Type Qualifiers for Java

David Greenfieldboyce Jeffrey S. Foster
University of Maryland, College Park
{dgreenfi,jfoster}@cs.umd.edu

August 8, 2005

Abstract

We describe Jqual, a practical framework for type qualifier inference in Java. Jqual is implemented as an Eclipse plug-in that allows researchers to quickly develop and test modifications to the Java type system based on the flexible, lightweight and familiar mechanism of type qualifiers. We demonstrate one such analysis which supports specification and checking of reference immutability for Java types modeled on the Javari language[2]. Finally, we describe the results of applying this analysis to a moderate-sized sample program.

1 Introduction

As the Java programming language gains popularity as a platform for large, widely-distributed applications, it becomes increasingly important for researchers to develop techniques for assuring security and correctness in Java programs. We believe that this goal can be advanced by the availability of a highly-adaptable framework for program analysis that will allow researches to quickly develop and test analysis techniques on large programs. Among the many approaches that have been investigated for program analysis, type-based analysis has proven to be powerful, efficient and broadly-applicable. Therefore, we seek to develop a method for researches to quickly define, implement, and test modifications to the Java type system with a minimum of repetition of effort.

Type qualifiers provide just such a means for modifying an existing type system in a convenient and easily understood way. Type qualifiers are modifiers that can be applied to the existing types in a language to provide more restrictive types that embody extra properties or specifications about terms in a program. Many languages already include a fixed set of type qualifiers as part of the language for just this purpose—`const` and `volatile` in C, for example, or `final` in Java. By presenting a general-purpose mechanism for introducing new type qualifiers into Java, we leverage the flexibility of type qualifiers as a way to add new properties to the language without the need to broadly reinvent the type system.

As with standard type qualifiers, our general purpose type qualifiers are introduced into a program as decorators on existing types. For example, as part of an analysis we implemented to provide reference immutability specifications into Java, based on the Javari variant of Java [2], we define a qualifier `readonly`, which indicates that an object reference cannot be used to modify the target object to which it refers. A programmer could define a variable `x` as a `readonly` reference of type `C` as:

```
readonly C x;
```

See Section 3 for a full discussion of our implementation of reference immutability.

Type qualifiers can be applied to any part of any type within the program. For example, it might be useful to specify that the return value of a method is `readonly` to guarantee that the caller of the method cannot modify the returned object:

```
class C {
  private List data;
  readonly List getData() { return data; }
}
```

In order to allow rich semantics for type qualifiers we allow subtyping relationships between different qualifiers. In our implementation of reference immutability we define two different qualifiers, `readonly` and `mutable`, where `mutable` means that a type is not `readonly`. We define a relationship `mutable` $\leq$ `readonly` meaning `mutable` references are a subtype of `readonly` references and so can be used in place of `readonly` references. Intuitively, a `mutable` reference can do everything a `readonly` reference can, plus it can be used to modify the object to which it refers. It has a superset of the functionality of a `readonly` reference and is therefore a subtype of `readonly`.

To simplify the introduction of type qualifiers into Java programs, we describe Jqual, a framework for performing type inference on type qualifiers. In the Jqual system, programmers can specify properties by annotating key terms with qualifiers. We then perform type inference determine the qualifiers on the remaining terms in the program. For example, in the following code the parameter `p` to the function `f` is annotated as `readonly`:

```
void f(readonly List p){
  List x = p;
  x.clear();
}
```

If the variable `x` is not annotated as `readonly`, the semantics of `p` as a `readonly` parameter are not preserved—after the reference is assigned from `p` to `x` it can be used to modify the underlying object. In our type system, the qualifier on `x` will be inferred to be a supertype of the qualifier on `p` at the point where `p` is assigned to `x`, which will constrain it to be `readonly`. When `x` is subsequently used to modify the input value, this will be flagged as an error.

Previous work on type qualifier inference has been done for the C language [4]. The primary technical challenge in extending this work to Java is to provide handling for object-oriented features of the language. See Section 2 for a full description of our type inference system. We introduce a *class table* into the type system as a means for keeping track of class definitions and to maintain the inheritance relationships between types. We use *class hierarchy analysis* to account for dynamic method dispatch. We use *object-insensitive* modeling of the fields of objects, meaning that a single set of types is shared among the fields of all objects of the same class.

We have developed a practical implementation of Jqual as an Eclipse plug-in. In Section 4 we describe an experiment in which we model Javari-style reference immutability using Jqual and analyze a sample program that was hand-annotated with Javari qualifiers. The Jqual implementation was able to infer the same types specified by hand for most terms in the program, though it raises a number of unnecessary warnings as a result of lack of precision arising from conservative approximations in the Jqual type system. In addition, Jqual identified a number of terms that could correctly be annotated `readonly` but had not been.

The principal contributions of this work are the extension previous type qualifier inference systems for the C language to address the additional language features of Java and the implementation of a practical tool for performing type qualifier analysis based on our type system.

2 Type System

We present our inference on the source language shown in Figure 1, which is a slightly simplified variation on Featherweight Java [6]. In section 4 we describe our implementation, which handles the full Java language.

