Mostly-Functional Behavior in Java Programs

William C. Benton* and Charles N. Fischer

University of Wisconsin–Madison
1210 W. Dayton St. Madison, WI 53706

Abstract. We present a lightweight type-and-effect system for Java programs that features two major innovations over extant object-oriented effects systems: *initialization effects*, which are writes to an object's state while it is being constructed, and *quiescing fields*, which are fields that are never written after an object is constructed. We also present a novel taxonomy of *degrees of method purity* in object-oriented programs, which characterizes methods whose effects are confined to their receiver object. Finally, we find significant amounts of *mostly-functional* behavior in realistic Java programs: in the benchmarks we analyzed, between 48–53% of declared fields were identifiable as quiescing and between 24–78% of dynamic field reads were from quiescing fields.

Key words: Java, Program Analysis, Type-and-effect Systems

1 Introduction

Effect systems extend classical type systems with information about the computational effects exhibited by expressions, statements, and methods. Just as type signatures characterize the range of values an expression may assume, effect signatures can provide concise, useful summaries of the potential effects of a particular method invocation. Because of this capability, effect systems currently enjoy widespread application in several problem domains, including program analysis, semantics-preserving program transformation, software understanding, verification, and compile-time memory management.

In this paper, we present two innovations that can increase the expressivity and precision of effect signatures. *Initialization effects* are writes that occur to the state of an object while it is being constructed but before it is available to the rest of the program; *quiescing fields* are instance variables of an object whose values remain constant after its constructor returns. We present these in the context of a fairly simple effects system for Java, but these novel features are based on concepts orthogonal to the underlying effects system and could be adapted to more expressive systems.

We also present a notion of function purity that exploits the engineering properties of object-oriented programs: namely, that mutable state is typically accessed through the interface of the object that contains it. We describe instance

* Current affiliation: Red Hat, Inc. and University of Wisconsin–Madison

methods that are pure with respect to all mutable state outside of their receiver object as *externally pure*; we refer to methods that may read (but not write) external mutable state as *externally read-only*.

Perhaps most surprisingly, we show that realistic Java programs exhibit a substantial degree of *mostly-functional* behavior. "Mostly-functional," as coined by Knight [1], describes a programming discipline in which the presence and extent of computational effects are limited as much as possible. In the context of Java, this includes both accesses to quiescing fields — which are read-only after the object is available to the rest of the program — and the prevalence of externally-pure and externally read-only methods, whose updates to mutable state are only visible via an object's interface.

These results have several consequences for program analysis, verification, and understanding. Annotations on (externally-) pure or read-only methods aid interprocedural dependence analysis, serve as part of a method's specification, and provide documentation to human programmers. Furthermore, some such methods may be amenable to aggressive code scheduling optimizations, including asynchronous execution on multicore processors.

1.1 Overview

In the remainder of this paper, we will introduce object-oriented effects systems (Section 3) and present the simple effects system and inference rules for Java that will form the basis for our subsequent developments (Section 3.2). After this preliminary discussion, we shall introduce our major contributions:

1. The concept of *initialization effects*, which are writes that occur to a field of an object while it is being constructed, and rules to infer these (Section 4);
2. The concept of *quiescing fields*, which are never written after their containing object is constructed, and a rule to infer such fields (Section 5); and
3. A novel taxonomy of *degrees of method purity*, which extend conventional definitions of function purity to account for methods whose effects are confined to their receiver object (Section 6).

We conclude by placing our work in the context of related research efforts (Section 7) and suggesting future investigations.

2 Evaluation Infrastructure

We evaluated the applicability and feasibility of our extensions to effect systems by identifying the static and dynamic prevalence of quiescent fields in Java programs selected from the DaCapo benchmark suite [2] and by characterizing the purity of the methods in these programs. The characteristics of the benchmarks we used as inputs for our analyses are given in Figure 1; note that these counts include reachable library classes.

We analyzed these programs with version 0.92 of the GNU Classpath library, using the Soot compiler framework [3] and the DIMPLE$^+$ version of our DIMPLE

static analysis tool [4], running on version 5.1.3 of the Yap Prolog system [5]. We timed our analyses by running them on one core of a 2 GHZ Opteron workstation with 16 GB of RAM. Finally, we evaluated the dynamic prevalence of quiescing fields by instrumenting the Jikes RVM to record effects on instance fields.

Program	Statements	Classes	Fields	Methods
antlr	1390456	3729	14082	32709
bloat	1413919	3827	14524	33609
eclipse	1384719	3895	15161	33408
hsqldb	1593586	4190	17566	38504
jython	1452433	4058	14737	35604
luindex	1350107	3903	14511	32759
pmd	1509108	4265	15489	36393

Fig. 1. Characteristics of the benchmark programs and transitively reachable library code, including total statement counts, number of analyzed classes, and numbers of declared fields and methods

3 Effects and Objects

Type-and-effect systems [6] extend classical type systems so that they characterize not only the values that expressions may assume but also the *computational effects* (reads or writes to shared state) exhibited by evaluating expressions or executing statements and the (abstract) *regions* of the store in which these effects might occur. Lucassen and Gifford's original work on effects systems focused on finding expressions with noninterfering effects in ML-family languages for parallel scheduling, but effects systems have since found a wide range of applications, some of which we review in Section 7.

