{X} :: \overline{\mathtt{X}},\ \overline{\mathtt{T}},\ \overline{\mathtt{S}},\ \mathtt{U},\ \mathtt{V}) = \mathtt{T}' :: \mathit{inferTypeArgs}(\overline{\mathtt{X}},\ \overline{\mathtt{T}},\ \overline{\mathtt{S}},\ \mathtt{U},\ \mathtt{V})}$$

$$\frac{\mathit{inferFromArgs}(\mathtt{X},\ \overline{\mathtt{T}},\ \overline{\mathtt{S}}) = \bullet \qquad \mathit{matchTypes}(\mathtt{X},\ \mathtt{U},\ \mathtt{V}) = \mathtt{T}'}{\mathit{inferTypeArgs}(\mathtt{X} :: \overline{\mathtt{X}},\ \overline{\mathtt{T}},\ \overline{\mathtt{S}},\ \mathtt{U},\ \mathtt{V}) = \mathtt{T}' :: \mathit{inferTypeArgs}(\overline{\mathtt{X}},\ \overline{\mathtt{T}},\ \overline{\mathtt{S}},\ \mathtt{U},\ \mathtt{V})}$$

$$\frac{\mathit{inferFromArgs}(\mathtt{X},\ \overline{\mathtt{T}},\ \overline{\mathtt{S}}) = \bullet \qquad \mathit{matchTypes}(\mathtt{X},\ \mathtt{U},\ \mathtt{V}) = \bullet \qquad \mathit{bound}_{\Delta}(\overline{\mathtt{X}}) = \mathtt{T}'}{\mathit{inferTypeArgs}(\mathtt{X} :: \overline{\mathtt{X}},\ \overline{\mathtt{T}},\ \overline{\mathtt{S}},\ \mathtt{U},\ \mathtt{V}) = \mathtt{T}' :: \mathit{inferTypeArgs}(\overline{\mathtt{X}},\ \overline{\mathtt{T}},\ \overline{\mathtt{S}},\ \mathtt{U},\ \mathtt{V})}$$

$$\frac{\mathit{LUB}(\mathit{matchTypes}(\mathtt{X},\ \overline{\mathtt{T}},\ \overline{\mathtt{S}})) = \mathtt{U}}{\mathit{inferFromArgs}(\mathtt{X},\ \overline{\mathtt{T}},\ \overline{\mathtt{S}}) = \mathtt{U}}$$

$$\frac{\mathit{location}(\mathtt{X},\ \mathtt{T}) = \overline{\mathtt{loc}} \qquad \mathit{getPart}(\mathit{asType}_{\Delta}(\mathtt{S}, \mathtt{T}),\ \overline{\mathtt{loc}}) = \overline{\mathtt{U}}}{\mathit{matchTypes}(\mathtt{X},\ \mathtt{T},\ \mathtt{S}) = \overline{\mathtt{U}}}$$

Figure 6-20: Auxiliary functions for inferring method invocation type arguments. The arguments of *inferTypeArgs* are as follows: $\mathtt{X} :: \overline{\mathtt{X}}$ is the type parameters of the invocated method, $\overline{\mathtt{T}}$ is the types of the method's parameters, $\overline{\mathtt{S}}$ is the types of the actual arguments to the method invocation, `U` is the return type of the method, and `V` is the type of the reference to which the method's return value is assigned. $\mathtt{X} :: \overline{\mathtt{X}}$ denotes concatenation of `X` to $\overline{\mathtt{X}}$. Thus, *inferTypeArgs* infers the method invocation's type argument for each method type parameter one-by-one, concatenating the result into *inferTypeArgs*'s result.

```
class <T> Quax extends List<Number> { ... }
static <S> foo(List<S> x);
Quax<Date> q;
foo(q); // Type argument inferred to be Number not Date.
```

matchTypes must be sure to match `S` with `Number`, the type argument to `List`, and not to `Date`, the type argument to `Quax`. *matchTypes* ensures the correct mapping by using the *asType* auxiliary function (section 6.2.2).

In the case of a method with multiple formal parameters, each actual argument could indicate a different type for the type argument. In such a case, the type argument is inferred to be the common supertype of each type:

```
static <T> baz(T x, T y);
Integer i;
Float f;
bar(i, f); // Type argument inferred to be Number.
```

Calculating the common supertype of the types inferred from each of the method invocation's actual arguments in handled by the auxiliary function LUB. The same invocation of LUB also calculates the common supertype of the types inferred from different parts of an actual argument's type (see last example).

At this point, we have explained how method invocation type arguments can be inferred from examining the types of the method invocation's actual arguments. This technique is

encapsulated by the *inferFromArgs* auxiliary function. However, the general *inferTypeArgs* must also handle the (less prevalent) cases where a method invocation's type arguments cannot be determined by examining the types of the invocation's actual arguments. These cases can arise when "untyped" `null` is passed a method's argument or when a parameterized method does not use its type parameters in the types of its formal parameters.

A `null` value is untyped when passed directly as an actual argument to a method invocation: `foo(null);`. A `null` value is typed when it is assigned to a typed reference before hand:

```
Date d = null;
foo(null);
```

In the case that a `null` value is typed, the *inferFromArgs* previously described is applicable; thus, the remaining examples will only deal with untyped `null`s.

First, we examine the case that no information is known about the types of a method invocation's actual arguments, but the type of the return value is known. Take for example, the following method invocation:

```
static <T> T foo(T x);
Integer i = foo(null); // Type argument inferred to be Integer
```

In this case, we able to infer from the use of `foo`'s return value, that the method invocation's type argument is `Integer`.

Next, we examine the case where no information is know about the types of a method invocation's actual arguments or return value:

```
static <T extends Number> void bar(T x);
bar(null); // Type argument inferred to be Number
```

In this case, we are forced to rely on the upper-bound placed on `T` to serve as the inferred type.

From these examples we can see that *inferTypeArgs* should determine the type of a method invocation's type arguments by first using the actual arguments of a method invocation; then, if that fails, try to use the method's return value; and in the case that both of the above fail, use the type parameter's declared bound. This intuition is reflected in the three *inferTypeArgs* rules.

Finally, we conclude our discussion of method invocation type argument inference with a comparison to previous work on the subject. Section 6 of *GJ: Extending the Java programming language with type parameters* presents a similar type inference algorithm. Their step of calculating "the smallest type that yields a valid call" is similar to our use of LUB to calculate the smallest common supertype of the types suggested by each of the actual argument's types. Thus, we ensure that the arguments passed to the method yields a valid call. However, our technique ignores other restrictions that are required to yield a valid call We do not check that the common subtype found is a subtype of the declared bound on the type parameter or a subtype of a reference to which the return value is assigned. Our algorithm does not consider these cases, because we assume the code we are inferring on has already type checked.

The algorithms differ in how they proceed if a there does not exist type information about a method invocation's actual arguments. Where we use the method's return type or the type parameter's bound to determine the type, they declare that such a type argument has the special type "$*$". This special type is a subtype of every other type and, therefore,

can be used to assign the result of the method to any type needed. To prevent this special type from creating typing loop-hole (see their paper), there is the restriction that the special type may only appear once in the return type.