Along with the source program, Jqual requires as input a *lattice file*, which is a list of the *qualifier constants* that are used in the program. This list can also specify the subtyping relationships between different qualifiers. In order to infer types for a program, we will create *qualifier variables* for each position in the source language where qualifiers can appear. For some applications programmers may include qualifier annotations in the source program to specify the types of terms. These are applied to the appropriate qualifier variables via the auxiliary function $mkq()$. The function $mkq()$ returns a fresh *qualifier variable* κ. If the function $mkq()$ is passed a qualifier constant Q as an argument, then $mkq()$ constrains κ to be equal to Q. If $mkq()$ is passed ε, then $mkq()$ makes no constraints on κ. If the target term of an annotation is a reference $ref^{Q}(C^{Q'})$, the qualifier will be applied to the value qualifier Q'.

$$
\begin{array}{rcl}
P & ::= & \varepsilon \mid \textbf{class } C \textbf{ extends } C' \ \{F_1; \ldots; M_1; \ldots\} \ P \\
F & ::= & T \ l \\
M & ::= & T \ l^{Q_s}(T_1 \ x_1, \ldots, T_n \ x_n) \ Q \ \{e\} \\
e & ::= & x \mid \textbf{null} \mid e.l \mid e_1.l := e_2 \mid e.l(e_1, \ldots, e_n) \\
 & \mid & \textbf{new } T \mid (T) \ e \\
C & ::= & \textbf{Object} \mid \langle \text{class names} \rangle \\
Q & ::= & \varepsilon \mid \langle \text{qualifier constants} \rangle \\
T & ::= & \mathit{null} \mid C^Q
\end{array}
$$

Figure 1: Source Language

In our formal language, programs P are made up of a sequence of class definitions. Each class extends exactly one other class, except for the base class *Object*. A class definition cannot appear in the program before the definition of its superclass. A class definition contains a sequence of field and method declarations. Each field is declared with a *qualified type* T, which is either *null* or a class name C paired with a qualifier Q. Qualifiers can be specified as annotations in the text of the program or left blank to indicate no explicit qualifier at that position. Methods similarly have qualified parameter and result types, as well as a qualifier Q_s on the invocation target **this** and an overall qualifier Q which can be used to track properties or effects of the method, for example whether the method has side effects or accesses shared memory. The overall method qualifier is not used in the analyses described in this paper. When a method is invoked, it evaluates to its body, which is an expression e. Expressions include variables x, which we allow to be the current object **this**. Explicit target expressions e must be given for all field accesses $e.l$ and method invocations $e.l(\ldots)$. The expression **new** C^Q creates a new instance of class C, with its fields initialized to the distinguished value **null**. We also allow casts on qualified types. The soundness of casts with respect to base types is checked at runtime. The soundness with regards to qualifiers is unchecked.

Our goal is to present the simplest language possible that will allow us to illustrate the interesting aspects of modeling type qualifiers for objects. Besides the inclusion of qualifiers, our language differs from Featherweight Java in three significant ways. First, in place of the auxiliary definitions used by Featherweight Java, we handle inherited field and method resolution differently (see below). Second, we do not include constructors in our language. We also do not include calls to `super` in our language.

2.1 Type System

The goal of our type system is to assign qualified types to every expression in the program. A key choice in designing a type system for an object oriented language is how to model objects. Our type rules construct a *class table CT* which maps class names C to *object types* σ [1], which contain fields and methods labeled with l_i:

$$
\begin{array}{rcl}
\phi & ::= & \mathit{ref}^Q(T) \mid (Q_s; T_1 \times \ldots \times T_n) \rightarrow^Q T \\
\sigma & ::= & \{l_i : \phi_i\}_{i \in [1..n]}
\end{array}
$$

Our type system is *object insensitive*, meaning that all instances of a class share the same object type. The implication is that constraints inferred on the fields of one object will also apply to the fields of all other objects of the same class.

In our type system Fields and methods are assigned *object component types* ϕ. The object component type for a field is of the form $\mathit{ref}^Q(T)$ to denote that fields are updatable and to provide a clear separation of reference-level qualifiers from value-level qualifiers [4]. Reference-level qualifiers represent properties of the reference, while value-level qualifiers represent properties of the target object referred to by the reference. For example, a field $x.f$ has the type $\mathit{ref}^Q(C^{Q'})$, where Q is the qualifier on the location of $x.f$ and Q' is the qualifier on the contents of $x.f$. The object component type for a method is of the form $(Q_s; T_1 \times \ldots \times T_n) \rightarrow^Q T$, where Q_s is the qualifier on **this**, the T_i are the argument types, T is the return type, and Q is the overall qualifier on the method. For object types σ, we write $\sigma[l_i]$ to denote the ϕ corresponding to label l_i.

