Automatic Inference of Stationary Fields: a Generalization of Java's Final Fields

Christopher Unkel Monica S. Lam

Computer Systems Laboratory
Stanford University
Stanford, CA 94305
{cunkel,lam}@stanford.edu

Abstract

Java programmers can document that the relationship between two objects is unchanging by declaring the field that encodes that relationship to be `final`. This information can be used in program understanding and detection of errors in new code additions. Unfortunately, few fields in programs are actually declared `final`. Programs often contain fields that could be `final`, but are not declared so. Moreover, the definition of `final` has restrictions on initialization that limit its applicability.

We introduce *stationary fields* as a generalization of `final`. A field in a program is stationary if, for every object that contains it, all writes to the field occur before all the reads. Unlike the definition of `final` fields, there can be multiple writes during initialization, and initialization can span multiple methods.

We have developed an efficient algorithm for inferring which fields are stationary in a program, based on the observation that many fields acquire their value very close to object creation. We presume that an object's initialization phase has concluded when its reference is saved in some heap object. We perform precise analysis only regarding recently created objects. Applying our algorithm to real-world Java programs demonstrates that stationary fields are more common than `final` fields: 44–59% vs. 11–17% respectively in our benchmarks.

These surprising results have several significant implications. First, substantial portions of Java programs appear to be written in a functional style. Second, initialization of these fields occurs very close to object creation, when very good alias information is available. These results open the door for more accurate and efficient pointer alias analysis.

Categories and Subject Descriptors F.3.3 [*Studies of Program Constructs*]: Object-oriented constructs; F.3.2 [*Semantics of Programming Languages*]: Program analysis

General Terms Algorithms, Languages

Keywords stationary, Java, final, initialization

This work was supported in part by the National Science Foundation under Grant No. 0326227.

1. Introduction

In programs written in an object-oriented programming language, it is often the case that one object has a fixed relationship to another object. For example, an object composed of smaller objects will have a fixed relationship to each of its components. Such relationships are frequently captured in some fields of an object soon after the object is created. The field may be written multiple times during the initialization phase, but it stabilizes before it is used and remains constant for the rest of its object's lifetime. We refer to such fields as *stationary*.

Knowing which fields are stationary provides an object-oriented basis for reasoning about aliases for objects across time. Programmers naturally use these invariants when reasoning about their programs. They don't worry that an object's parent has changed, because they know—or perhaps assume—that no code ever alters a child's parent. When programmers want to know the identity of the window's parent, they will start looking close to where the window is created to find out what the relationship is. Stationary fields offer a different approach to reasoning about objects. We shall show that it is relative easy to find stationary fields in a program. Once we know that a field is stationary, we can conclude that two reads of the same field of an object are the same without tracking all the pointers that may possibly point to the object of interest. This suggests that it is beneficial for compilers and program analysis tools to understand stationary fields, as programmers already do. This may lead to a more precise points-to analysis with less effort.

1.1 Final Fields in Java

The presence of unvarying relationships between objects is not a new observation, as this notion motivated the design of Java's final fields. A Java programmer can declare an *object instance field* (Java nomenclature for fields inside objects) to be `final`, which, informally, means that it does not change once initialized. *The Java Language Specification* observes: "Declaring a variable `final` can serve as useful documentation that its value will not change and can help avoid programming errors." (Gosling et al. 2005) The Java compiler provides aid in avoiding mistakes by enforcing properties in code it compiles. These properties combine to ensure that `final` fields, once initialized by the constructor, are not modified by ordinary means. (Declaring a field `final` does not guarantee absolutely that it is constant: it may still be modified by extraordinary means such as reflection, native methods, and other implementation-dependent functionality.) Declaring a field `final` prevents programmers from making the mistake of modifying fields that should be constant, by directing the compiler to reject programs that erroneously change those fields.

The properties of `final` fields are defined so that verifying them fits within Java's model of compilation and dynamic class loading. Specifically, because Java classes may be compiled separately, and because they are then loaded separately, it must be possible to verify the property by examining a single class at a time. In fact, it is possible to verify the property by examining a single method at a time.

1.2 Automatic Inference of Final Fields

We speculate that there are many more fields that are being used like they are final without being declared as such. Automatic final field inference is useful because the information can serve as documentation and may be used for other analyses. It can also be used to find errors in subsequent code modifications in the software development process. Suppose a field is used originally as a final field; if subsequent code additions violate this pattern, then it may be worthwhile to flag the inconsistency as a possible error. Many recent projects have used the idea of intent to infer properties in the absence of accurate declarations or documentation (Engler et al. 2001; Heine and Lam 2003, 2006; Kremenek et al. 2006; Livshits and Zimmermann 2005; Williams and Hollingsworth 2005; Yang et al. 2006). Such properties have been used to find thousands of critical errors in programs.

We can infer if a field can be legally declared as final in a program, by examining the usage pattern of the field in the code. Note that in the presence of dynamic loading, the inferred property holds only for the code examined.

The Java final field modifier is defined to admit a relatively simple verification procedure. As a result, it is not as generally applicable as it could be. There are several important limitations:

- A final field must be assigned exactly once on each execution path through each constructor defined for the class containing it. This definition excludes the case where the field is first defined for the common case, and based on some exceptional conditions, the field can be updated before the field is ever accessed otherwise.
- A final field must not be assigned outside the constructors of the class declaring it. This is restrictive because the constructor might invoke utility procedures to perform initialization; or, an object factory may create and return the object, with the method using the factory responsible for the initialization.
- A final field is defined based on textual properties. A program may use a field as if it is a final even though it contains unexecuted code that violates such properties. This is especially prevalent if large, general Java libraries are used.
- Not all cases where a final field is read before it is assigned are identified and prohibited: those through aliases to `this`, or in methods invoked by the constructor, are not found. However, the fact that most cases are identified and prohibitied suggests that it is undesirable to read such a field before it is written.

As we show in our empirical results, the definition of final fields greatly restricts the number of fields encoding unvarying relationships to which it can be applied.

1.3 Stationary Fields

In practice, the initialization of a stationary field may span different methods. The field may be written to multiple times during the initialization process. What is important is that the value of the field must stabilize before the field is used. That is, all reads of a stationary field of the same object are guaranteed to follow all the writes, and thus must return the same value.

Thus, we say that a field f in class c is *stationary* in a program if all writes to field f in every object o of class c occur before all reads of f in o.

1.4 Inferring Stationary Fields

This paper presents an algorithm for finding stationary fields in Java programs. The analysis is subject to the usual caveats regarding reflection and native methods, but is conservative otherwise. The results are thus useful for error detection and program understanding tools, but should not be used for program optimization in the general case.

Our algorithm makes the simplifying assumption that an object's stationary fields are initialized soon after the object is created, and that the initialization, which may involve multiple writes and in multiple procedures, is complete before it is released for use through other objects. Releasing an object for use usually means making some other object refer to it: placing it in a list, registering a handler that uses it, etc. In general, once an object is connected to other heap objects, it can be touched by many different pieces of code. Thus, the object's initialization phase usually ends when it is pointed to by some heap object. (In Java, all objects are allocated on the heap; only references to objects and primitive types such as `int` may be stored in stack variables.)

We use a flow and context sensitive analysis to track the reads and writes of all pointer variables and heap object fields. We keep track of the identity of each newly created object accurately during its initialization phase. We accurately follow how its references are assigned to local variables and passed as input parameters or return values. However, once the object is stored into some heap reference, it is assumed to exit its initialization phase. We abstract its identity away and represent it with a special *lost* object. Any write to a field of this special object would render the field not stationary. Because we do not have to keep track of the pointees in the fields of heap objects, the algorithm converges quickly.

1.5 Experimental Results

We have implemented the algorithms presented in this paper to automatically infer final fields and stationary fields in Java programs. We have applied the algorithms to 19 real-world Java programs and analyzed the results. To help validate our static analysis algorithm, we have also developed a dynamic analysis that records, for each field, if a write follows a read of the same location. Clearly, no such pairs should ever be found for any stationary fields identified by our algorithm. We have found this sanity check to be helpful in providing confidence that the algorithm and the implementation are correct. In addition, we also record the number of times stationary fields are read so as to assess if these fields are important in the execution of a program.