Instead of using * as unknown type, our algorithm calculates the type * should be instantiated at. The advantage of this technique, is that we can allow * to appear multiple places within the return type, as long as allow instances of * are calculated to have the same type. Experimenting with Javac seems to indicate that it uses an approach similar to ours. Note, however, unlike the GJ previous work, our approach is only correct when the code is known to type check. The actual algorithm used by Javac is most likely more complicated than the one given in this paper or in the GJ paper.

Limitations

In the case of poorly parameterized code, this approach is not guaranteed to always provide the most general methods[6], or, therefore, the maximum number of read-only annotations. We are unable to always infer the most general method because in some cases one does not exist. For example, take the Java method below:

```
void addDate(List<Date> l, Date d) {
   l.add(d);
}
```

Three type-correct Javari methods can be inferred (our algorithm as stated above infers the first):

```
addDate(/*mutable*/ List<readonly    Date> l, readonly    Date d)
addDate(/*mutable*/ List<? readonly  Date> l, /*mutable*/ Date d)
addDate(/*mutable*/ List</*mutable*/ Date> l, /*mutable*/ Date d)
```

The last method is strictly less general than the second because the second can be applied to list of read-only elements in addition to lists of mutable elements. However, the first and second methods are incomparable. The first method allows a more general type for `d` because could be applied to `readonly` or `mutable Date`s. The second method, on the other hand, allows a more general type for `l` because it can be applied to lists with `readonly`, `mutable`, or `? readonly` type arguments. Thus, neither the first or second method is the most general and, therefore, there does not exist a most general method signature to be inferred.

However, parameterizing the method would make the method more general than all three of the previous methods:

```
<T extends readonly Date> void addDate(/*mutable*/ List<T> l, T d) {
   l.add(d);
}
```

Thus, in the case that programmers follow good method parameterization practices, our algorithm is capable of always providing the most general methods.

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

```
class Bicycle {
  private Seat seat;

  Seat getSeat() {
    return seat;
  }
}

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

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

Figure 6-21: Type inference should declare the `getSeat` method to have an `romaybe` receiver and return type.

6.3 Inferring `romaybe`

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

As described in section 5.1, `romaybe` is used to create two versions of a method: a read-only version, which takes a read-only parameter and returns a read-only type, and a mutable version, which takes mutable parameter and returns a mutable type. For example, in figure 6-21, the `getSeat` method could be declared to have a `romaybe this` parameter and return type, as in figure 6-22. Then `lowerSeat`, which mutates the returned `Seat` object, can use the mutable version of `getSeat` and can take a mutable `Bicycle` parameter. However, `printSeat`, which does not mutate the returned `Seat` object, can use the read-only version of `getSeat` and can take a read-only `Bicycle` parameter.

Without the ability to infer `romaybe`, our algorithm would infer that `getSeat` has a mutable return type and a mutable formal parameter. In this case, methods such as `printSeat` will be forced to supply `getSeat` with a mutable argument even though `printSeat` does not mutate the object returned by `getSeat`. The results of this type-safe but imprecise analysis is shown in figure 6-23.

6.3.1 Approach

We extend our read-only inference algorithm to infer `romaybe` by recognizing that `romaybe` methods have two contexts. In the first context, the `romaybe` return type and `romaybe` parameters are mutable. In the second context, the `romaybe` return type and `romaybe` parameters are read-only. Since our analysis does not know which methods are `romaybe` before executing, it creates both contexts for every method. In the case of a method that should not have `romaybe` parameters, the mutabilities of parameters from each context will be identical. In the case of a method that should have `romaybe` parameters, the parameters will be mutable in the mutable context and read-only in the read-only context.

```
class Bicycle {
  private Seat seat;

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

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

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

Figure 6-22: Sound and precise Javari code which gives `getSeat` `romaybe` type. `lowerSeat` uses the mutable version of the method while `printSeat` uses the readonly version.

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

6.3.2 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 read-only and mutable versions of the methods. For example, `x = y` now generates the constraints:

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

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

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

The method invocation rule (shown in figure 6-24) must be modified to invoke the mutable version of a method when a mutable return type is needed and to invoke the read-only version otherwise. That is, for the method invocation, $\mathtt{x} = \mathtt{y}.\mathtt{m}(\overline{\mathtt{y}})$, the mutable version of the method is used when `x` is mutable. Otherwise, the read-only version of the method is used. The invocation rule checks whether `x` is mutable, and in the case that it is, adds constraints that the actual arguments must be mutable if the formal parameters of the mutable context of the method are mutable. The rule also, without checking the mutability of `x`, adds the constraints that the the actual arguments must be mutable if the formal parameters of the read-only context of the method are mutable. The check can be

```
class Bicycle {
  private Seat seat;

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

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

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

Figure 6-23: Sound but imprecise Javari code, which does not use `romaybe`. `getSeat` is given the imprecise mutable type instead of the precise `romaybe` type.

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

Figure 6-24: The core algorithm's INVK rule is replaced by INVK-ROMAYBE, which is used for method invocation in the presence of `romaybe` references.

skipped because it is guaranteed that the parameters of the mutable version of the method are mutable if the parameters of the read-only version of the method are mutable.

The constraints are solved as before.

6.3.3 Interpreting the results of the solved 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 read-only and mutable method contexts must be mapped to a single method-local Javari type: read-only, mutable, or `romaybe`.

A reference is declared mutable, if both the mutable and read-only context of the reference's constraint variable are found to be in the simplified, unguarded constraint set. A reference is declared read-only, if both mutable and read-only contexts of the reference's constraint variable are absent from the constraint set. Finally, a reference is declared `romaybe`, if the mutable context's constraint variable is in the constraint set but the read-only constraint variable is not in the constraint set, because the mutability of the reference depends on which version of the method is called.[7]

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

[7]The case that read-only constraint variable is not found in the constraint set but the mutable context's constraint variable is not cannot occur by the design of the INVK-ROMAYBE constraint generation rule.

```
class BankAccount {

   int balance;