$$\textsc{Sub-Class}\quad \frac{C \ \mathbf{extends}\ C' \in P}{P \vdash C \leq C'}$$

$$\textsc{Sub-Trans}\quad \frac{P \vdash C \leq C' \qquad P \vdash C' \leq C''}{P \vdash C \leq C''}$$

$$\textsc{Sub-Null}\quad \frac{}{P \vdash \mathit{null} \leq T}$$

$$\textsc{Sub-Object}\quad \frac{P \vdash C \leq C' \qquad Q \leq Q'}{P \vdash C^{Q} \leq C'^{Q'}}$$

$$\textsc{Sub-Field}\quad \frac{Q \leq Q'}{\mathit{ref}^{Q}(T) \leq \mathit{ref}^{Q'}(T)}$$

$$\textsc{Sub-Method}\quad \frac{Q'_s \leq Q_s \qquad P \vdash T'_i \leq T_i \quad i \in [1..n] \qquad Q \leq Q' \qquad P \vdash Q_r \leq {Q_r}'}{P \vdash (Q_s; {C_1}^{Q_1} \times \ldots \times {C_n}^{Q_n}) \rightarrow^{Q} {C_r}^{Q_r} \leq (Q'_s; {C_1}^{Q'_1} \times \ldots \times {C_n}^{Q'_n}) \rightarrow^{Q'} C_r^{{Q_r}'}}$$

Figure 2: Subtyping Rules

Jqual incorporates subtyping both among base types, as in Java, and among type qualifiers. Figure 2 shows the subtyping rules for Jqual. The first four rules are for classes and qualified types. In the Sub-Object rule for subtyping qualified types, notice that we place no constraints on the fields of C and C'. Such constraints are already produced by the class table construction rules (see below). The Sub-Field rule, for subtyping field types, is the standard rule for subtyping references and specifies that subtyping is only allowed for references to the same type [7]. The Sub-Method rule specifies subtyping of methods. Following the Java typing rules, the classes of the parameters and the return type must be identical in the subtype and supertype. However, the qualifiers Q_i on the parameters are treated less restrictively and can be subtyped contravariantly. Like the qualifiers on the parameter types, the qualifier Q_s for **this** is used as an input to the method and is also subtyped contravariantly.

Jqual uses a two-step type inference procedure to verify that consistent types can be assigned to all expressions in a program P. The first step proves a judgment $\varepsilon \vdash_{CT} P; CT$, meaning that the class table CT matches the class, field, and method declarations of the program P. The second step uses type inference, subtyping rules, and constraint solving to prove the judgment $CT \vdash P$, meaning that the program P type checks under class table CT. The type inference and subtyping rules can reduce the judgment to a set of subtyping relationships between qualifiers of the form $Q \leq Q'$. If a consistent solution can be found for these constraints, the type inference is successful.

Figure 3 shows the rules used to prove the class table judgment. The Empty rule is self-explanatory. The Class rule examines each class definition in turn and augments the class table to reflect the class' field and method declarations. For a class C, this rule constructs an object type σ that associates each field or method label l_i in the class definition with an object component type. For a label l_i corresponding to a field declaration F_i, a mapping is made to a field object component type. For a label l_i corresponding to a method declaration M_i, a mapping is made to a method object component type. Labels in the object type $CT[C']$ for the superclass C' that are not defined for C are mapped to the object component type associated with that label in the superclass, as dictated by the class hierarchy mechanisms of our language. The Class rule uses an the auxiliary function $mkq(Q)$ to create fresh qualifier variables and constrain them

$$\textsc{Empty}\quad \frac{}{CT \vdash_{CT} \varepsilon; CT}$$

$$\textsc{Class}$$

$$\sigma[l_i] = \left\{ \begin{array}{ll} \mathit{ref}^{\mathit{mkq}(\varepsilon)}(C^{\mathit{mkq}(Q)}) & i \in \{1 \ldots n\} \\ & \quad \text{where } F_i = C^Q\ l_i \\ (\mathit{mkq}(Q_s); C_1^{\mathit{mkq}(Q_1)} \times \ldots \times C_n^{\mathit{mkq}(Q_p)}) \rightarrow^{\mathit{mkq}(Q')} C^{\mathit{mkq}(Q)} & i \in \{n+1 \ldots n+m\} \\ \qquad\qquad \text{where } M_i = C^Q\ l_i^{Q_s}(C_1^{Q_1}\ x_1, \ldots, C_p^{Q_p}\ x_p)\ Q'\ \{e\} & \\ CT(C')[l_i] & \text{otherwise} \end{array} \right\}$$

$$\frac{\sigma[l] \leq CT(C')[l] \quad l \in \mathit{dom}(CT(C')) \qquad CT \vdash_{CT} P; CT'}{CT \vdash_{CT} C \textbf{ extends } C'\ \{F_1; \ldots; F_n; M_{n+1}; \ldots; M_{n+m}\}\ P; CT'[C \mapsto \sigma]}$$

Figure 3: Class Table Construction

to be equal to the qualifier Q if Q is non-empty. For field types, value-level qualifiers Q' may appear in the source code. There is no mechanism for specifying qualifier constraints at the reference-level in the current design of Jqual.

To model the relationship between parent and child types, we also create subtyping constraints $CT(C)[l'] \leq CT(C')[l']$ for each label l in the parent class. This reflects the class hierarchy analysis [9] approach to call graph resolution used by Jqual. This is a conservative assumption that any term in the program may actually contain at runtime any subtype of the declared type of the term. As a result, any constraints inferred on the subtypes must also be applied to the supertype. The subtyping prerequisite in the Class rule forces this to happen in the process of constructing the class table.