1.6 Contributions

The key contributions of this paper are:

- The concept of *stationary fields*, a generalization of Java's `final` that captures a wider range of unchanging fields, primarily by relaxing the requirements on initialization.
- An efficient interprocedural algorithm for finding stationary fields in Java programs.
- Empirical results on the number of inferred final and stationary fields across 19 real-world Java programs. In total, our algorithm analyzed 88292 classes and 620884 methods. We found that stationary fields are widely present in real-world Java programs and more common than `final` fields: 44–59% vs. 11–17%. The experiments also address quantitatively how much each difference between the definitions of stationary and `final`

contributes to the prevalence of stationary fields, and show that inferring `final` fields is valuable in its own right.

- Dynamic results showing that stationary fields are an important part of the runtime behavior of programs.

1.7 Paper Organization

The remainder of this paper is organized as follows. We show an algorithm for inferring `final` fields in Section 2 and then give our algorithm for finding stationary fields (Section 3). In Section 4 we present our experiences applying our algorithm to Java programs. We briefly discuss related work in Section 5 and conclude in Section 6.

2. Final Fields

This section describes the algorithm used by the Java compiler to verify the legality of code that includes `final` instance fields. We then progress to how we might infer `final` fields.

2.1 Verifying Final Fields

The *Java Language Specification* provides an algorithm that is used by a Java compiler to verify that programs that declare `final` fields do not misuse them. Without loss of generality, let us just consider the more complex case where the final instance field is not initialized in the declaration in the following.

Intuitively, a final field is one that is assigned exactly once during any normal execution of the constructor of the class declaring it, and never assigned elsewhere. (There is no requirement on the assignment of a final field should the constructor terminate abruptly, that is, by throwing an exception.) However, since it is undecidable to determine all the possible executed paths statically, the definition of a final field is specified by the verification technique used, as outlined below.

Let `f` be a final field in class `c`.

1. There is an error if any of the methods that are not constructors of `c` contain an assignment to field `f`. Note that constructors of classes derived from `c` do not count as constructors of `c`.
2. Perform a data flow analysis to determine if the field `f` is *definitely assigned*, *definitely unassigned*, or neither, at each program point.
   - Field `f` is definitely unassigned at the entry of the constructor.
   - Executing an assignment "`this.f=`" or "`f=`", provided there is no local variable `f`, leaves field `f` definitely assigned.
   - At a control-flow join point, `f` is definitely assigned or unassigned, respectively, iff it is definitely assigned or unassigned, respectively, on both incoming branches. Otherwise, `f` is considered to be neither.
   - No other statements alter whether `f` is definitely assigned or definitely unassigned.
3. Report an error if:
   - `f` is not definitely assigned at the normal exit of the constructor.
   - `f` is not definitely unassigned immediately prior to an assignment "`this.f=`" or "`f=`".
   - `f` is not definitely assigned immediately prior to a use of "`this.f`" or "`f`".
   - the constructor assigns to `f` through some variable other than `this`, e.g. with "`other.f=`".

```
class Button {
  private Window parent;

  void setParent(Window parent) {
    this.parent = parent;
  }

  void destroy() {
    parent.removeChild(this);
  }

  void onClick() {
    parent.closeWindow();
  }
}


Button b = ButtonFactory.newButton();
b.setParent(mainWindow);
mainWindow.addChild(b);
```

Figure 1. Code fragment from and using button object showing stationary fields.

Notice that the legality of code that uses `final` can be verified by examining a single method at a time. This property of `final` allows it to fit within Java's compilation and loading process.

2.2 Inferring Final

The algorithm for verifying `final` can also be used to infer if a program uses a field like it is final, assuming all the code to be executed is available. We simply assume that a field is final, and execute the procedure for verifying a final field. If no errors are produced, the field may safely be declared `final`, given all the code we have at hand. If we do this for all fields, the result is a superset of fields that are declared final, given that the input code does not contain errors.

3. Stationary Fields

We have previously introduced stationary fields. We will next give some intuition on some kinds of stationary fields we might expect to find in program, and then describe our algorithm for finding stationary fields.

3.1 Stationary and Nonstationary Fields

Consider the code fragment in Figure 1, which might appear in a GUI-based program. Assume that this is only a fragment of our button class, but that none of the other methods modify field `parent`. The button's `parent` field shows a common type of stationary field: it encodes the object's position in a hierarchy in which objects may be created or destroyed, but in which they never move. This example also shows the use of a factory method, a common motivation for initialization outside the constructor. The factory method itself calls the constructor for `Button`, so we cannot provide arguments. The example also shows why stationary fields are useful for reasoning about programs: we know that the button is always removed from the window to which it is added because the `parent` field is stationary. Another common case of stationary fields occurs when an object is created as a composite of smaller objects.

Some examples of fields we would expect to be nonstationary include those encoding mutable current state of an object. For example, the current position of an iterator changes repeatedly, as does the current state of a pattern matcher such as a tokenizer. Another example is that of a *role* (Kuncak et al. 2002), where the state of an object is captured by the existence of a reference from

some other object. Changing roles are encoded by nonstationary fields.

3.2 Algorithm Overview

Our algorithm for inferring stationary reads accepts a Java program as input and outputs a set of stationary fields in that program. As with the definition of final fields, the algorithm is subject to the same caveats respecting reflection and native methods.

For stationary fields, programmers generally just read the initialization code and assume that the field is not changed in the rest of the program because they understand the meaning of the field. (Note that there may be errors in the program that violate this assumption.) Like the programmer, our algorithm tracks the reads and writes to each field carefully during initialization. But unlike the programmer, who instinctively knows which fields are stationary, our algorithm needs to analyze if a field is written into after initialization before it can declare the field to be stationary.

Our algorithm makes the simplifying assumption that an object's stationary fields are initialized before its reference is stored into any objects. We say that an object is *lost* once some other object points to it. Thus, a field f is *stationary* if

1. all the reads of field f in any object before it is lost occur after all the writes, and
2. there are no writes to field f upon any lost object.

With this assumption, we can derive a relatively efficient stationary field analysis. Our algorithm uses a hybrid approach to modeling objects. The algorithm tracks aliases precisely for all objects until they are lost; it models all lost objects coarsely by representing them with a special object denoted $\perp$. Conversely, all field dereferences are modeled as yielding the $\perp$ object. Our algorithm is fast because tracking pointees in heap objects is what makes pointer alias analysis expensive.

Our analysis is a flow-sensitive and context-sensitive interprocedural summary-based algorithm. It computes a fixedpoint that summarizes how each method may lose objects and render fields nonstationary in terms of its input parameters and return value. The result of the entire program is given by the summary computed for the entry method of the whole program. We track all the input and return parameters accurately across method invocations, and the assignments of parameters and local variables flow-sensitively.

Newly created objects are given local names preserving the context-sensitivity to the location of their creation. We rename newly created objects that have not been lost, and therefore have no aliases, as they are returned by methods. By giving such an object the name of the method that most recently returned it, we can distinguish between objects created at the same line in the program, if the call stack to that spot is different. This precision is important for analyzing factory methods. Using only local names also keeps the set of potential pointees small, speeding the computation.

3.3 Program Representation

While our algorithm is defined for the full Java programming language, for the sake of simplicity, we present the analysis over the following simplified language. There is a set of methods $m \in M$. Methods have a set of local variables $x, y \in V$. There is also a set $f \in F$ of object fields, which are accessed using syntax of the form $v.f$. The language has statements $s \in ST$ of the following forms:

- an assign statement $x = y$
- an object creation statement $x = \texttt{new}()$
- a load statement $x = y.f$
- a store statement $x.f = f$
- a seq statement $s_1; s_2$
- an if statement $\texttt{if } \sim \texttt{ then } s_1 \texttt{ else } s_2$
- a while statement $\texttt{while } \sim \texttt{ do } s$
- a method statement $m(x_1, \ldots, x_i)\{s; \texttt{return } y\}$
- an invoke statement $x = m(y_1, \ldots, y_i)$

We use the notation $\sim$ as a placeholder for the branch and loop conditions, which are irrelevant to the analysis.

3.4 Effect of Each Method

The input to our algorithm is a Java program and the main function to be invoked, and the output of the algorithm is a set of stationary fields. The fixpoint computation keeps track of all the nonstationary fields found so far in a set called $\bar{S}$; the complement of $\bar{S}$ at the end of the algorithm represents the result of the algorithm.