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

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

}
```

Figure 6-25: A user has annotated `balance`'s receiver as read-only; however, `balance` mutates the field `securityLog`. Thus, `securityLog` should be declared mutable.

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

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

6.4 Inferring `mutable` and `assignable`

The core algorithm infers references to objects that are not modified as `readonly`. While it is able to leverage user-provided `assignable` and `mutable` annotations for inferring `readonly`, it is not able to infer the `assignable` and `mutable` annotations because that requires knowledge about the programmer's intent. Declaring a field `mutable` or `assignable` overrides Javari's default of transitive immutability and declares that the field is not in the object's abstract state. First, we present a technique that provides recommendations when the default transitive immutability guarantees should be overridden by leveraging references that the user marked as `readonly` but are used to modify the concrete state of their referents. Additionally, we present two such heuristic to detect fields that are used as caches and mark method references as `readonly`.

6.4.1 Leveraging `readonly` annotations

Our first technique for identifying fields that should be declared assignable or mutable relies on a programmer annotating some references read-only. In the case, that a user annotation conflicts with the results Javarifier—a user believes that a reference should be read-only but Javarifier find that the references is used to mutate its referent—it is likely that the mutation observed by Javarifier is only a mutation of the object's concrete state and not of object's abstract state. Therefore, the concrete state that is modified by the reference in question should be declared to be either assignable or mutable.

For example, consider the `balance` method shown in figure 6-25:

In this example, the read-only inference algorithm finds the receiver of `balance` ($\texttt{this}_{\texttt{balance}}$) to be mutable because it is used to mutate `this.securityLog`. However, since the user has specified that $\texttt{this}_{\texttt{balance}}$ should be read-only, there is a conflict between the user's annotation and Javarifier's results. This conflict is caused because the mutation of the receiver is only a mutation of the object's concrete state, not of the objects abstract state. To rectify this situation, `securityLog` needs to be excluded from `BankAccount`'s abstract state; thus, our technique recommends that `securityLog` is declared `mutable`.

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 always more favorable to modify a field of the class containing the reference in question than modifying a field of another class. For example, in the case of `BankAccount` (figure 6-25), making `List`'s `add` method read-only be declaring `List`'s internal fields mutable, would also make `balance` read-only. However, changing `List`'s internals is a non-local change, and, therefore, less desirable. Our technique, therefore, will list possible modifications in order of their locality.

This technique of using programmer provided read-only annotations can be applied in a three-step process. First, after running Javarifier a programmer examine the results to see if any references that he expected to be read-only were determined to be mutable. Second, after the programmer explicitly mark those references read-only, he/she can rerun Javarifier. At this point, Javarifier will provide fields that can be declared mutable or assignable to make the references in question read-only. The programmer can select the fields that should be excluded from abstract state, and then run Javarifier a final time yielding the correct results.

We can implement this technique by extending the constraint set to record the cause for select constraint variable being added to the set. In the case of field assignment and field reference, we record in the constraint variable of the enclosing class the field that is being manipulated. This field is referred to as the "target field". The target field of a constraint variable is shown after the constraint variable and separated by a $:_a$ in the case of a field assignment and a $:_r$ in the case field reference. For example, 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 being assigned.

Constraint variable target fields with the constraint variable during constraint solving, but does not affect constraint solving.

By examining the results of the solved constraint set[8] one can calculate which fields need to be modified by assignable or mutable. Before presenting formalization of our algorithm, we will motivate the algorithm's rules through a series of examples The example shown in figure reffig:example-assign.

In figure 6-27, the receiver of the method `foo`, $\texttt{this}_{\texttt{foo}}$, was explicitly marked read-only by the user; however, Javarifier determined that $\texttt{this}_{\texttt{foo}}$ should be declared mutable, as seen by the fact that the constraint variable $\texttt{this}_{\texttt{foo}} :_a \texttt{tire}$ is present in the constraint set. The cause of the constraint variable $\texttt{this}_{\texttt{foo}} :_a \texttt{tire}$ being in the constraint set is that `this.tire` was assigned to the method body of `foo`—for a `this-assignable` field of

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

$$\mathtt{x} = \mathtt{y} : \{\mathtt{x} \rightarrow \mathtt{y}\} \text{ (ASSIGN)}$$

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

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

$$\mathtt{x} = \mathtt{y.f} : \{\mathtt{x} \rightarrow \mathtt{f},\ \mathtt{x} \rightarrow \boxed{\mathtt{y} :_r \mathtt{f}}\} \text{ (REF)}$$

$$\mathtt{x.f} = \mathtt{y} : \{isnew\mathtt{x} :_a \mathtt{f},\ \mathtt{f} \rightarrow \mathtt{y}\} \text{ (SET)}$$

Figure 6-26: Constraint generation rules extended to record target fields. Modifications to the rules are indicated by boxes.

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

class Bicycle {
  Tire tire;

  void foo() |alert[readonly] {
    Tire t = new Tire(); |jcmt[none]
    this.tire = t;       |jcmt[\assign{\jv{this_foo}}{\jv{tire}}, \tire \guards \t]
   }
}
```

Solved constraint set:

$$\{\mathtt{this_{foo}} :_a \mathtt{tire}, \mathtt{tire} \rightarrow \mathtt{t}\}$$

Figure 6-27: `foo` has been annotated explicitly by the user to be read-only. Constraints generated for each line of code are shown after the line of code. $\mathtt{this_{foo}}$ can be read-only if `tire` is declared assignable.

`this` to be assigned to, `this` must be mutable. The cause can be seen by the constraint variable $\mathtt{this_{foo}}$ being tagged by the target field $:_a$ `tire`. The cause also suggests that $\mathtt{this_{foo}}$ could be read-only if `tire` was declared to be an assignable field. If `tire` was assignable, then it could be assigned to through a read-only reference. This observations can be generalized to following rule: if there is a conflict between a user annotation and Javarifier about reference `x`, then `x` can be made mutable by declaring all fields `f` assignable if the unguarded constraint $\mathtt{x} :_a \mathtt{f}$ appears in the solved constraint set.

In this example, $\mathtt{this_{bar}}$ was marked by the user to be read-only; however, Javarifier found the reference to be mutable as indicated by the unguarded constraint variable $\mathtt{this_{bar}} :_r \mathtt{tire}$ being present in the solved constraint set. The cause of this constraint

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

class Bicycle {
  Tire tire;