The second step of our type inference procedure is the application of the type inference rules shown in Figure 4. Successive applications of the Class rule are used to prove the judgment $CT \vdash P$ by examining each of the class definitions. Applying the rule for each class requires the application of the Method rule to each of the methods defined in the class. The judgment $CT, C \vdash M$ means that the method M of the class C is consistent with the class table CT. It is not necessary to type the fields defined for the class since there can not be a body or initializer code associated with a field in our language.

The Method rule retrieves the type for the method from the class table. It then proves that the body of the method has a subtype of the declared return type for the method, given that the types for **this** and the method parameters are given by the class table. The process of proving this leads to the application of the remainder of the type inference rules, which are for expressions.

The Var rule applies to the typing of variables. The Var rule also applies to **this**, since **this** is treated as a variable in the source language. The Null rule simply types **null**. The FieldAccess rule infers the type T of the expression e. It then uses the class table to look up the type of the field l for the type T. The type T_l of the object to which the field type refers is the resulting type for the field access expression since a field access in Java returns the contents of the field, which has the type T_1. Notice that since we retrieve field types from the class table, all instances of the same class share field types. The Assign rule infers the type for an assignment to a field $e_1.l$. The type of the field l is determined as in the FieldAccess rule, and the type of e_2 is inferred to be a subtype of the type of $e_1.l$. The Invoke rule infers the return type for an invocation of the method l on an object e. The type of e is resolved via the inference rules. The type of the method l as invoked on e is retrieved from the class table. Note that Jqual uses *monomorphic* analysis, where the same type is used for all invocations of a method. The types of all of the actual parameters of the method invocation are inferred via the inference rules. The qualifier on e is constrained to be a subtype of the type of **this** for the method. The types of the actual parameters are likewise constrained to be subtypes of the types of the formal parameters for the method, and the type of the invocation expression is the return type for the method. Finally, the New rule takes the type of a **new** expression from the textual class name in the expression. The qualifier is determined using the $\mathit{mkq}()$ function, as in class table construction. The Cast rule behaves similarly. Casts to a supertype are sound because the subtyping rules are respected in class table construction. A cast to a supertype followed by a cast to the original type will not cause constraints on the fields of the subtype to be unsoundly ignored since, in our system, all instances of the same class share field types.

The result of the application of the type inference and subtyping rules is a collection of qualifier constraints of the form $Q \leq Q'$. The qualifiers are either qualifier constants or qualifier variables associated with the types in the program. Solving the constraints determines the qualifier constants that apply to type for each program expression. We solve the constraints via graph reachability [4]. Contradictory qualifier constraints on an expression indicate a type error in the program.

2.2 Algorithm Complexity

The Jqual type inference algorithm is of time complexity $O(n^2)$ where n is the length of the program text. Our analysis of the algorithm complexity assumes class table lookup is implemented using a hash table large enough that lookup times reliably approach constant time complexity.

Our algorithm has two parts. The first part, class table construction, is of complexity $O(n^2)$. Class table construction requires a fixed number of operations for each field and method declaration in the program, which implies a linear running time. Applying the subtyping rules between types and their supertypes generates one constraint per qualifier location in the text of the program, so this will be require a linear number of operations. However, creating the mappings for inherited fields and methods requires the creation of one mapping for each field and method in the supertype. Therefore, the number of mappings created may be on the order of the number of fields and methods in the base type multiplied by the number of class declarations in the program, which implies quadratic running time. Alternately, we could not create these mappings directly and instead have a procedure for class table lookups that looks in the supertype for any labels that are missing from the type. However, this gives class table lookups linear running time in the size of the program, which will again make the running time of various parts of the algorithm quadratic.

The second part of our algorithm, type inference, is of complexity $O(n)$. Exactly one type inference rule is applied for each term in the program and the complexity of applying the rules is either constant time or linear in the length of the term, assuming again that class table lookups are constant time operations. Except for the Sub-Trans rule, the subtyping rules can be applied in either constant time or in time linear in the length of the terms to which they are applied. To make the work of applying the Sub-Trans rule constant, we first solve the set of constraint's generated during type inference. The number of constraints will be linear in the length of the program. Since the qualifiers we are using form a lattice, the constraints can be solved in linear time [4].

3 Reference Immutability with Jqual

As one application of type qualifier inference for Java, we have explored using Jqual to infer Javari-style qualifiers for reference immutability. Javari is a variant of the Java language that allows programmers to label variables with a new keyword `readonly` to specify that those variables will never be used to modify the objects to which they refer. The Javari compiler verifies that these specifications are always respected and produces a compilation error if they are not. While Javari can check such specifications, it cannot infer them—programmers must make all annotations by hand. Implementing Javari-style qualifiers using Jqual has the advantage of allowing these qualifiers to be inferred.

3.1 Javari

We present a brief overview of Javari. For a full discussion see [2].