Our algorithm uses class hierarchy analysis (CHA) to compute the possible call targets at each invocation (Dean et al. 1995). It assumes that the target of a call site may be any method consistent with the method being called and the type of the reference through which the method is invoked. We include the class initializers of used classes as if they occured at the start of the main function.

The analysis of each method m operates on the following set of abstract objects $o \in O$:

- method input parameters $\mathbf{A} = \{\alpha_1, \alpha_2, \ldots\}$. The summary is computed parametrically with respect to the input parameters.
- allocation-site objects. This is the set of objects allocated by m and named by the allocation site. These objects are not aliased at the time of creation.
- call-site objects. This is the set of objects representing objects created by m's callees and are named by the call site.
- the $\perp$ object, used as a placeholder for untracked objects.

A method m and its callees may produce the following side effects that affect the computation of stationary fields:

- generate a set of nonstationary fields, which are added to the set of nonstationary fields found so far $\bar{S}$.
- lose a set of input objects, $L_m \subseteq \mathbf{A}$.
- write to a set of fields for each input parameter, W_m.
- read a set of fields for each input parameter, E_m.
- return a possible set of objects. It may
  - return some lost object or input parameters. These are represented by the set R_m.
  - return some object created by m or its callees that has not been lost. In addition, it may have read a set of fields D_m from the returned object, before it is returned. For the sake of simplicity, and without loss of precision, we assume that a method always returns a new object that has not yet been lost. That is, a fresh call-site object is always created to represent such an object where m is invoked.
- generate a set of alias conditions C_m of the form $\langle \alpha_i, \alpha_j, f \rangle$, indicating that f is nonstationary if arguments α_i and α_j can be the same object.

3.5 Effect of Each Statement

To compute the effect of each method, the algorithm uses a flow-sensitive analysis statement by statement to track where objects become lost, and the pointer aliases and reads of objects before they are lost. Thus, besides computing the effect of each statement on the terms introduced above for each method summary, the analysis keeps track of the following terms flow-sensitively, before and after every statement:

- P: a set of points-to relations $\langle x, o\rangle$ indicating that local variable x may point to object o.
- U: untracked objects, objects that are lost in this method or its callees.
- Q: a set of object fields $o.f$ indicating that field f of object o may have been read previously.

3.6 Inference Rules

The properties of the solution are given by inference rules in Figure 2. The notation $P[x \mapsto B]$ indicates P with all points-to relations involving x removed, and relations added for x pointing to each element of B:

$$P[x \mapsto B] \equiv \{\langle y, o\rangle \mid \langle y, o\rangle \in P \wedge x \neq y\} \cup \{\langle x, o\rangle \mid o \in B\}$$

This notation is used to discard the current contents of a variable and insert new ones, that is, to destructively update the points-to set of a variable.

The inference rules relate the points-to relations, lost objects, and reads prior to a statement to those after it. For example, rule LOAD, which reads:

$$\frac{P' = P[x \mapsto \{\bot\}] \qquad Q' = Q \cup \{o.f \mid \langle y, o\rangle \in P\}}{m, P, U, Q \vdash x = y.f \Rightarrow P', U, Q'}$$

says that $x = y.f$, when executed within method m with prior points-to relations P, untracked objects U, and prior reads Q, results in points-to relations P', unchanged untracked objects, and prior reads Q'; P' is P modified such that x points only to $\bot$; and Q' is the union of Q' with $o.f$ for every object to which y may point.

Analysis proceeds from a main function: our final conclusion is $\vdash \texttt{main}(x_1, \ldots)\{\ldots\}$. The inference rules provide the following:

- Following an allocation $x = \texttt{new}()$, x points to a fresh object that represents all objects created by any execution of that allocation statement (NEW).
- Following an assignment $x = y$, x points to all objects that y pointed to; nothing is lost or read (ASSIGN).
- After a load $x = y.f$, x points to the lost object $\bot$, which is used in place of all objects that are fetched from the heap. Field f has been read on all objects that y may point to (LOAD).
- After a store $x.f = y$, objects pointed to by y are lost. If x pointed to any lost object, then f is nonstationary. If x points to an object passed as a parameter of m, then m writes field f of that parameter. If the store overwrites a previous read, the field is nonstationary. If the store is to a parameter, and the same field of some other parameter has previously been read, the store will overwrite the read if the two parameters alias; record this alias condition so that the caller may check it (STORE).
- In a sequence of statements, $s_1; s_2$, the points-to relations after s_1 are those before s_2; likewise for the lost objects and previous reads (SEQ).
- Anything that may be lost after either branch of an `if ... then ... else ...` statement is lost after the statement; any points-to relation possible after either branch is possible after the statement; likewise with prior reads (IF).
- The points-to set, lost objects, and prior reads at the exit of a loop must be a fixed point under the body of the loop, and the fixed point must include all points-to relations, lost objects, and prior reads at the entry of the loop (WHILE).
- All substatements of compound statements execute within the same method as the compound statement (SEQ, IF, WHILE).

(NEW)
$$\frac{P' = P[x \mapsto \beta] \qquad \beta \textit{ fresh}}{m, P, U, Q \vdash x = \texttt{new}() \Rightarrow P', U, Q}$$

(ASSIGN)
$$\frac{P' = P\left[x \mapsto \{o \mid \langle y, o\rangle \in P\}\right]}{m, P, U, Q \vdash x = y \Rightarrow P', U, Q}$$

(LOAD)
$$\frac{P' = P[x \mapsto \{\bot\}] \qquad Q' = Q \cup \{o.f \mid \langle y, o\rangle \in P\}}{m, P, U, Q \vdash x = y.f \Rightarrow P', U, Q'}$$

(STORE)
$$\frac{\begin{array}{c} W_m \supseteq \{\alpha.f \mid \alpha \in \mathbf{A} \wedge \langle x, \alpha\rangle \in P\} \\ U' = U \cup \{o \mid \langle y, o\rangle \in P\} \\ (\exists o)(\langle x, o\rangle \in P \wedge o \in U) \rightarrow \left(f \in \bar{S}\right) \\ (\exists o)(o.f \in Q \wedge \langle y, o\rangle \in P) \rightarrow \left(f \in \bar{S}\right) \\ C_m \supseteq \{\langle \alpha_i, \alpha_j, f\rangle \mid i \neq j \wedge \alpha_i \in A \wedge \alpha_i.f \in Q \wedge \langle y, \alpha_j\rangle \in P\} \end{array}}{m, P, U, Q \vdash x.f = y \Rightarrow P, U', Q}$$

(SEQ)
$$\frac{\begin{array}{c} m, P, U, Q \vdash s_1 \Rightarrow P', U', Q' \\ m, P', U', Q' \vdash s_2 \Rightarrow P'', U'', Q'' \end{array}}{m, P, U, Q \vdash s_1; s_2 \Rightarrow P'', U'', Q''}$$

(IF)
$$\frac{\begin{array}{c} m, P, U, Q \vdash s_1 \Rightarrow P_1, U_1, Q_1 \\ m, P, U, Q \vdash s_2 \Rightarrow P_2, U_2, Q_2 \\ U' = U_1 \cup U_2 \qquad P' = P_1 \cup P_2 \qquad Q' = Q_1 \cup Q_2 \end{array}}{m, P, U, Q \vdash \texttt{if } \sim \texttt{ then } s_1 \texttt{ else } s_2 \Rightarrow P', U', Q'}$$

(WHILE)
$$\frac{\begin{array}{c} m, P', U', Q' \vdash s \Rightarrow P', U', Q' \\ P' \supseteq P \qquad U' \supseteq U \qquad Q' \supseteq Q \end{array}}{m, P, U, Q \vdash \texttt{while } \sim \texttt{ do } s \Rightarrow P', U', Q'}$$

(METHOD)
$$\frac{\begin{array}{c} R_m \supseteq \{o \mid \langle y, o\rangle \in P' \wedge o \in \mathbf{A}\} \\ (\exists o)(\langle y, o\rangle \in U') \rightarrow (\bot \in R_m) \\ L_m = \{\alpha \mid \alpha \in U' \wedge \alpha \in \mathbf{A}\} \\ D_m = \{f \mid (\exists o)(o.f \in Q' \wedge \langle y, o\rangle \in P')\} \\ E_m = \{\alpha.f \mid \alpha \in \mathbf{A} \wedge \alpha.f \in Q'\} \\ P = \{\langle x_1, \alpha_1\rangle, \ldots, \langle x_i, \alpha_i\rangle\} \\ U = \{\bot\} \qquad Q = \emptyset \qquad m, P, U, Q \vdash s \Rightarrow P', U', Q' \end{array}}{\vdash m(x_1, \ldots, x_i)\{s;\ \texttt{return } y\}}$$