  void bar() readonly {
    Tire t = this.tire; // "jv–t" "guards "refen–"jv–this˙bar""–"jv–tire""
    t.radius = 16;      // "assign–"jv–t""–"jv–radius""
   }
}
```

Solved constraint set:

$$\{\texttt{t} :_a \texttt{radius}, \texttt{t}, \texttt{this}_{\texttt{bar}} :_r \texttt{tire}, \texttt{t} \rightarrow \texttt{this} :_r \texttt{tire}\}$$

Figure 6-28: `bar` has been annotated explicitly by the user to be read-only. $\texttt{this}_{\texttt{bar}}$ can be made read-only by declaring `tire` to be mutable.

variable being in the set is that `tire` is used in a mutable way, therefore, it must be accessed through a mutable reference to $\texttt{this}_{\texttt{bar}}$. This cause is recorded by the fact that the $\texttt{this}_{\texttt{bar}}$ constraint variable is tagged with $:_r$ `tire`. Furthermore, this cause suggests a way to make $\texttt{this}_{\texttt{bar}}$ read-only would to be to make `tire` a mutable field. In this case, $\texttt{this}_{\texttt{bar}}$ could be read-only but still provide a mutable reference to the `tire` field. These observations can be generalized to the following rule: for a reference, `x`, for which there is a conflict between user annotations and Javarifier results, the conflict can be removed by declaring all fields, `f`, mutable if $\texttt{x} :_a \texttt{f}$ appears in the solved constraint set.

The two rules suggested thus far must be applied together simultaneously. That is, if the code from `foo` and `bar` was combined into a single method with a user annotated read-only receiver , then `tire` need to be declared both assignable and mutable.

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

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

1. It checks if a user-annotated `readonly` reference is marked as `mutable`.
2. Goes backwards through constraint-set, from the reference to collect references which are modified causing the current reference to be mutable.
3. Maintain the order of collected fields — a breadth-first backwards traversal through

the constraint-set lists the fields from the most recent and local causes to indirect causes.
4. Recommend the collected fields to the user for annotations, either as `mutable` or as `assignable`. If a field is added as the cause in `objectMutatorFieldSet` then it needs to be annotated as `assignable`, otherwise the field needs to be annotated as `mutable`.

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

6.4.2 Heuristics and cache detection

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

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

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

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

6.5 Implementation

The Javarifier tool is a prototype implementation of the read-only and romaybe inference algorithms. To enable the annotation of library class-files in the absence of source code, Javarifier takes classfiles as input. Javarifier can either output its results to text files or apply the results directly to class files in the form of classfile attributes.

6.6 Evaluation

To evaluate Javarifier, we compared Javarifier's output against hand-annotating the Barnes-Hut (BH) benchmark from the JOlden benchmark suite [11]. The BH benchmark contains 1130 (non-comment, non-whitespace) lines of code among 7 classes. The BH benchmark is written using raw types, therefore, before we hand-annotated or ran Javarifier, we converted the BH source code to use generics.

	read-only	mutable	this-mutable	romaybe	`? readonly`	Total
field	2	0	19	2	0	23
receiver	18	30	N/A	6	0	54
parameter	32	13	N/A	0	0	45
return	4	13	N/A	12	0	29
local	4	48	N/A	2	0	54
Total	60	104	19	22	0	205

Figure 6-29: For each kind of reference in Java (fields, method receivers, parameters, and return values, and local variables), the table shows number of those references declared with a given Javari modifier. In the case of parametric types, each type argument is treated as a separate reference from the top-level type. Thus, `List<Date>` is counted as two references. Note that `romaybe` occurs on two fields. Although not discussed in this paper, `romaybe` can occur on fields of method-local classes.

To evaluate the results of hand-annotation and Javarifier, we produced a set of "ideal" annotations. These annotations where created by comparing the hand annotations against Javarifier's annotations. Wherever a difference occurred, we further examined the code to determine which annotation was correct. Although it is possible that the hand annotations and Javarifier erred in the same way, we have a high degree of confidence that the ideal annotations are correct. Figure 6-29 summarizes the ideal results.

Hand annotation of the BH benchmark took 30 minutes and contained 22 (11%) errors. Five of the errors were due to the tedium of the task: the programmer forgot to annotate obviously read-only methods such as `toString`. The remainder of the errors were due not investigating the program sufficiently.

Javarifier was able to annotate the BH benchmark in under three minutes (3.6GHz Pentium 4, 3GB of RAM). Javarifier erred in 35 (17%) of cases. The majority of cases (21) were due to use of Javarifier in closed-world mode: Javarifier inferred a field or method's return type to be read-only when it would be reasonable to mutate the reference. Those references were not mutated in the program, but code inspection shows that it would be reasonable for an unseen client to mutate the references.

The remaining errors were due to the prototype implementation being incomplete.

Chapter 7

Related Work

This paper is an extended version of a previous paper by same author [48]. The main changes from [48] are:

1. Introduction of the `? readonly` keyword for handling this-mutable type arguments (section 3.6.4).
2. An improved core language which handles the `? readonly` keyword (section 4).
3. A type inference to convert Java programs to Javari, including a discussion of a prototype implementation of the analysis (section 6).
4. A discussion of how Javari could be implemented using annotations (section B).

7.1 Javari2004

The Javari language presented in this paper and [48] is an evolutionary improvement of an earlier dialect [5], which we call "Javari2004".

Experience with 160,000 lines of Javari2004 code indicated that Javari2004 is an easy-to-use language that retains the flavor and style of Java while providing substantial benefits, including improved documentation, extended ability to reason about code, and detecting errors in well-tested code. However, the Javari2004 design is deficient in a number of ways.

1. Conflates notions of assignability and mutability
2. Incompatible with generic types
3. Inflexible multi-dimensional arrays
4. Extra-linguistic macro-expansion templates
5. No support for reflection
6. No formal type rules

The current Javari language corrects these problems. The changes are significant but are relatively small from the point of view of a user: most uses of the language, and its overall character, remain the same.

Distinguishing assignability from mutability

Javari2004's `mutable` keyword declares that a field is both assignable and mutable: there is no way to declare that a field is only assignable or only mutable. Javari's `assignable` and

`mutable` keywords (section 3.5.3) highlight the orthogonality of assignability and mutability, and increase the expressiveness of the language. See appendix A for examples of the use of `assignable` and `mutable`.

Generic types

This paper provides a detailed treatment of generic classes that smoothly integrates reference immutability into them. Javari2004 does not support generic classes, though the OOPSLA 2004 paper speculates about a macro expansion mechanism that is syntactically, but not semantically, similar to the way that Java 5 treats type parameters. Java 5 compiles type parameters via type erasure, but Javari2004 treated the mutability parameters (which appeared in the same list as the type parameters) via code duplication; this distinction complicates implementation, understanding, and use.

Javari2004 also proposed that a generic class could declare whether a field whose type is a type parameter is a part of the object's abstract state. We have discovered that such a declaration makes no sense. For a field whose type is a type parameter to be a part of the object's abstract state, it would have to be this-mutable; however, such a field cannot be this-mutable (section 3.2). Javari also disallows type parameters to be modified with the `mutable` keyword (section 3.6).

Arrays

As with generic classes, Javari permits programmers to independently specify the mutability of each level of an array (section 3.7). By contrast, Javari2004's specification states: "`readonly int[][]` and `readonly (readonly int[])` are equivalent," forbidding creation of a read-only array of mutable items.

Method templates

Javari2004 integrated the syntax for templating a method over mutability with the syntax for Java 5's generic types. Whether a parameter is intended to be a normal type parameter or a mutability type parameter must be inferred from its usage, greatly complicating a compiler (and the prototype Javari2004 implementation required distinct syntax to ease the compiler's task [4, 5]).

Furthermore, Javari2004 allows declaring multiple mutability type parameters. As noted in section 5.1.2, a single mutability type parameter is generally sufficient, so Javari uses a much simpler mechanism (`romaybe`) for indicating a variable mutability. This new approach highlights the orthogonality of the Java 5's generic types and Javari's mutability polymorphism for methods. Furthermore, it does not require any run-time representation of the polymorphism.

Reflection

Reflection creates objects whose types are not known to the static type checker. In Javari2004 (as in many other proposed type systems for Java), reflection creates loopholes in the type system. The current Javari language is sound with respect to reflection by introducing quick, local checks that can be performed at run time. See section 5.2.

Formal type rules

Javari2004's type-checking rules [5, 4] are stated, for the full Javari2004 language (including inner classes and other Java idiosyncrasies) in the semi-formal style of the Java Language Specification [22]. This formulation is natural for many Java programmers, but it is unsatisfying to formalists. This paper formalizes a core calculus for the Javari language. It builds on Wild FJ (WFJ) [23], adding side effects and assignability and mutability type modifiers. By presenting the typing rules for a language that is stripped to the bare essentials, we have made it easier to grasp the key features of Javari. Equally importantly, the formalization can enable a type soundness proof for the Javari type system.

7.2 Other immutability proposals

Many other researchers have noticed the need for a mechanism for specifying and checking immutability. This section discusses other proposals and how ours differs from them.

Similarly to Javari, JAC [24] has a `readonly` keyword indicating transitive immutability, an implicit type `readonly T` for every class and interface `T` defined in the program, and a `mutable` keyword. However, the other aspects of the two languages' syntax and semantics are quite different. For example, JAC provides a number of additional features, such as a larger access right hierarchy (`readnothing` < `readimmutable` < `readonly` < `writeable`) and additional keywords (such as `nontransferrable`) that address other concerns than immutability. The JAC authors propose implementing JAC by source rewriting, creating a new type `readonly T` that has as methods all methods of `T` that are declared with the keyword `readonly` following the parameter list (and then compiling the result with an ordinary Java compiler). However, the return type of any such method is `readonly`. For example, if class `Person` has a method `public Address getAddress() readonly`, then `readonly Person` has method `public readonly Address getAddress() readonly`. In other words, the return type of a method call depends on the type of the receiver expression and may be a supertype of the declared type, which violates Java's typing rules. Additionally, JAC is either unsound for, or does not address, arrays of `readonly` objects, casts, exceptions, inner classes, and subtyping. JAC `readonly` methods may not change any static field of any class. The JAC paper suggests that `readonly` types can be supplied as type variables for generic classes without change to the GJ proposal, but provides no details. By contrast to JAC, in Javari the return type of a method does not depend on whether it is called through a read-only reference or a non-read-only one. Javari obeys the Java type rules, uses a type checker rather than a preprocessor, and integrates immutability with type parameterization. Additionally, we have implemented Javari and evaluated its usability [5].