The key addition to Java made by the Javari language is the keyword `readonly`. For example, the code in this method would produce an error:

```
class C {
  int x;
```

$$\textsc{Empty}\quad \frac{}{CT \vdash \varepsilon}$$

$$\textsc{Class}\quad \frac{CT, C \vdash M_k \quad k \in [1..] \qquad CT \vdash P}{CT \vdash C \textbf{ extends } C' \; \{F_1; \ldots; M_1; \ldots\} \; P}$$

$$\textsc{Method}\quad \frac{\begin{array}{c} CT(C)[l] = ({Q_s}'; {T_1}' \times \ldots \times {T_n}') \rightarrow^{Q} T' \\ [\textbf{this} \mapsto C^{{Q_s}'}, x_i \mapsto {T_i}'], CT \vdash e : T'' \qquad i \in [1..n] \qquad T'' \leq T' \end{array}}{CT, C \vdash T \; l^{Q_s}(T_1 \; x_1, \ldots, T_n \; x_n) \; Q \; \{e\}}$$

$$\textsc{Var}\quad \frac{x \in \mathit{dom}(\Gamma)}{\Gamma, CT \vdash x : \Gamma(x)}$$

$$\textsc{Null}\quad \frac{}{\Gamma, CT \vdash \textbf{null} : \mathit{null}}$$

$$\textsc{FieldAccess}\quad \frac{\Gamma, CT \vdash e : C^{Q} \qquad CT \vdash C[l] = \mathit{ref}^{Q'}(T_l)}{\Gamma, CT \vdash e.l : T_l}$$

$$\textsc{Assign}\quad \frac{\Gamma, CT \vdash e_1 : C^{Q} \qquad CT \vdash C[l] = \mathit{ref}^{Q'}(T_1) \qquad \Gamma, CT \vdash e_2 : T_2 \qquad T_2 \leq T_1}{\Gamma, CT \vdash e_1.l = e_2 : T_2}$$

$$\textsc{Invoke}\quad \frac{\begin{array}{c} \Gamma, CT \vdash e : {C_e}^{Q_e} \\ CT(C_e)[l] = (Q_s; T_1 \times \ldots \times T_n) \rightarrow^{Q'} T \qquad \Gamma, CT \vdash e_i : T_i' \qquad Q_e \leq Q_s \qquad T_i' \leq T_i \qquad i \in [1..n] \end{array}}{\Gamma, CT \vdash e.l(e_1, \ldots, e_n) : T}$$

$$\textsc{New}\quad \frac{}{\Gamma, CT \vdash \textbf{new } C^{Q} : C^{mkq(Q)}}$$

$$\textsc{Cast}\quad \frac{}{\Gamma, CT \vdash (C^{Q}) \; e : C^{mkq(Q)}}$$

Figure 4: Type Qualifier Inference Rules

```
  void f() {
     readonly C c;
     c = new C(); //ok
     c.x = 17; //error
  }
}
```

In this example, the variable `c` is labeled as `readonly`. There is an error because `c` is used to modify the value of the object to which it refers. The assignment to `c` itself is not an error because it is legal to modify the value of a `readonly` reference, just not the value of the object to which it refers. It is useful to contrast the meaning of `readonly` with the meaning of the Java keyword `final`. The *value* of a variable labeled as `final` cannot be modified:

```
class D {
  int x;
  void f() {
     final D d;
     d = new D(); //error
     d.x = 17; //ok
  }
}
```

The variable `d` is annotated `final`, so in this case the assignment to `d` is illegal. Modifying the value of `d.x`, however, is valid.

The definition of "modifying an object" includes modifying the object's *transitive state*, which includes the fields of the object as well as the objects to which those fields refer, transitively. For example, consider the following code:

```
class A { int x; }

class B { A a = new A(); }

class C {
  readonly B b = new B();

  void f() {
     b.a.x = 17; //error
  }
}
```

This code produces an error because the `readonly` reference `b` is used to modify the `x` field of `b`'s `a` field. In general, any field reached in any way from a `readonly` reference cannot be modified.

The Javari compiler also requires that `readonly` references are never assigned to variables that are not labeled `readonly`, since if this were permitted it would allow code use `readonly` references to write to memory. The following code gives an example of such a subterfuge:

```
class C {
  int x;
  void f() {
     readonly C c = new C();
     C d = c; //error
     d.x = 17;
  }
}
```

In Javari, a term that is not labeled `readonly` is considered `mutable`. In this code, the object referred to by `c` is modified by first assigning `c` to the non-`readonly` variable `d`, then using `d` to modify the object. Javari flags the assignment statement as an error. The Jqual implementation of reference immutability will handle this example differently. The assignment statement will not immediately generate an error. Instead, at that point the type of the value-level qualifier on `d` will be inferred to be a supertype of the value-level qualifier on `c` and, therefore, `readonly`. At the assignment statement to `d.x`, the type of the value-level qualifier on `d` will be inferred to be `mutable`. Since there is a contradiction between these two inferences, Jqualwill flag these statements as being badly-typed.

Along with local variables and fields, programmers can label method parameters, return types, methods themselves, or entire classes as `readonly`. If a method parameter is `readonly`, the caller can assume that objects passed to that position will not be modified. Similarly, if a method `m` is labeled as `readonly`, it is guaranteed that invoking `m` on an object `o` will not modify the object `o`. Only methods labeled as `readonly` can be invoked via `readonly` references, as demonstrated by the following code:

```
class C {
  int x;

  void f() {
     readonly C c = new C();
     c.g(); //error
  }

  void g() {
    x = 17;
  }
}
```

In this case the method `g()` is not labeled as `readonly`, so invoking it on the `readonly` variable `c` is not allowed. In Javari, methods must be explicitly labeled `readonly` in order for them to be invoked via `readonly` references. The type inference capability of Jqual allows it to infer `readonly` methods without explicit labeling. In this case `g()` could not be labeled as `readonly` and would not be inferred to be `readonly` by Jqual because it modifies the field `x` of its invocation target.