(INVOKE)
$$\frac{\begin{array}{c} a_i.f \in W_n \wedge (\exists o)(\langle y_i, o\rangle \in P \wedge o \in U')) \rightarrow \left(f \in \bar{S}\right) \\ W_m \supseteq \{a_i.f \mid a_i \in \mathbf{A} \wedge (\exists j)(a_j.f \in W_n \wedge \langle y_j, a_i\rangle \in P\} \\ P' = P[x \mapsto B] \qquad B \supseteq (\exists i)(\alpha_i \in R_n \wedge \langle y_i, o\rangle \in P)\} \\ \beta \in B \qquad \beta \textit{ fresh} \qquad \bot \in R_n \rightarrow \bot \in B \\ U' = U \cup \{o \mid (\exists i)(\alpha_a \in L_n \wedge \langle y_i, o\rangle \in P)\} \\ Q' \supseteq Q \qquad Q' \supseteq \{\beta.f \mid f \in D_n\} \\ Q' \supseteq \{o.f \mid (\exists i)(\alpha_i.f \in E_n \wedge \langle y_i, o\rangle \in P\} \\ (\exists i, j, o)(\langle \alpha_i, \alpha_j, f\rangle \in C_n \\ \qquad \wedge \langle y_i, o\rangle \in P \wedge \langle y_j, o\rangle \in P) \rightarrow \left(f \in \bar{S}\right) \\ C_m \supseteq \{\langle \alpha_i, \alpha_j, f\rangle \mid i \neq j \wedge \alpha_i.f \in Q \wedge \\ \qquad (\exists k)(\alpha_k.f \in W_n \wedge \langle y_k, \alpha_j\rangle \in P)\} \\ C_m \supseteq \{\langle \alpha_i, \alpha_j, f\rangle \mid i \neq j \\ \qquad \wedge (\exists k, l)(\langle \alpha_k, \alpha_l, f\rangle \in C_n \\ \qquad\qquad \wedge \langle y_k, \alpha_i\rangle \in P \wedge \langle y_l, \alpha_j\rangle \in P)\} \\ \qquad\qquad \vdash n(z_1, z_2, \ldots)\{\ldots\} \end{array}}{m, P, U, Q \vdash x = n(y_1, y_2, \ldots) \Rightarrow P', U', Q'}$$

Figure 2. Inference rules for finding nonstationary fields $\bar{S}$.

- The statements inside a method definition are executed in the context of that method; when they begin only $\perp$ is lost, no fields have been read, and each formal parameter points to the corresponding placeholder object. Any parameter object that is lost at the end of the method is lost by the method. If a parameter object is pointed to by the return variable at the end of the method, the function returns that parameter. Any parameter field that has been read at the end of the method is lost by the method. Likewise, if a field of some object has been read, and that object is pointed to by the return variable, then that field has been read from the object returned by the method. If the return variable points to any lost object, the function also returns the lost object (METHOD).
- Rule INVOKE is the most complicated, as invoking a function has effects equivalent to some set of loads, stores, assignments, and allocations. Let $x = n(y_1, \ldots)$ be the invocation. If actual parameter i may refer to an object that is lost after the invocation, and n writes f of its i^{th} argument, then f is assumed to be nonstationary, because the write in n is to a lost object. This condition refers to the lost objects *after* the invocation because the summary for a method contains no information on the relative ordering of the writes and objects lost. The rule must conservatively assume that some objects are lost, and then the writes occur. If the called function n writes its i^{th} argument, and the i^{th} actual parameter may point to parameter j of the *calling* method m, then the calling method writes field f of argument j; writes are propagated outward. The properties of invocations just given are parallel to those for stores. Likewise, the fields written by the callee may overwrite previous reads, or imply that a field will be overwritten if two parameters of the caller alias, just as for a store.

  After the invocation, x points to anything n may have returned. If n may return parameter i, x may point to anything parameter i pointed to; x may also point to $\perp$ if n returns that.

  It is also assumed that any function may return some newly created, not lost object, so x points to a fresh placeholder object just as with an allocation. In this fashion, we keep the object names for allocation sites and call sites internal to a method. The effect is to substitute the caller's call-site name for the callee's internal name.

  If n loses parameter i, then anything pointed to by the argument i is lost after the invocation of n.

  Any fields of the returned object that were read by the callee have been read on the placeholder return object. If n reads fields from a parameter, then those fields have been read on any object pointed to by the corresponding argument.

3.7 Solution Procedure

We seek the least solution consistent with the rules given above: we wish $\bar{S}$ as to be small as possible, and likewise for all the sets that comprise the method summaries, and P, U, and Q for each statement.

The algorithm is efficient because the side effects of each method are represented in terms of its parameters, so no information needs to flow from the caller to the callee. Furthermore, each method can only refer to the placeholder objects representing the input parameters and the newly created objects returned by its callees. All indirect references are modeled simply as the lost object $\perp$.

Figure 3 shows the pseudocode for our algorithm. The procedure is as follows:

1. Compute a call graph for the input program rooted at the entry function using CHA.

```
procedure findStationaryFields(main):
    callgraph = computeCHACallgraph(main)
    S̄ := ∅
    for each method m ∈ callgraph:
        W_m := ∅; L_m := ∅; R_m := ∅
        C_m := ∅; D_m := ∅; E_m := ∅
    for each scc ∈ topologicalSort(findSCCs(callgraph))
        repeat
            for each method m ∈ scc:
                summarizeMethod(m)
        until W_m, R_m, L_m, C_m, D_m, E_m
            stabilize for all m ∈ scc

procedure summarizeMethod(m):
    for each p ∈ programPoints(m):
        U_p := ∅; P_p := ∅; Q_p := ∅
    visitRule(methodRule(m))
    repeat
        for each s ∈ statements(m):
            visitRule(statementRule(s))
        until U_p, P_p, Q_p stabilize for all p ∈ programPoints(m)
    visitRule(methodRule(m))

procedure visitRule(r):
    add minimum items to sets referenced by r so that r holds
```

Figure 3. Pseudocode for our algorithm for identifying stationary fields.

2. Begin by initializing sets we aim to compute to the empty set: we have found no fields that are written lost, and we have not discovered the side effects of any method.
3. Find strongly-connected components within the call graph and sort them in topological order.
4. Summarize all methods within each SCC, starting with leaf SCCs, and ending with the SCC containing `main`.
5. Nontrivial strongly connected components contain recursive cycles; the summaries methods within these SCCs depend on each other. Within such an SCC, iterate until a fixed point is reached for the summaries of all methods it contains.

The procedure to summarize a method m is:

1. Maintain a value for P, U, and Q at each program point; initialize these to the empty set.
2. Set P, U, and Q prior to the first statement in the method in accordance with the METHOD rule.
3. Iterate over all statements in m until a fixed point for P and U at all points is reached.
4. When visiting each statement, add the minimum items to P, U, and Q following the statement to satisfy the appropriate rule. (Also add items to the summary for m in L_m, or note the discovery of a lost write in $\bar{S}$, when appropriate.)
5. Update the summary for m in accordance with the METHOD rule.

The above procedure constitutes a particular order for visiting the inference rules. We could of course iterate over them in any order until they converge. By using this order, we may recompute the P, U, and Q for the program points inside a method each time we visit it. This reduces the space required for the algorithm, by eliminating the need to store this information for every program point simultaneously.

Our implementation incorporates an important optimization: once a field f is assumed not to be stationary (known to be contained in $\bar{S}$), our implementation ignores side effects with respect to f in W, C, D, and E for every method. Information about such side effects has no further use; suppressing their storage keeps the method summaries small.

4. Experimental Results

In this section, we present results of applying our algorithms to infer final and stationary fields in real-world Java programs.

4.1 Methodology

We applied our algorithms to a selection of real-world programs in order to gauge its effectiveness and discover the prevalence of stationary fields.