The above comments also explain why use of read-only interfaces in Java is not satisfactory for enforcing reference immutability. A programmer could define, for every class `C`, an interface `RO_C` that declares the readonly methods and that achieves transitivity through changing methods that returned (say) `B` to return `RO_B`. Use of `RO_C` could then replace uses of Javari's `readonly C`. This is similar to JAC's approach and shares similar problems. For instance, to permit casting, `C` would need to implement `RO_C`, but some method return and argument types are incompatible. Furthermore, this approach does not allow readonly versions of arrays or even `Object`, since `RO_Object` would need to be implemented by `Object`. It also forces information about a class to be maintained in two separate files, and it does not address run-time checking of potentially unsafe operations or how to handle various other Java constructs. Javari sidesteps these fundamental problems by extending the Java

type system rather than attempting to work within it.

Skoglund and Wrigstad [43] take a different attitude toward immutability than other work: "In our point of [view], a read-only method should only protect its enclosing object's transitive state when invoked on a read reference but not necessarily when invoked on a write reference." A `read` (read-only) method may behave as a `write` (non-read-only) method when invoked via a `write` reference; a `caseModeOf` construct permits run-time checking of reference writeability, and arbitrary code may appear on the two branches. This suggests that while it can be proved that read references are never modified, it is not possible to prove whether a method may modify its argument. In addition to read and write references, the system provides `context` and `any` references that behave differently depending on whether a method is invoked on a read or write context. Compared to this work and JAC, Javari's type parameterization gives a less ad hoc and more disciplined way to specify families of declarations.

The functional methods of Universes [32] are pure methods that are not allowed to modify anything (as opposed to merely not being allowed to modify the receiver object).

Pechtchanski and Sarkar [36] provide a framework for immutability specification along three dimensions: lifetime, reachability, and context. The lifetime is always the full scope of a reference, which is either the complete dynamic lifetime of an object or, for parameter annotations, the duration of a method call. The reachability is either shallow or deep. The context is whether immutability applies in just one method or in all methods. The authors provide 5 instantiations of the framework, and they show that immutability constraints enable optimizations that can speed up some benchmarks by 5–10%. Javari permits both of the lifetimes and supplies deep reachability, which complements the shallow reachability provided by Java's `final` keyword.

Capabilities for sharing [10] are intended to generalize various other proposals for immutability and uniqueness. When a new object is allocated, the initial pointer has 7 access rights: read, write, identity (permitting address comparisons), exclusive read, exclusive write, exclusive identity, and ownership (giving the capability to assert rights). Each (pointer) variable has some subset of the rights. These capabilities give an approximation and simplification of many other annotation-based approaches.

Porat et al. [37] provide a type inference that determines (deep) immutability of fields and classes. (Foster et al. [19] provide a type inference for C's (non-transitive) `const`.) A field is defined to be immutable if its value never changes after initialization and the object it refers to, if any, is immutable. An object is defined to be immutable if all of its fields are immutable. A class is immutable if all its instances are. The analysis is context-insensitive in that if a type is mutable, then all the objects that contain elements of that type are mutable. Libraries are neither annotated nor analyzed: every virtual method invocation (even `equals`) is assumed to be able to modify any field. The paper discusses only class (static) variables, not member variables. The technique does not apply to method parameters or local variables, and it focuses on object rather than reference immutability, as in Javari. An experiment indicted that 60% of static fields in the Java 2 JDK runtime library are immutable. This is the only other implemented tool for immutability in Java besides ours, but the tool is not publicly available for comparison.

Effect systems [26, 46, 34] specify what state (in terms of regions or of individual variables) can be read and modified by a procedure; they can be viewed as labeling (procedure) types with additional information, which the type rules then manipulate. Type systems for immutability can be viewed as a form of effect system. Our system is finer-grained than typical effect systems, operates over references rather than values, and considers all state

reachable from a reference.

Our focus in this paper is on imperative object-oriented languages. In such languages, fields are mutable by default. In our type system, when a type is read-only, the default is for each field to be immutable unless the user explicitly marks it as mutable. Functional languages such as ML [29] use a different policy: they default all fields to being immutable. OCaml [25] combines object-orientation with a `mutable` annotation on fields (for example, references are implemented as a one-field mutable record). However, without a notion of read-only types, users are forced to hide mutability via use of interfaces and subtyping, which is less flexible and expressive than our proposal.

A programming language automatically provides a sort of immutability constraint for parameters that are passed, or results that are returned, by value. Since the value is copied at the procedure call or return, the original copy cannot be modified by the implementation or client, respectively. Pass- and return-by-value is typically used for values that are small. Some programming languages, such as Pascal and Ada, permit variables to be explicitly annotated as in, out, or in/out parameters; this is an early and primitive form of compiler-enforced immutability annotation.

7.3 C++ `const`

C++'s `const` keyword is intended to aid in interfaces, not symbolic constants [45]. Our motivation is similar, but our notion of immutability, and our type system, differ from those of C++, thus avoiding the pitfalls that led Java's designers to omit `const`.

Because of numerous loopholes, the `const` notation in C++ does not provide a guarantee of immutability even for accesses through the `const` reference. An unchecked cast can remove `const` from a variable, as can (mis)use of type system weaknesses such as unions and varargs (unchecked variable-length procedure arguments).

C++ permits the contents of a read-only pointer to be modified: read-only methods protect only the local state of the enclosing object. To guarantee transitive non-mutability, an object must be held directly in a variable rather than in a pointer. However, this precludes sharing, which is a serious disadvantage. Additionally, whereas C++ permits specification of `const` at each level of pointer dereference, it does not permit doing so at each level of a multi-dimensional array. Finally, C++ does not permit parameterization of code based on the immutability of a variable.

By contrast to C++, Javari is safe: its type system contains no loopholes, and its downcast is dynamically checked. Furthermore, it differs in providing guarantees of transitive immutability, and in not distinguishing references from objects themselves; these differences make Javari's type system more uniform and usable. Unlike C++, Javari permits mutability of any level of an array to be specified, and permits parameterization based on mutability of a variable. Javari also supports Java features that do not appear in C++, such as nested classes.

Most C++ experts advocate the use of `const` (for example, Meyers advises using `const` wherever possible [27]). However, as with many other type systems (including those of C++ and Java), some programmers feel that the need to specify types outweighs the benefits of type checking. At least three studies have found that static type checking reduces development time or errors [31, 21, 38]. We are not aware of any empirical (or other) evaluations regarding the costs and benefits of immutability annotations. Java programmers seem eager for compiler-checked immutability constraints: as of March

2005, support for `const` is the second most popular Java request for enhancement. (See `http://bugs.sun.com/bugdatabase/top25_rfes.do`. The most popular request is "Provide documentation in Chinese.")

A common criticism of `const` is that transforming a large existing codebase to achieve `const` correctness is difficult, because `const` pervades the code: typically, all (or none) of a codebase must be annotated. This propagation effect is unavoidable when types or externally visible representations are changed. Inference of `const` annotations (such as that implemented by Foster et al. [19]) eliminates such manual effort. Even without a type inference, we found the work of annotation to be greatly eased by fully annotating each part of the code in turn while thinking about its contract or specification, rather than inserting partial annotations and attempting to address type checker errors one at a time. The proper solution, of course, is to write `const` annotations in the code from the beginning, which takes little or no extra work.

Another criticism of C++'s `const` is that it can occasionally lead to code duplication, such as the two versions of `strchr` in the C++ standard library. Mutability templates (section 5.1) make the need for such duplication rare in Javari. Finally, the use of type casts (section 5.3.5) permits a programmer to soundly work around problems with annotating a large codebase or with code duplication.

7.4 Related analyses

Reference immutability can help to prevent an important class of problems, in a simple and intuitive way. However, it is no panacea. Other techniques can address some of these issues, and there are many software engineering challenges that reference immutability does not address. We mention just a sample of other techniques.

Boyland [9] observes that mutational representation exposure (in which external code can corrupt a data structure) and observational exposure (in which external code can observe an internal representation changing) are duals: in each case, modifications on one side of an abstraction boundary are observable on the other. Reference immutability does not address observational exposure. Boyland argues that a language extension should not solve one of these problems without also solving the other. However, the problems are arguably different, since the latter is a result of a client improperly retaining a reference "too long," and even a value returned from `size()` may become out of date if it is retained too long (though it will never become an invalid integer). Mechanisms for solving all representation exposure problems are less mature, and it may be valuable to solve some important problems without solving them all.

Ownership types [12, 3, 7] provide a compiler-checked mechanism for preventing aliasing to the internal state of an object. As noted previously, alias, escape, and ownership analyses can enhance reference immutability. However, they do not directly address issues of immutability, including those not associated with abstraction boundaries. Ownership type annotations such as `rep` describe whether a reference is part of the object's state, whereas mutability annotations such as `readonly` indicate whether it can be modified; each approach has its advantages, and it would be interesting to combine them. Fractional permissions [8] are another mechanism for helping to avoid representation exposure. Finally, a type system based on linear logic [49, 17] can prevent multiple uses of a value, which may be useful, for example, when preventing representation exposure through constructor arguments.

Effect analyses for Java and in general [13, 41, 39, 40, 33, 28] have been widely studied.