In Javari, the return type of a method can be labeled `readonly` to specify that the caller cannot modify the object referred to by the return value of the method. Lastly, entire classes can be declared `readonly` as a shorthand to specify that all references to objects of that type must be `readonly`.

Javari includes further functionality to make immutability specifications more flexible. Programmers can use the `mutable` keyword to label individual fields of a class as being writable even via a `readonly` reference. This can be useful for specifying ancillary fields that do not represent abstract state, for example, a field that caches the value of the hash code of an object. Method overloading allows a class to have two methods which are identical in name and type except with respect to whether the elements of the method signature are `readonly` or not. Javari also includes parameterized polymorphism, which allows the programmer to create methods or classes as templates in which variables representing the mutability of elements can be specified for each usage of the method or class. Casts to `readonly` allow the programmer to identify references as `readonly` in cases where the compiler does not have enough information to verify their immutability. These casts are checked at runtime.

3.2 Implementing Inference for Javari using Jqual

We added Javari-style qualifiers to Jqual and modified the type inference rules for Jqual to allow them to be inferred correctly.

To represent the mutability of references requires a lattice of two qualifiers, `mutable` and `readonly`. These qualifiers are in a subtyping relationship `mutable` $\leq$ `readonly` since `mutable` types can be used as `readonly`, but not vice-versa. Note that `mutable` here means only that a reference is not `readonly`. The `mutable` annotation, used in Javari to identify fields that are not part of the abstract state of an object, is

$$\textsc{Assign}\quad \frac{\Gamma, CT \vdash e_1 : {C_1}^{Q_1} \qquad CT \vdash C_1[l] : \mathit{ref}^{Q_2}({C_2}^{Q_3}) \qquad \Gamma, CT \vdash e_2 : T \qquad T \leq {C_2}^{Q_3} \qquad Q_2 \leq \texttt{mutable} \qquad Q_3 \neq \texttt{Mutable} \Rightarrow Q_1 \leq \texttt{mutable}}{\Gamma, CT \vdash e_1.l := e_2 : {C_2}^{Q_3}}$$

$$\textsc{FieldAccess}\quad \frac{\Gamma, CT \vdash e : {C_1}^{Q_1} \qquad CT \vdash T[l] = \mathit{ref}^{Q_2}({C_2}^{Q_3}) \qquad Q_1 \leq_{\texttt{mutable},\texttt{readonly}} Q_3}{\Gamma, CT \vdash e.l : {C_2}^{Q_3}}$$

Figure 5: Modified Type Qualifier Inference Rules for Javari Analysis

represented by a qualifier `Mutable` that can appear as a program annotation but is never inferred. `Mutable` is never inferred because we believe it represents an exception to normal behavior that should not be assumed if it is not explicitly specified by the programmer.

We model Javari by placing `readonly` and `mutable` qualifiers on the value level of a type. It is natural to extend these qualifiers to also model Java's `final`, which we do by placing them on the reference level of a type. If the reference-level qualifier on a field is `readonly`, that field is `final`. If the reference-level qualifier is `mutable`, that field is non-`final`.

Making Jqual qualifiers `readonly` and `mutable` represent the semantics of Javari—and also the extra semantics of Java's `final`—requires modifications to two of the Jqual type inference rules, as shown in Figure 5. The new Assign rule specifies that if a value is assigned to the field of an object via a reference, that reference must be `mutable` unless the field that is being modified is `Mutable`. Since the field is being assigned to, it must also be non-final, thus its reference-level qualifier must be `mutable`.

The new FieldAccess rule models the behavior of `readonly` over the transitive state of an object. The relation notated $\leq_{\texttt{mutable},\texttt{readonly}}$ is called a *gated constraint* and means that the constraint holds only with respect to the qualifiers `mutable` and `readonly`. Since the semantics implying this constraint are particular to the Javari qualifiers, it does not make sense to apply the constraint with respect to other qualifiers [5].

The other aspects of Javari semantics are either handled naturally by Jqual or are implemented through small modifications, except for templated generics. The restrictions on `readonly` methods follow from the constraints generated between the **this** qualifier on method types and the qualified types of their invocation targets. If the fields of the enclosing object are modified in the method, the **this** qualifier on the method type will be inferred to be `mutable`. Since the Invoke rule constrains the qualifier on the target object of the method invocation to be a subtype of the **this** qualifier on the method type, the target object will be inferred to be `mutable`—which will cause Jqual to raise an error if the target object is annotated as or inferred to be `readonly`. The behavior of casts matches the standard behavior of Jqual casts, but Jqual casts are not checked at runtime. Finally, `readonly` classes are implemented by constraining the qualifiers on all references to `readonly` classes to be `readonly`.

3.3 Limitations of Jqual Implementation of Javari

As previously discussed, Jqual uses an object-insensitive analysis, where all instances of a class share the same object type. Thus, templated generics are not supported by Jqual. To analyze Javari code, Jqual ignores the template parameters and forces parameterized classes and methods to have a single type, which results in Jqual producing some unnecessary warnings (see below). Jqual also does not support method polymorphism. Finally, in Javari a method of an inner class is not `readonly` if it modifies the state of the enclosing class. The Jqual implementation does not currently account for this.