We implemented our analysis using the Joeq compiler infrastructure (Whaley 2003). Experiments were conducted on an Opteron 150 (2.4 GHz) with Sun JDK 1.5.0_03 running on CentOS 4.

We selected a group of benchmarks from the most-downloaded Java programs on Sourceforge, selecting only those that would compile as standalone programs. (The version we use in each case is what was at the head the source code repository as of 15 November 2003.) These are all real-world programs with thousands of users. In addition, we include the SPEC JVM98 benchmarks because these benchmarks come with input sets and thus make dynamic analysis possible. We exclude the benchmark check from the SPEC JVM98 benchmarks, which is a JVM testsuite.

Figure 4 lists the programs we used as benchmarks, along with a description, statistics about their size, and the runtime of our analysis. The figure gives the number of classes in the call graph for each program; the number of methods defined by those classes; and the number of methods actually included in the call graph for the program. These numbers include library methods and classes. Many of these are sizeable programs. Gruntspud, the largest, uses over four thousand classes, approximately half of which are in the application itself rather than the Java libraries.

The number of methods in the call graph is typically about two-thirds of those defined by the included classes. This indicates that many applications that use classes, especially from libraries, use only a portion of the functionality that those classes offer. The number of methods in the call graph appears to fall into two categories: less than 17,000 methods, or more than 32,000 methods. This is because the call graph for the Java libraries includes some very large strongly connected components. The smaller call graphs exclude one of these very large components.

The presence of this large component has a strong effect on the analysis time. Those applications that do not include it are completed in under 15 minutes; those that do include it generally take more than an hour. The algorithm must iterate over all the methods in each SCC until a fixed point is reached for their summaries; in these cases one SCC includes nearly half the program. Even on the largest programs, analysis is complete in under two hours, an acceptable amount of time for yielding whole-program information. Our implementation is not highly optimized. It would be possible to apply techniques such as precomputation of results for the system library to minimize the runtime.

4.2 Static Analysis of Stationary and Final Fields

Figures 5, 6, and 7 show the results of applying our algorithm to our benchmarks. Fields are classified either stationary or nonstationary. Only instance fields from which the methods in the call graph include at least one load are included. Figures 5 and 7 include all packages except `sun.*`, which contains primarily JVM internals, for fields of reference and primitive type respectively. The results for reference-typed fields in Figure 6 also exclude the core Java libraries in packages `java.*` and `javax.*` and should generally represent the application itself and its libraries.

The results show that stationary fields are prevalent in Java programs: for reference-typed fields, the stationary percentage ranges from 44 to 59 in the programs, when the Java libraries are included. Even when the Java libraries are excluded, more than 30% of fields are stationary in every application. In appliation portion of some of the smaller programs, more than 60% of the fields are stationary. For fields that hold primitive types, stationary fields are modestly less common, but are still more than 30% in all the benchmarks.

It is surprising that about half of all the fields in each of these Java programs program are stationary. This suggests that significant portions of Java programs are "functional" in nature, where data are initialized and not changed later. Obviously, garbage collection plays an important role in encouraging this style of programming.

4.2.1 Comparing Stationary with Final Fields

We also analyzed all the fields of the programs to determine if they were or could be declared final, using the algorithm given in Section 2.2. In the figures, fields are classified as one of:

- declared as **final**;
- undeclared final (**uf**), fields that are not declared final but for which such a declaration would be legal considering all the code contained in each program and its libraries;
- call-graph final (**cgf**), fields that are not final or undeclared final, but that could legally be declared final when considering only the code within the call graph we used for each program; or
- not final (**nf**).

There are many fewer declared `final` fields than stationary fields: less than 20% of the fields are declared final in both the full programs as well as just the application codes. The results suggest that automatic inference of final fields is useful, as many fields are used like they are final but are not declared as such.

Nonetheless, almost 20% of the fields in full programs are found to be stationary yet cannot be inferred to be final. This means that the relaxed definition of initialization of stationary fields is significant.

4.3 Stationary Nonfinal Fields

The table in Figure 8 quantifies the reasons why stationary is stronger than final. The three main reasons are:

- **uninitialized**, potentially uninitialized at the end of a constructor;
- **multiply initialized**, potentially multiply initialized within the constructor; or
- **outside constructor**, assigned outside the constructor of the class defining that field.

Notice that fields can belong to more than one category, so the columns sum to more than 100%. The results show that potential lack of initialization is the most common reason that fields cannot be declared final, appearing for a majority of fields. Assignment outside the constructor is next most common, occurring for around half of fields. Multiple assignment in the constructor is not common, but does occur in a handful of places in all programs.

In every program, the number of uninitialized fields is strictly greater than the number of fields assigned outside the constructor. This might be surprising; after all, if we don't assign the field inside the constructor, we would certainly expect an assignment somewhere else. Bear in mind, however, that this result only reports that a field is *potentially* unassigned. Constructors may leave fields

project	description	classes	methods	methods in callgraph	analysis time (m)
azureus	bittorrent client	1710	16243	11576	8
columba	graphical email client	4447	37400	30494	76
findbugs	find bugs in Java programs	1929	16804	11511	13
freetts	speech synthesis system	3614	32291	25551	62
gruntspud	graphical CVS client	4479	37258	31245	78
jbidwatcher	auction site tracking tool	3816	33714	27375	81
jboss	j2ee application server	3687	33045	26052	65
jedit	programmer's text editor	4182	35856	30280	92
jetty	HTTP server and servlet container	1450	13001	9171	7
jgraph	graph objects and algorithms	3653	33090	26322	75
joone	neural net framework	3646	32937	25914	71
jxplorer	LDAP browser	4053	35459	29087	106
l2j	game server	1601	14170	10008	7
megamek	networked Battletech game	3983	36130	30105	79
nfcchat	distributed chat client	3610	32349	25565	60
openwfe	workflow engine	3669	33048	25937	62
pmd	Java program analyzer	1599	13680	10228	8
spec/compress	modified Lev-Zimpel compression	3613	32314	25608	61
spec/db	in-memory database	3605	32321	25613	62
spec/jack	parser generator	3654	32405	25865	62
spec/javac	Java compiler	3769	33256	26718	66
spec/jess	expert shell system	3749	32772	26170	63
spec/mpegaudio	decompress MP3 audio	3645	32508	25823	65
spec/mtrt	ray tracer	3627	32406	25734	61
sshtools	ssh terminal	3830	33827	26809	77
umldot	make UML class diagrams from Java code	3672	32542	26123	74

Figure 4. Benchmarks used in our experiments.

	total	% stationary					% nonstationary					%	%
project	fields	final	uf	cgf	nf	total	final	uf	cgf	nf	total	final	sta.
azureus	1601	15	16	2	17	50	1	4	0	44	50	17	50
columba	4541	10	17	3	17	46	1	7	0	45	54	11	46
findbugs	1548	12	21	6	20	58	1	3	0	38	42	13	58
freetts	3289	12	17	5	17	51	1	3	0	44	49	13	51
gruntspud	4928	13	14	3	14	45	2	8	0	46	55	15	45
jbidwatcher	3601	11	17	3	16	48	1	4	0	47	52	12	48
jboss	3377	12	17	5	17	51	1	3	0	44	49	13	51
jedit	4511	13	16	3	14	46	1	7	0	45	54	15	46
jetty	1087	14	18	2	21	56	1	3	0	40	44	15	56
jgraph	3483	12	17	4	17	50	1	3	0	46	50	13	50
joone	3368	11	17	5	17	50	1	3	0	45	50	13	50
jxplorer	4334	13	14	3	15	45	1	7	0	46	55	14	45
l2j	1219	12	23	3	21	59	2	3	0	36	41	14	59
megamek	4679	9	15	3	16	44	1	11	0	44	56	11	44
nfcchat	3300	12	18	5	17	51	1	3	0	44	49	13	51
openwfe	3375	12	18	5	17	51	1	3	0	44	49	13	51
pmd	1160	14	19	2	22	57	1	3	0	39	43	14	57
spec/compress	3290	12	17	4	17	51	1	3	0	45	49	13	51
spec/db	3280	12	17	4	17	51	1	3	0	45	49	13	51
spec/jack	3336	12	17	4	17	51	1	3	0	44	49	13	51
spec/javac	3455	11	18	4	16	50	1	3	0	45	50	12	50
spec/jess	3347	12	18	4	17	51	1	3	0	44	49	13	51
spec/mpegaudio	3368	11	18	4	17	51	1	3	0	44	49	13	51
spec/mtrt	3314	12	18	4	17	51	1	3	0	45	49	13	51
sshtools	3616	11	17	4	20	53	1	3	0	43	47	12	53
umldot	3492	12	16	4	17	49	2	3	0	46	51	14	49