```
/** Copies of the day of d1 into d2. */
void copyDay(readonly /*unsafe*/ Date d1, /*mutable, unsafe*/ Date d2) {
  d2.setDay(d1.getDay());
}

Date d = new Date();
copyDate(d, d);
```

Figure 7-1: In the presence of the method invocation shown, `copyDay`'s `d1` parameter is read-only but not safe. Effect analysis determines `d1` to be unsafe because of the method invocation shown passes the same `Date` object to both parameters; thus, the object referred to by `d1` is mutated through the alias `d2`. The read-only-ness of method parameters, as expected for type system, can not be affected by the particular manner in which the method is invoked.

```
/** Returns a copy of d with the month set to m */
Date copy(/*mutable*/ /*safe*/ Date d, readonly /*safe*/ Month m) {
  d = new Date(d); // Assume copy-constructor's parameter is read-only and safe.
  d.setMonth(m);
  return d;
}
```

Figure 7-2: `d` is safe because any object passed to `d` will not be modified; however, `d` is not read-only because it is used to modify a different object—the newly created `Date` returned by the method.

An effect analysis returns the set of references that initially refer to objects that are not mutated in a given scope such as a particular method invocation. Similar to our algorithm's read-only parameters, effect analysis can be used to determine which method parameters are "safe"[42]. A method parameter is safe if the method never modifies the object passed to the parameter during method invocation. Unlike read-only references, a parameter is not safe if the method mutates the object to which the parameter refers through a different aliasing reference. An example of this difference is shown in figure 7-1. On the other hand, if a safe parameter is reassigned with a different object, the parameter remains safe even if it mutates that object. A read-only parameter, on the other hand, may not be used to mutate any object. This difference is shown in figure 7-2 Safety and read-only-ness are orthogonal: safety is a property over objects while read-only-ness is a property over references.

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

Chapter 8

Conclusion

We have presented a type system that is capable of expression, compile-time verification, and run-time checking of reference immutability constraints. Reference immutability guarantees that the reference cannot be used to perform any modification of a (transitively) referred-to object. The type system is generally applicable to object-oriented languages, but for concreteness we have presented it in the context of Javari, an extension to the full Java 5 language, including generic types, arrays, reflection, serialization, inner classes, exceptions, and other idiosyncrasies. Immutability polymorphism (templates) for methods are smoothly integrated into the language, reducing code duplication. We have provided a set of formal type rules for a core calculus that models the Javari language.

Javari provides a practical and effective combination of language features. For instance, we describe a type system for reference rather than object immutability. Reference immutability is useful in more circumstances, such as specifying interfaces, or objects that are only sometimes immutable. Furthermore, type-based analyses can run after type checking in order to make stronger guarantees (such as object immutability) or to enable verification or transformation. The system is statically type-safe, but optionally permits downcasts that transform compile-time checks into run-time checks for specific references, in the event that a programmer finds the type system too constraining. The language is backward compatible with Java and the Java Virtual Machine, and is interoperable with Java. Together with substantial experience with a prototype for a closely related dialect [5], these design features provide evidence that the language design is effective and useful.

Additionally we have designed and implemented an analysis to convert Java programs to Javari. This analysis is necessary for converting legacy programs and libraries such as the JDK to Javari. Experience with the analysis provides further evidence that Javari is practical.

Appendix A

Assignability and Mutability Examples

As shown in figure 3-6, each instance field can be declared with one of three assignabilities (assignable, unassignable, or this-assignable) and also with one of three mutabilities (mutable, read-only, and this-mutable). This appendix illustrates the use of Javari's reference immutability system through examples of all nine possibilities. As with the rest of this paper, we omit most visibility modifiers (`public`, `private`, ...) for brevity. Furthermore, for brevity this appendix does not explicitly address uses of type parameters.

A.1 this-assignable, this-mutable

This is the standard type for a field in a possibly-mutable class. All fields not declared `final` in Java code are interpreted as this type.

Suppose there is a class `Wheel` that is mutable (its pressure can change) and a class `Bicycle` that contains two `Wheel`s that may be changed (different wheels for different terrains).