4 Implementation and Experiments

4.1 Jqual implementation

We implemented Jqual as a plug-in for the Eclipse development environment using the Java Development Toolkit (JDT). We used Cqual [4] for our constraint solver via a JNI interface. Cqual maintains the constraints among qualifier variables and constants. When all of the constraints have been accumulated, Cqual searches for a solution to the constraint graph and reports the results. There are a number of features of the Java language not included in our formal system that are handled in the implementation. To distinguish among overloaded methods, we used enhanced method identifier strings comprising the method names and the names of the parameters types. Constructors are generally handled as ordinary methods by Jqual. In Javari, constructors can modify the instances of `readonly` classes that they create, and Jqual allows this as well. Local variables are treated in essentially the same way as method parameters. Interfaces are handled similarly to classes. Annotations are made to source programs as Javadoc-style comments. The following is an example of Jqual-annotated code:

```
/** @qual readonly **/ class C {

  /** @qual readonly **/ Object x;

  /** @qthis readonly
   ** @qreturn readonly
   **/
  Object f(/** @qual readonly **/ Object p) {
        /** @qual readonly **/  Object v;
         return p;
  }
}
```

Annotations can be placed as comments before any type name or before the keyword `class`. The `@qual` annotation specifies a qualifier on the subsequent type or class. Comments before a method declaration allow qualifiers to be associated with the **this** or the return type of a method using the annotations `@qthis` and `@qreturn` respectively. The lattice file for Jqual follows the same format as is used by Cqual. Jqual allows the user to specify a set of source code files to be analyzed as a group. Jqual analyzes the specified classes plus any other classes they reference (transitively) within the Eclipse project. Any classes for which source code is not available must be modeled conservatively—in the case of the Javari analysis, this means all qualifiers in the types in those classes are assumed to be mutable.

4.2 Experiment: GizmoBall

To test the Jqual implementation of Javari, we analyzed a sample program called GizmoBall, a pinball game written in the Javari variant of Java. The program consists of 68 Java files containing approximately 8,000 lines of code. In addition, we analyzed 168 files from the Java standard libraries including approximately 24,000 lines of code. We used a script to convert the Javari annotations to the format required by Jqual. We also stripped annotations for generics, which are not supported by Jqual.

We ran Jqual on the resulting program and compared the qualifiers inferred by Jqual with the annotations made by hand by Javari developers. We examined terms which were annotated as `readonly` by the Javari developers but which Jqual inferred `mutable`. Any discrepancies indicate either false positives resulting from imprecision in Jqual or incorrect annotations in the sample program. We also looked for terms on which Jqual inferred `readonly` but that were not annotated `readonly` in the sample program. These indicate either limitations in Jqual or terms that the programmers overlooked in annotating the program.

The total running time for the Jqual analysis was approximately 64 seconds on a Linux machine four 2-gigahertz processors and two gigabytes of RAM. The analysis requires approximately 35 megabytes of memory above the Eclipse overhead.

4.2.1 Invalid Annotations Inferred

There are 18 places where Jqual infers `mutable` for elements that are annotated as `readonly` in the sample program. Two of these appear to be errors in the Javari program. The other 16 are false positives, which fall into five different categories.

The most common type of false positive results from the lack of object sensitivity in Jqual. The following code shows an example of this type of false positive:

```
class A { int x; }

class B {
  A a = new A();

  void f() {
    this.a.x = 17;
  }

  readonly void g() {
    int y = this.a.x;
  }
}
```

In `f()`, Jqual applies the Assign rule and infers that `this.a` is `mutable`. It also applies the FieldAccess rule and infers that `f_this`—the *this* qualifier for the method `f()`—is `mutable`, so the function is mutable, which is correct. In `g()`, however, the FieldAccess rule is applied again to `this.a`, and `g_this` is inferred to be `mutable`, which is incorrect. The error occurs because the type for the `a` field of the class `B` is shared among all instances of `B`. If Jqual maintained separate types for the fields of `f_this` and `g_this`, the incorrect inference about `g()` would not be made. This type of false positive occurred 9 times in the sample program.

Because Jqual is not object-sensitive, it conflates the types of objects in the Javari code that use templated polymorphism. For example, in one case a false positive is given when a method in one class inserts a `readonly` reference into a `Vector` and a method in another class reads a reference from a different `Vector` and assigns it to a `mutable` reference. The Javari version of this code uses polymorphism to distinguish between `Vector`s containing `readonly` and `mutable` objects, which prevents confusion. Because Jqual cannot not distinguish between the `Vectors`, it infers an error.

Jqual generates one unnecessary warning where it conflates the constraints generated in two different calls to a method. This is a result of Jqual's monomorphic method typing.