Figure 5. Percentages of reference-typed fields by stationary and final status, excluding packages `sun.*`.All percentages are of total fields. (**final**: declared final; **uf**: undeclared final; **cgf**: final in program's call graph; **nf**: cannot be inferred final; see Section 4.2.1 for definitions.)

project	total fields	% stationary final	uf	cgf	nf	total	% nonstationary final	uf	cgf	nf	total	% final	% sta.
azureus	533	18	13	1	7	40	1	7	0	52	60	20	40
columba	1313	6	18	2	14	41	1	14	1	44	59	7	41
findbugs	487	7	29	14	18	67	1	3	0	29	33	8	67
freetts	186	4	22	6	15	47	1	1	0	51	53	5	47
gruntspud	1692	16	9	1	9	36	2	16	0	45	64	19	36
jbidwatcher	499	4	16	3	10	33	1	10	0	56	67	5	33
jboss	268	9	24	8	16	57	2	1	0	41	43	11	57
jedit	1401	16	15	1	7	39	1	15	0	44	61	18	39
jetty	43	7	37	0	21	65	0	5	0	30	35	7	65
jgraph	400	7	18	4	12	42	0	3	0	55	58	8	42
joone	289	3	17	11	13	45	1	3	0	51	55	4	45
jxplorer	1100	14	7	1	8	30	1	20	0	49	70	16	30
l2j	157	2	60	8	17	87	0	3	0	10	13	2	87
megamek	1586	4	13	1	14	31	1	26	0	42	69	5	31
nfcchat	221	4	28	7	13	52	1	1	0	46	48	5	52
openwfe	286	4	22	10	14	50	1	3	0	45	50	5	50
pmd	116	6	33	1	25	65	0	8	0	28	35	6	65
spec/compress	210	4	23	6	14	47	1	3	0	49	53	5	47
spec/db	200	4	24	6	14	48	1	2	0	50	52	5	48
spec/jack	256	3	23	5	18	50	1	4	0	46	50	4	50
spec/javac	375	2	23	3	10	39	1	5	2	53	61	3	39
spec/jess	266	3	24	5	16	47	1	6	0	45	53	4	47
spec/mpegaudio	288	3	32	5	17	56	1	2	0	41	44	3	56
spec/mtrt	234	3	24	5	15	48	1	3	0	48	52	4	48
sshtools	529	4	23	7	33	67	0	2	0	31	33	4	67
umldot	402	14	11	3	9	37	4	2	0	57	63	18	37

Figure 6. Percentages of reference-typed fields by stationary and final status, excluding packages `java.*`, `javax.*`, and `sun.*`. All percentages are of total fields. (**final**: declared final; **uf**: undeclared final; **cgf**: final in program's call graph; **nf**: cannot be inferred final; see Section 4.2.1 for definitions.)

project	total fields	% stationary final	uf	cgf	nf	total	% nonstationary final	uf	cgf	nf	total	% final	% sta.
azureus	1099	6	10	3	24	42	1	2	0	54	58	6	42
columba	2409	3	11	5	15	34	0	2	0	64	66	3	34
findbugs	803	6	11	8	19	45	1	2	0	52	55	7	45
freetts	2072	3	12	7	15	37	0	2	1	60	63	3	37
gruntspud	2437	3	11	4	17	35	0	2	1	62	65	3	35
jbidwatcher	2238	3	12	5	16	36	0	2	0	62	64	3	36
jboss	2095	3	12	7	16	37	0	2	1	60	63	3	37
jedit	2586	3	11	4	14	32	0	2	0	65	68	3	32
jetty	615	9	11	4	17	41	1	2	0	56	59	10	41
jgraph	2140	3	12	6	15	36	0	1	1	62	64	3	36
joone	2126	3	12	7	15	37	0	2	1	60	63	3	37
jxplorer	2251	3	11	4	16	34	0	2	0	64	66	3	34
l2j	891	6	20	11	17	55	1	1	0	43	45	7	55
megamek	2662	2	10	5	15	32	0	2	1	65	68	3	32
nfcchat	2072	3	12	7	15	37	0	2	1	60	63	3	37
openwfe	2098	3	12	7	15	37	0	2	1	60	63	3	37
pmd	665	8	11	4	18	42	1	2	0	55	58	9	42
spec/compress	2098	3	12	7	15	37	0	2	1	61	63	3	37
spec/db	2082	3	12	7	15	37	0	2	1	61	63	3	37
spec/jack	2130	3	12	7	15	37	0	2	1	61	63	3	37
spec/javac	2154	3	12	7	15	36	0	2	1	61	64	3	36
spec/jess	2191	3	14	6	15	39	0	2	1	59	61	3	39
spec/mpegaudio	2144	3	12	7	15	36	0	1	1	62	64	3	36
spec/mtrt	2116	3	12	7	15	37	0	2	1	61	63	3	37
sshtools	2173	3	12	6	16	37	0	1	1	61	63	3	37
umldot	2088	3	12	6	16	36	0	2	1	62	64	3	36

Figure 7. Percentages of primitive-typed fields by stationary and final status, excluding packages `sun.*`.All percentages are of total fields. (**final**: declared final; **uf**: undeclared final; **cgf**: final in program's call graph; **nf**: cannot be inferred final; see Section 4.2.1 for definitions.)

project	uninit.	init. outside ctor.	mult. init.
azureus	88	42	3
columba	90	32	5
findbugs	87	46	3
freetts	90	33	5
gruntspud	92	36	5
jbidwatcher	90	33	6
jboss	90	33	6
jedit	92	31	5
jetty	87	42	3
jgraph	91	32	6
joone	91	33	5
jxplorer	90	33	6
l2j	88	43	3
megamek	89	34	6
nfcchat	90	32	5
openwfe	90	32	5
pmd	89	43	2
spec/compress	90	33	5
spec/db	90	33	5
spec/jack	90	34	5
spec/javac	90	33	5
spec/jess	90	33	5
spec/mpegaudio	90	32	5
spec/mtrt	90	33	5
sshtools	76	43	5
umldot	90	32	6

Figure 8. Reasons why stationary fields cannot be declared `final`. Percentages of stationary, non-final, reference-typed fields.

project	all fields sta.	all fields semi-sta.	application only sta.	application only semi-sta.
azureus	50	52	40	40
columba	46	48	41	42
findbugs	58	61	67	69
freetts	51	53	47	49
gruntspud	45	46	36	37
jbidwatcher	48	49	33	36
jboss	51	53	57	59
jedit	46	48	39	40
jetty	56	59	65	72
jgraph	50	51	42	42
joone	50	52	45	48
jxplorer	45	47	30	33
l2j	59	62	87	89
megamek	44	45	31	32
nfcchat	51	53	52	53
openwfe	51	53	50	52
pmd	57	60	65	71
spec/compress	51	53	47	50
spec/db	51	53	48	52
spec/jack	51	53	50	55
spec/javac	50	52	39	44
spec/jess	51	52	47	50
spec/mpegaudio	51	53	56	57
spec/mtrt	51	53	48	52
sshtools	53	55	67	68
umldot	49	50	37	38

Figure 9. Percentage of reference-typed fields that are semi-stationary. **All fields** excludes `sun.*`; **application only** also excludes `java.*` and `javax.*`.

unassigned on some path, allowing them to retain their default null value. Also, it is possible for some constructors of a class to initialize a field while others do not.

4.4 Inferred Final Fields

Our results show that many programs seem to be missing opportunities to declare fields to be final—even including the Java libraries, fewer than half the fields that could be marked final are.

There are several reasons why there are so many undeclared final fields. First, our analysis only considers classes that are used by the program; there may be other classes inside these libraries that mutate these fields. Second, there may be public fields for which legitimate client code could modify them. Third, the declaration of final could simply be missing.