```
class Wheel {
  int pressure;
}

class Bicycle {
  Wheel frontWheel;
  ...
}
```

A read-only reference to a `Bicycle` cannot be used to modify the `Bicycle` by reassigning `frontWheel` or mutating `frontWheel` by reassigning `pressure`. A mutable reference can modify `Bicycle` by reassigning `frontWheel` or changing its pressure.

A.2 `final`, this-mutable

Consider a file abstraction that contains a `StringBuffer` holding the contents of the file.

```
class MyFile {
  final StringBuffer contents;
}
```

The contents of a read-only file cannot be changed. `contents` should not be reassigned, and `StringBuffer` operations can be used to alter mutable files' contents as needed.

A.3 `assignable`, this-mutable

`assignable` fields can be used for caching. In the case of an `assignable` this-mutable field, it is caching an object that is this-mutable. For example, recall the `Bicycle` example from above. Suppose one wished to provide a method that returned the wheel with the greatest aerodynamic drag. If this method was costly to execute, one would wish to cache the result until one of the `Wheels` changed.

```
class Bicycle {
  Wheel frontWheel;
  Wheel backWheel;

  private assignable Wheel mostDrag;
  romaybe Wheel mostDrag() romaybe {
    if (mostDrag == null || /*wheels have changed*/) {
      mostDrag = ...;
    }
    return mostDrag;
  }
}
```

Even when the reference to the `Bicycle` is read-only, `mostDrag` can be assigned with the result of the method. The cache and what it is caching — one of the `Wheels` — are this-mutable.

A.4 this-assignable, `readonly`

Consider a class, `ChessPiece`, that represents a chess piece including the piece's position on a board. The position field, `pos`, should be modifiable for mutable references but not for read-only references. If the programmer wishes to reassign `pos`, instead of mutating the object assigned to the field, each time the piece is moved, then the field should be declared to be this-assignable and read-only.

```
class ChessPiece {
  readonly Position pos;

  readonly Position getPosition() readonly {
    return pos;
  }

  void setPosition(readonly Position pos) {
    this.pos = pos;
  }
}
```

A.5 `final readonly` fields

`final readonly` fields are useful for state that should never change, including constants. A `ChessPiece`'s color should never change.

```
class ChessPiece {
  final readonly Color color;
}
```

A.6 `assignable readonly`

Suppose that `ChessPiece` has a costly method, `bestMove`, that returns the best possible move for that piece. Since the method is costly, one would like the method to cache its result in case it is called again (before anything on the board has altered). Calling the `bestMove` method does not change the abstract state of the class, so it should be a `readonly` method. However, to allow the `bestMove` method to assign to the field that caches its result, the field must be declared `assignable`. Furthermore, since there is never a reason to mutate the position calculated, the field should be declared `readonly` to avoid programmer error.

```
class ChessPiece {
  private assignable readonly Position bestMove;
  readonly Position bestMove() readonly {
    if ((bestMove == null) || /* board changed */) {
      bestMove = ...;
    }
    return bestMove;
  }
}
```

A.7 this-assignable, `mutable`

Suppose one wishes to represent a set by an array and, for efficiency, move the last successfully queried item to the beginning of the array. The author must declare the array to be `mutable` to allow moving the last successfully queried item to the beginning of the array even when the set is read-only. The array must be declared this-assignable to allow reassigning the array when it reaches its capacity due to calls to `addElm`.

```
class MoveToFrontSet {
  private mutable Object[] elms;
  private int size;
  boolean contains(Object obj) readonly {
    for (int i = 0; i < size; i++) {
      if (elms[i].equals(obj)) {
        // should also check for null
        moveToFront(elms, i);
        return true;
      }
    }
    return false;
  }
```

```
  // Be sure to not declare method readonly
  void add(Object elm) {
    if (elms.length == size) {
      Object[] tmp = new Object[2*elms.length];
      for (int i = 0; i < elms.length; i++) {
        tmp[i] = elms[i];
      }
      elms = tmp;
    }
    elms[size] = elm;
    size++;
  }

  // Be sure to return readonly Object[]
  readonly Object[] toArray() readonly {
    return elms;
  }
}
```

The order that the elements appear in `elms` is not a part of the abstract state of the `Set` object; however, the fact that they are contained by the array assigned to `elms` is a part of the abstract state. This relationship is too complicated for the type system to capture, so the field must be declared `mutable`.

`elms` must be `mutable` so that the elements can be rearranged even when the instance of `Set` is read-only. Unfortunately, methods that add or delete elements could be declared `readonly` and still type check. Therefore, when writing code such as this, the programmer must be careful not to declare those methods read-only and to ensure that a mutable reference to `elms` does not escape.

A.8 `final`, `mutable`

Suppose one wishes to monitor the users who access a file. A simple way to do this is to require a user ID to be passed as an argument to the file's accessors and then record the ID in a set.

```
class File {
  final mutable Set<UserID> accessedFile;
  StringBuffer contents;

  readonly StringBuffer
  getContents(UserID id) readonly {
    accessed.add(id);
    return contents;
  }
}
```

The set `accessedFile` must be mutable so that a users ID may be added to it within the read-only method `getContents`.

A.9 `assignable`, `mutable`

Consider an implementation of a splay tree [44]:

```
class SplayTree<T extends readonly Comparable> {

  // The internal representation of a splay tree
  assignable mutable BinarySearchTreeNode<T> root;

  // Adjusts the tree so that newRoot becomes the root.
  splay(BinarySearchTreeNode newRoot) { ... }

  void insert(T val) { root.insert(val); }
  void delete(T val) { root.delete(val); }