Jqual generates three unnecessary warnings as a result of its conservative class hierarchy analysis. In class hierarchy analysis, constraints applied to a class are also applied to objects of the class's superclass. In the cases that generate the false positives, it is clear upon inspection that the objects in question can never contain objects of the subtypes upon which the constraints were applied, so the class hierarchy analysis is overly-conservative. The following code shows an example of this:

```
class A { int x; }

class B {
  A a;
  void g() {
    b = new B();
    b.x = 17;
  }
}

class C extends B{
  readonly A a;
}
```

In this example, a constraint `Q_B.a` $\leq$ `Q_C.a` is made during class table construction, where `Q_X.a` is the value-level qualifier on `X`'s `a` field. Thus `B.a` is inferred to be `readonly`, even though we can see that the variable `b` created in `g()` does not contain an instance of `C`.

Finally, one false positive occurs because Jqual makes conservative assumptions about the types of native method, in this case `Object.clone()`.

4.2.2 Missing Annotations Inferred

In addition to using Jqual to check the correctness of the annotations in the code, we also analyzed whether all possible `readonly` annotations had been made. In particular, we looked at methods that Jqual infers to be `readonly` but which are not annotated as such in the sample program. We found 113 such results.

There are two instances in the sample program where limitations in Jqual implementation causes it to incorrectly infer `readonly`. In one of these, Jqual infers `readonly` on a method of an inner classes that modifies its enclosing class. In the other, Jqual's monomorphic analysis of methods does not understand that a method is parameterized such that it can be either `readonly` or `mutable` depending on type arguments. Jqual conflates `mutable` uses of the method with `readonly` uses.

In the remaining cases, methods are correctly inferred to be `readonly`. One method appears to be an oversight on the part of the developers. The other methods are trivially `readonly`. Some are static methods or methods of static inner classes, which can only modify static data of their class. Javari does not consider this to be modification of the invocation target. Some are abstract methods, and the remainder are empty methods.

5 Related Work

The theory of type qualifier inference for C was developed in the Cqual project [4]. We have extended this work, as described above, to handle the language features of Java. In our implementation of Jqual we use Cqual as a constraint solver.

Further work on type qualifiers for C includes a general framework for specifying the semantics of type qualifiers [3]. A type system is defined by rules that describe how types are implied by language constructs. For Jqual we use a set of rules modeling the flow of data through the program (see Figure **??**), and these rules are reflected in the implementation. When the semantics of a particular analysis dictate a different set of rules, as is the case for the Javari analysis, the implementation must be modified. The work in [3] describes a language for defining how type qualifiers interact with the constructs of the C language. Users can also specify invariants that must hold under the type rules. The work includes a tool for proving that the type rules respect the invariants and for type-checking C programs under the rules. The tool does not currently support type inference. This work suggests an approach to allowing Jqual to support a richer set of analyses without requiring modifications to the implementation.

Our model for Java types is based on Featherweight Java [6] and work on object type modeling [1].

While we are unaware of previous work that has provided a general framework for type qualifier analysis in Java, previous work has used type qualifiers to perform specific analyses. [8] uses type qualifiers to automate and check the use of proxies in Java programs. This work also uses a type system modeled on Featherweight Java. It analyzes Java code at the byte-code level instead of the source-code level, utilizes a less-conservative set-type mechanism for static type resolution, and performs byte-code transformation to implement runtime checks and support for proxies. The Javari language [2], described previously, is another application of type qualifiers in Java.

6 Conclusion

We have described Jqual, a type system for type qualifier inference and a practical implementation of that system. We have shown the ability of Jqual to implement a qualifier-based analysis in Java. Our experiment has shown that the Jqual was able to infer the correct types for the majority of terms in the sample application, although imprecision, particularly the lack of object sensitivity and method polymorphism, caused a number of false positives in our analysis.

References

[1] M. Abadi and L. Cardelli. *A Theory of Objects*. Springer, 1996.

[2] A. Birka and M. D. Ernst. A practical type system and language for reference immutability. In *Object-Oriented Programming Systems, Languages, and Applications (OOPSLA 2004)*, pages 35–49, Vancouver, BC, Canada, October 26–28, 2004.

[3] B. Chin, S. Markstrum, and T. Millstein. Semantic type qualifiers. *SIGPLAN Not.*, 40(6):85–95, 2005.

[4] J. S. Foster. *Type Qualifiers: Lightweight Specifications to Improve Software Quality*. PhD thesis, University of California, Berkeley, Dec. 2002.

[5] J. S. Foster, R. Johnson, J. Kodumal, and A. Aiken. Flow-Insensitive Type Qualifiers. *ACM Transactions on Programming Languages and Systems*, to appear.

[6] A. Igarashi, B. Pierce, and P. Wadler. Featherweight Java: A minimal core calculus for Java and GJ. In L. Meissner, editor, *Proceedings of the 1999 ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages & Applications (OOPSLA'99)*, volume 34(10), pages 132–146, N. Y., 1999.

[7] B. C. Pierce. *Types and Programming Languages*. The MIT Press, 2002.

[8] P. Pratikakis, J. Spacco, and M. Hicks. Transparent Proxies for Java Futures. In *Proceedings of the nineteenth annual conference on Object-oriented programming systems, languages, and applications*, pages 206–223, Oct. 2004.

[9] V. Sundaresan, L. Hendren, C. Razafimahefa, R. Valle-Rai, P. Lam, E. Gagnon, and C. Godin. Practical virtual method call resolution for Java. *ACM SIGPLAN Notices*, 35(10):264–280, 2000.