When we look at final in the application code itself, the percentage of declared final fields is even lower. Indeed, some applications, such as l2j, do not declare *any* fields to be `final`! (The 3 final fields reported for l2j are not within the application itself, but rather within the `com.sun.management` package.) This is probably because there is no feedback from the compiler when a `final` declaration is missing, as it is compiling only a single class at a time. However, many of these fields are private, so the fields could not be modified outside the translation units. Java compilers might consider emitting a warning when a private field could be declared final but is not, in the same vein that some emit warnings for private fields that are never read within the body of the defining class.

The number of callgraph final fields suggests that many library methods that mutate some fields are unused by applications. That is, even though a library defines an object as mutable, an application treats it as fixed. It could also represent object configuration parameters that applications tend to leave in the default settings. The presence of call-graph-final fields outside of the Java libraries indicates some dead code within applications, but also that applications are using only portions of the (non-Java core) libraries that they are packaged with.

4.5 Nonstationary Final Fields

In general, we would expect most `final` fields to be stationary as well. However, there are a number of nonstationary fields reported in the final, undeclared final, and call-graph final categories, primarily undeclared final. First, there is a category of fields that are final but that cannot be shown stationary by our analysis. It is possible for a constructor to create a reference to `this` in another object, or to pass `this` as an argument to a function that does so. Any field initialized after that point in the constructor is nonstationary. In our experience, a common reason is the creation of a object of non-static inner class during construction. The inner class contains a reference to its outer class, and so the outer class is necessarily lost. For example, classes in the Java SWING library often create an inner class to use as an event handler. This implies that no field in a class derived from these classes is found to be stationary, because base class constructors execute before derived class constructors. The programs with significant numbers of nonstationary, undeclared final fields are all GUI programs.

Second, because our algorithm is conservative, it is also possible for an imprecision within it to report a stationary field as nonstationary, even if no lost writes or overwritten reads are in fact possible. The primary cause of this is the lack of relative ordering information between what is written and what is lost in the summaries of methods.

4.6 Semi-Stationary Fields

Stationary fields are defined such that they are read only after all the writes have been performed. Sometimes, a field may need to be read as part of the initalization. For example, a program may choose to write to a field only if it has not previously been assigned. This would require that the field be read first to check for nullness before the write operation. Such an action would render the field nonstationary by our definition.

Even though reads during initalization may access fields before they stabilize, our algorithm can identify the reaching definitions for such reads accurately as long as the objects accessed have not been lost. Thus, we refer to a field as *semi-stationary* if all the reads of a field of an object either occur before the object is lost, or after all the writes to the field have taken place. Semi-stationary fields are interesting because precise alias information is available for all their writes, and definitions reaching the read accesses are well identified.

Figure 9 compares the fraction of fields that are stationary and semi-stationary. The difference between the two is relatively small, never comprising more than 7% of total fields. This suggests that the simpler definition of stationary fields provides most of the benefits of our approach to track objects carefully before they are lost.

We note that if all objects of a certain class never escape into the heap, but are used and modified locally, then all the fields in the class are considered semi-stationary. For example, in the SPEC program db, the top level database object has this property. Its fields are repeatedly changed, but it is only stored in local variables.

4.7 Implementation Validation with Dynamic Analysis

As a way of validating our implementation, we compared the dynamic behavior of the SPEC JVM98 programs with the results of our inference algorithm. We used bytecode rewriting to instrument the programs, recording all instance field reads and writes, and when heap references to objects were created. For ease of implementation, we only instrumented the application code, and not the Java libraries (specifically, we excluded classes in the `java` and `sun` packages.) This allowed the instrumentation to use the Java libraries. Behavior of native methods and reflection was not captured by bytecode instrumenation.

In all cases where a field was overwritten at runtime, or where it was written while after a heap reference existed, the static analysis correctly identified this possibility. While not exhaustive, the dynamic analysis serves as a substantial test suite, which our algorithm passes.

4.8 Dynamic Analysis of Stationary and Final Fields

To determine if stationary and final fields are important to a program's execution, we instrumented the SPEC JVM98 benchmarks programs to count the number of times each instance field was read during execution. Figures 10 and 11 show the results, for reference- and primitive-typed fields respectively. The dynamic numbers exhibit more variation than the static numbers, ranging from 4 to 78% for reference fields. These numbers suggest that stationary fields are accessed in real programs. Reads of final fields are rare; reads of undeclared final fields on the other hand are common, demonstrating the value of inference of final fields. For fields of primitive type, most reads are directed at nonstationary fields; this suggests that fields of reference and primitive types are used differently in programs. For example, it is common to create a data structure out of stationary reference fields, and then the primitive fields within that structure are accessed and mutated.

We also analyzed some of the more frequently used fields to gain a better understanding of how stationary and final fields are used in practice.

Among frequently accessed stationary fields are such things as: the input stream of a scanner, the table for a hash table, as well as the hash code and key of a table entry, virtual "this" pointers (e.g. "`this$0`") used by inner classes, and input buffers.

79% of the reference field accesses in mpegaudio are directed at stationary fields that can be declared final, but are not declared as such in the program. The application mpegaudio is obfuscated so it is hard to tell what these fields are exactly. However, they are of type array of array of float. From this and the function of the program we may guess that these are statically allocated arrays.

36% of reference fields in mtrt are to stationary but not final fields. The main reason is that the initialization of an important field is performed outside the constructor, in a method called `Initialize`. This again shows that it is important to relax the initialization constraints.

Among nonstationary fields, indexes are the most common, such as into buffers and other arrays and vectors. Also appearing as nonstationary fields are reallocated structures, such as an input or vector buffer that may need to be resized, or a lazily generated list of database entries in sort order. In mtrt, 28% of the field reads are directed at nonstationary fields but can be declared as final fields. In fact, these fields are also stationary. Our algorithm cannot identify these fields as such because of imprecision discussed in Section 4.5. For primitive-typed fields, the only substantial reads of stationary, non-final fields occur in jack; the fields carry information about whether a buffer is read-only.

5. Related work

We briefly review some of the most directly related work.

We begin with previous explorations of immutability in Java. Porat et al. (2000) presented an analysis that infers final fields in Java; their implementation focused on static (class) fields rather than instance fields. Their analysis operated under an open-world assumption, limiting its ability to infer that non-private fields were final. Nonetheless, they found that static fields not declared final could be inferred to be so. Pechtchanski and Sarkar (2002) used annotations about the immutability of Java fields to enable optimizations; their implementation used programmer-supplied annotations, rather than inference. Salcianu and Rinard (2005) use an analysis similar to ours to infer purity of Java methods. Foster et al. (1999) presented an inference of type qualifiers for C, in which they discovered that many more `consts` can be used in C programs than are actually present; this corroborates our experience that programmers often neglect declarations of this kind.

Several proposals have been made for adding a "read-only" qualifier to Java, including JAC (Kniesel and Theisen 2001), Universes (Müller and Poetzch-Heffter 2001), ModeJava (Skoglund and Wrigstad 2001), and Javari (Tschantz and Ernst 2005). These all propose creating a qualifier to reference types; programs would be forbidden from using a read-only reference to modify fields. That is, with a read-only reference, the *referenced* object cannot be changed. This is orthogonal to `final` and stationary, which indicate that the qualified field may not be altered to reference another object. This distinction is also present in C and C++, where the `const` keyword may indicate either a pointer must continue to point to the same object, or that it may not be used to mutate the object pointed to (Harbison and Steele 1987). Many other languages include some method for indicating value immutability.

The Ruby programming language (Thomas et al. 2005) includes the ability to "freeze" an object, after which no field of the object may be altered. Enforcement is entirely dynamic; attempts to modify a frozen object result in an exception.

Escape analyses commonly model unescaped objects in a more precise manner than escaped objects, similarly to how we track objects without heap references more precisely (Choi et al. 1999; Whaley and Rinard 1999).