  boolean find(T val) readonly {
    BinarySearchTreeNode node = root.find(val);
    if (node == null) {
      return false;
    } else {
      splay(node);
      return true;
    }
  }
}
```

In this example, `assignable` and `mutable` are used because the type system is unable to capture how the abstract state of the class relates to its data structure at the field `root`. Without the `assignable` keyword, the root of the tree could not be reassigned by the `splay` method. The `mutable` keyword is also needed because the `splay` method needs to mutate the nodes rooted at `root` while rearranging the nodes within the tree.

Appendix B

Annotations

It is impractical to implement reference immutability using Java's current annotation system. Alternatively, Javari can be implemented using new keywords (as assumed in the description of the language up to this point) or through a extended annotation system. This section discusses the issues involved. We believe that Javari should either use all keywords or all annotations; it would be confusing, and would offer little benefit, to mix the two.

Use of annotations is attractive. It ensures that Javari code is valid Java code. This guarantees interoperability with existing tools: Javari code can be compiled using any Java compiler, then run either by a JVM whose byte code verifier checks the immutability types encoded in the annotations, or by any JVM (losing the benefits of immutability checking). The Javari system could additionally work as a stand-alone type-checker. There would be no worries about breaking existing code that uses Javari keywords as identifiers. Avoiding changes to the programming language could encourage programmers to adopt Javari.

Unfortunately, the current Java annotation system is too weak to use to implement a reference immutability system. There are two main problems.

1. Annotations can only be applied to type declarations, not to other uses of types.
   (a) Annotations cannot be applied to a cast.
   (b) Annotations cannot be applied to the receiver (`this`) of a method. This could be worked around with a new annotation such as `@rothis`, which is syntactically applied to the method return type but semantically applies to the receiver type.
   (c) Annotations cannot be inserted at arbitrary locations within arrays. To express "`(readonly Date[])[][]`", one would need to write something like the (unintuitive) annotation `@readonly(2) Date[][][]` where the integer argument to the annotation indicates at what level the read-only modifier should be applied.
   (d) Annotations are not permitted on type parameters, so expressing this type would be difficult:
   `Map<List<readonly Date>, readonly Set<Number>>`.
2. Annotations on local variables are not recorded within the classfile by the `javac` compiler. Therefore, if we wish to use annotations and perform type checking on classfiles, we would be required to extend the annotation system. This would require changing the compiler, possibly by recording local variables' annotations within the local variable symbol table. A compiler change eliminates one of the benefits of using annotations: not requiring people to use a new compiler to check reference immutability

constraints. (But the Javari code would remain backward-compatible with other Java compilers.)

Currently, there is work on extending Java's annotation system to overcome the limitations discussed above [14]. If such an extended annotation system was adopted, it would be practical to implement Javari in terms of annotations.

A language change could still achieve backward compatibility with standard Java compilers by providing a special comment syntax where any comment that begins with "`/*=`" is considered as part of the code by the Javari compiler. (This approach was taken in LCLint [16], for example.) This feature allows the programmer to annotate an existing Java program with Javari's keywords without losing the ability to compile that program with a normal Java compiler. That comment mechanism could be supported by external tools, but should not be part of Javari proper.

Bibliography

[1] Alfred V. Aho, Ravi Sethi, and Jeffrey D. Ullman. *Compilers: Principles, Techniques, and Tools*. Addison-Wesley, Reading, Massachusetts, 1986.

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

[3] Jonathan Aldrich, Valentin Kostadinov, and Craig Chambers. Alias annotations for program understanding. In *OOPSLA*, pages 311–330, October 2002.

[4] Adrian Birka. Compiler-enforced immutability for the Java language. Technical Report MIT-LCS-TR-908, MIT Lab for Computer Science, June 2003. Revision of Master's thesis.

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

[6] Joshua Bloch. *Effective Java Programming Language Guide*. Addison Wesley, Boston, MA, 2001.

[7] Chandrasekhar Boyapati, Barbara Liskov, and Liuba Shrira. Ownership types for object encapsulation. In *POPL*, pages 213–223, January 2003.

[8] John Boyland. Checking interference with fractional permissions. In *SAS*, pages 55–72, June 2003.

[9] John Boyland. Why we should not add `readonly` to Java (yet). In *FTfJP*, July 2005.

[10] John Boyland, James Noble, and William Retert. Capabilities for sharing: A generalisation of uniqueness and read-only. In *ECOOP*, pages 2–27, June 2001.

[11] Brendon Cahoon and Kathryn S. McKinley. Data flow analysis for software prefetching linked data structures in Java. In *PACT*, pages 280–291, September 2001.

[12] David G. Clarke, James Noble, and John M. Potter. Simple ownership types for object containment. In *ECOOP*, pages 53–76, June 2001.

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

[14] Michael D. Ernst and Danny Coward. JSR 308: Annotations on Java types. `http://pag.csail.mit.edu/jsr308/`, October 17, 2006.

[15] David Evans. Static detection of dynamic memory errors. In *PLDI*, pages 44–53, May 1996.

[16] David Evans, John Guttag, James Horning, and Yang Meng Tan. LCLint: A tool for using specifications to check code. In *FSE*, pages 87–97, December 1994.

[17] Manuel Fähndrich and Robert DeLine. Adoption and focus: Practical linear types for imperative programming. In *PLDI*, pages 13–24, June 2002.

[18] Matthew Flatt, Shriram Krishnamurthi, and Matthias Felleisen. Classes and mixins. In *POPL*, pages 171–183, January 1998.

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

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

[21] John D. Gannon. An experimental evaluation of data type conventions. *CACM*, 20(8):584–595, August 1977.

[22] James Gosling, Bill Joy, Guy Steele, and Gilad Bracha. *The Java Language Specification*. Addison Wesley, Boston, MA, second edition, 2000.

[23] Atsushi Igarashi, Benjamin C. Pierce, and Philip Wadler. Featherweight Java: a minimal core calculus for Java and GJ. *ACM TOPLAS*, 23(3):396–450, May 2001.

[24] Günter Kniesel and Dirk Theisen. JAC — access right based encapsulation for Java. *Software: Practice and Experience*, 31(6):555–576, 2001.

[25] Xavier Leroy. *The Objective Caml system, release 3.07*, September 29, 2003. with Damien Doligez, Jacques Garrigue, Didier Rémy and Jérôme Vouillon.

[26] John M. Lucassen and David K. Gifford. Polymorphic effect systems. In *POPL*, pages 47–57, January 1988.

[27] Scott Meyers. *Effective C++*. Addison-Wesley, second edition, 1997.

[28] 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.

[29] Robin Milner, Mads Tofte, and Robert Harper. *The Definition of Standard ML*. MIT Press, 1990.

[30] Robin Milner, Mads Tofte, Robert Harper, and Dave MacQueen. *The Definition of Standard ML (Revised)*. MIT Press, 1997.

[31] James H. Morris. Sniggering type checker experiment. Experiment at Xerox PARC, 1978. Personal communication, May 2004.

[32] P. Müller and A. Poetzsch-Heffter. Universes: A type system for alias and dependency control. Technical Report 279, Fernuniversität Hagen, 2001.

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

[34] F. Nielson and H. R. Nielson. Type and effect systems. In *Correct System Design*, number 1710 in LNCS, pages 114–136. Springer-Verlag, 1999.

[35] Jens Palsberg. Type-based analysis and applications. In *PASTE*, pages 20–27, June 2001.

[36] Igor Pechtchanski and Vivek Sarkar. Immutability specification and its applications. In *Java Grande*, pages 202–211, November 2002.

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

[38] Lutz Prechelt and Walter F. Tichy. A controlled experiment to assess the benefits of procedure argument type checking. *IEEE TSE*, 24(4):302–312, April 1998.

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

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

[41] 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, March 2001.

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

[43] Mats Skoglund and Tobias Wrigstad. A mode system for read-only references in Java. In *FTfJP*, June 2001.

[44] Daniel Dominic Sleator and Robert Endre Tarjan. Self-adjusting binary search trees. *JACM*, 32(3):652–686, July 1985.

[45] Bjarne Stroustrup. *The C++ Programming Language*. Addison-Wesley, Boston, MA, special edition, 2000.

[46] Jean-Pierre Talpin and Pierre Jouvelot. The type and effect discipline. In *LICS*, pages 162–173, June 1992.

[47] Mads Torgersen, Erik Ernst, and Christian Plesner Hansen. Wild FJ. In *FOOL*, Long Beach, CA, USA, January 2005.

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

[49] Philip Wadler. Linear types can change the world! In *IFIP TC 2 Working Conference on Programming Concepts and Methods*, pages 347–359, Sea of Galilee, Israel, April 1990.