6. Conclusions

This paper introduces the notion of a stationary field, a field whose value never changes once the object containing the field is accessible via a heap reference. We have developed an efficient algorithm to identify stationary fields. It was surprising that this simple algo-

project	total reads	% stationary final	uf	cgf	nf	total	% nonstationary final	uf	cgf	nf	total	% final	% sta.
spec/compress	1144M	0	33	0	13	46	0	38	0	16	54	0	46
spec/db	231M	14	0	0	0	14	0	22	0	64	86	14	14
spec/jack	47M	2	18	0	11	32	0	2	0	67	68	2	32
spec/javac	111M	1	6	0	17	23	0	1	0	75	77	1	23
spec/jess	104M	0	2	0	2	4	0	3	0	94	96	0	4
spec/mpegaudio	492M	0	79	0	6	85	0	0	0	15	15	0	85
spec/mtrt	129M	0	14	0	36	50	0	28	0	22	50	0	50

Figure 10. Percentages of dynamic reads of reference-typed fields, excluding packages `sun.*`.All percentages are of total fields. (**final**: declared final; **uf**: undeclared final; **cgf**: final in program's call graph; **nf**: cannot be inferred final; see Section 4.2.1 for definitions.)

project	total reads	% stationary final	uf	cgf	nf	total	% nonstationary final	uf	cgf	nf	total	% final	% sta.
spec/compress	795M	0	1	0	0	1	0	0	0	99	99	0	1
spec/db	92M	0	0	0	0	0	0	0	0	100	100	0	0
spec/jack	63M	0	1	0	11	12	0	0	0	88	88	0	12
spec/javac	132M	0	5	0	0	5	0	0	0	95	95	0	5
spec/jess	139M	0	19	0	0	19	3	0	0	78	81	3	19
spec/mpegaudio	304M	0	2	0	0	2	0	0	0	98	98	0	2
spec/mtrt	167M	0	0	0	0	0	0	0	0	100	100	0	0

Figure 11. Percentages of dynamic reads of primitive-typed fields, excluding packages `sun.*`.All percentages are of total fields. (**final**: declared final; **uf**: undeclared final; **cgf**: final in program's call graph; **nf**: cannot be inferred final; see Section 4.2.1 for definitions.)

rithm identifies approximately half of the fields in Java programs to be stationary.

While the inference of stationary fields is relatively simple compared to other pointer alias analysis, it yields an important invariant property about object fields. Previous context-sensitive points-to analyses forego flow sensitivity and objects are often named by the allocation site, without context sensitivity. A field in a heap object usually points to a multitude of objects; such analysis cannot tell if two accesses of the same field, even if they are right next to each other, yield the very same object. In contrast, our results show that for about half of the fields, reading the same field off an object always yields the same result throughout the program.

This result suggests new approaches to tackling pointer alias analysis in Java. We can devise different analysis techniques for stationary and nonstationary fields. We already showed that having a bit more precision during the initialization phase provides a lot of information for stationary fields. Further analysis of the initialization code of stationary fields will provide valuable information about the identities of what the stationary object fields point to. Similarly, better understanding of nonstationary fields may lead to specialized and more efficient analysis for them too.

References

Jong-Deok Choi, Manish Gupta, Mauricio Serrano, Vugranam C. Sreedhar, and Sam Midkiff. Escape analysis for java. In *OOPSLA '99: Proceedings of the 14th ACM SIGPLAN conference on Object-oriented programming, systems, languages, and applications*, pages 1–19, New York, NY, USA, 1999. ACM Press.

Jeffrey Dean, David Grove, and Craig Chambers. Optimization of object-oriented programs using static class hierarchy analysis. In *ECOOP '95: Proceedings of the 9th European Conference on Object-Oriented Programming*, pages 77–101, London, UK, 1995. Springer-Verlag.

Dawson Engler, David Yu Chen, Seth Hallem, Andy Chou, and Bejamin Chelf. Bugs as deviant behavior: A general approach to inferring errors in systems code. In *Proceedings of Eighteenth ACM Symposium on Operating System Principles*, pages 57–72, October 2001.

Jeffrey S. Foster, Manuel Fähndrich, and Alexander Aiken. A theory of type qualifiers. In *PLDI '99: Proceedings of the ACM SIGPLAN 1999 Conference on Programming Language Design and Implementation*, pages 192–203, New York, NY, USA, 1999. ACM Press.

James Gosling, Bill Joy, Guy Steele, and Gilad Bracha. *The Java Language Specification, Third Edition*. The Java Series. Addison-Wesley, Boston, Mass., 2005.

Samuel P. Harbison and Guy Steele. *C, A Reference Manual*. Prentice-Hall, Inc., New York, NY, 1987.

David L. Heine and Monica S. Lam. A practical flow-sensitive and context-sensitive C and C++ memory leak detector. In *PLDI '03: Proceedings of the ACM SIGPLAN 2003 Conference on Programming Language Design and Implementation*, pages 168–181, New York, NY, USA, 2003. ACM Press.

David L. Heine and Monica S. Lam. Static detection of leaks in polymorphic containers. In *ICSE '06: Proceedings of the 28th International Conference on Software Engineering*, pages 252–261, New York, NY, USA, 2006. ACM Press.

Gunter Kniesel and Dirk Theisen. JAC — acess right based encapsulation for Java. *Software — Practice and Experience*, 31(6):555–576, 2001.

Ted Kremenek, Paul Twohey, Godmar Back, Andrew Ng, and Dawson Engler. From uncertainty to belief: inferring the specification within. In *USENIX'06: Proceedings of the 7th conference on USENIX Symposium on Operating Systems Design and Implementation*, pages 12–12, Berkeley, CA, USA, 2006. USENIX Association.

Viktor Kuncak, Patrick Lam, and Martin Rinard. Role analysis. In *POPL '02: Proceedings of the 29th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 17–32, New York, NY, USA, 2002. ACM Press.

Benjamin Livshits and Thomas Zimmermann. DynaMine: finding common error patterns by mining software revision histories. In *ESEC/FSE-13: Proceedings of the 10th European software engineering conference held jointly with 13th ACM SIGSOFT international symposium on Foundations of software engineering*, pages 296–305, New York, NY, USA, 2005. ACM Press.

Peter Müller and Arnt Poetzch-Heffter. A type system for controlling representation exposure in Java. In *2nd ECOOP Workshop on Formal Techniques for Java Programs*, 2001.

Igor Pechtchanski and Vivek Sarkar. Immutability specification and its applications. In *JGI '02: Proceedings of the 2002 joint ACM-ISCOPE conference on Java Grande*, pages 202–211, New York, NY, USA, 2002. ACM Press.

Sara Porat, Marina Biberstein, Larry Koved, and Bilha Mendelson. Automatic detection of immutable fields in Java. In *CASCON '00: Proceedings of the 2000 Conference of the Centre for Advanced Studies on Collaborative research*, page 10. IBM Press, 2000.

A. Salcianu and M. Rinard. Purity and side effect analysis for Java programs. *Lecture Notes in Computer Science*, 3385:199–215, 2005.

Mats Skoglund and Tobias Wrigstad. A mode system for readonly references. In Akos Frohner, editor, *Formal Techniques for Java Programs*, number 2323 in Object-Oriented Technology, ECOOP 2001 Workshop Reader, pages 30–, Berlin, Heidelberg, New York, 2001. Springer-Verlag.

Dave Thomas, Chad Fowler, and Andy Hunt. *Programming Ruby: The Pragmatic Programmer's Guide, Second Edition*. Addison-Wesley, 2005.

Matthew S. Tschantz and Michael D. Ernst. Javari: Adding reference immutability to Java. In *Object-Oriented Programming Systems, Languages, and Applications (OOPSLA 2005)*, pages 211–230, San Diego, CA, USA, October 18–20 2005.

John Whaley. Joeq: A virtual machine and compiler infrastructure. In *Proceedings of the SIGPLAN Workshop on Interpreters, Virtual Machines, and Emulators*, pages 58–66, June 2003.

John Whaley and Martin Rinard. Compositional pointer and escape analysis for java programs. In *OOPSLA '99: Proceedings of the 14th ACM SIGPLAN conference on Object-oriented programming, systems, languages, and applications*, pages 187–206, New York, NY, USA, 1999. ACM Press.

Chadd C. Williams and Jeffrey K. Hollingsworth. Recovering system specific rules from software repositories. In *MSR '05: Proceedings of the 2005 international workshop on Mining software repositories*, pages 1–5, New York, NY, USA, 2005. ACM Press.

Jinlin Yang, David Evans, Deepali Bhardwaj, Thirumalesh Bhat, and Manuvir Das. Perracotta: mining temporal API rules from imperfect traces. In *ICSE '06: Proceeding of the 28th international conference on Software engineering*, pages 282–291, New York, NY, USA, 2006. ACM Press.