3.1 Background

Greenhouse and Boyland [7] developed an effects system for object-oriented languages like Java. Their effects system describes READ and WRITE effects that may occur in a hierarchy of regions:

1. The global region *All* contains all mutable state for an entire program,
2. *All* contains *static regions* that model the state of static fields and *instance regions* that contain part or all of the state of individual objects (an object may have several instance regions), and
3. individual instance regions contain regions corresponding to the state of individual instance fields.

The state of an object may contain the entire state of another object as part of its internal representation. For example, a dictionary object may contain a search

tree object that is only accessible from the instance methods of the dictionary object. To address this possibility, Greenhouse and Boyland also provide an *unshared* annotation on reference-valued fields. This annotation indicates that any object referred to by an unshared field may only be referred to by that field and thus may be considered logically part of the state of its containing object.

Greenhouse and Boyland present an intraprocedural algorithm to check user-provided effects signatures of methods and to check user-provided *unshared* annotations on object fields, but they do not present an algorithm for reconstructing effect, region, and sharing information for unannotated programs.

3.2 A Lightweight Object-Oriented Effects System

We now present a straightforward effects system and inference algorithm for Java programs. This system is based on that of Greenhouse and Boyland and is deliberately simple in order to clarify subsequent presentation of our novel techniques. While this system is not intended to be particularly sophisticated, our contributions are easily adaptable to more expressive effects systems.

Relation	Description
$\mathsf{formal}(l, i, m)$	Holds when local l respresents the formal parameter at position i (either this or a natural number) in method m.
$\mathsf{actual}(s, l, i)$	Holds when statement s invokes some method with local l as the actual parameter at position i.
$\mathsf{assign}(l_l, l_r)$	Holds when the assignment $l_l = l_r$ occurs in the program.
$\mathsf{load}(s, l, l_h, \kappa.\nu)$	Holds when a heap load statement s reads the value of the $\kappa.\nu$ field from the object referred to by l_h and copies it to l.
$\mathsf{store}(s, l_h, \kappa.\nu, l)$	Holds when a heap store statement s replaces the value of the $\kappa.\nu$ field in the object referred to by l_h with the value of l.
$\mathsf{pt}(l, O)$	Holds when O is the set of abstract objects possibly aliased by l.
$s \in m$	Holds when statement s is part of method body m.
$s \rightarrow m$	Holds when statement s contains a call that may select m, that is, if there is an edge from s to m in the call graph.

Fig. 2. Intermediate representation for simple object-oriented effects inference

We assume that the Java bytecodes of an input program and its libraries have been preprocessed to generate a conservative approximation of the call graph, a conservative may-alias relation, and the intermediate representation given in Figure 2. In Figure 2, metavariables beginning with s range over statements; S over sets of statements, l over local variables; τ over types; κ over class names; and ν over field names. (We also follow the convention that metavariables with distinct subscripts are assumed to refer to distinct object-level entities.) Note that, as in Java bytecodes, all field names are qualified with the name of their declaring class. Following Greenhouse and Boyland, we treat array loads and stores as accesses to a special field called `[]`. We treat static field loads and

stores as accesses to instance fields of a distinguished local l_ω; since we record the declaring class and field name of all field accesses, this sacrifices no precision.

The effects annotation on some statement, $\varphi(s)$, consists of READ and WRITE sets of abstract locations. Abstract locations denote sets of concrete locations in which an effect may occur and consist of a pair $\langle \rho, \kappa.\nu \rangle$, where ρ is an abstract region in which an effect may occur and $\kappa.\nu$ describes a field reference qualified by the declaring class of the field. (Because Java is a typed language, a given heap location may be referred to by exactly one kind of field reference.)

Abstract regions consist of (possibly-empty) sets of abstract object identifiers (as given by the may-alias relation pt), the distinguished abstract region $\top$, which includes all possible abstract object identifiers, or special region variables ρ_{this} or $\rho_{0\cdots n}$ denoting the regions reachable from formal parameters; these variables are used to expand method summaries at call sites. In this simple system, we summarize the effects of methods on objects referred to by their parameters but lose precision for objects reachable from the fields of method parameters.

Effect annotations, as pairs of sets, form a semilattice. The join of two effect annotations consists of the READ set of abstract locations formed by unifying the READ sets from each annotation and the WRITE set formed by unifying the WRITE sets from each annotation. Unifying two sets of abstract locations $\mathcal{A}_1$ and $\mathcal{A}_2$, as in a READ or WRITE set, proceeds as follows.

Divide each set $\mathcal{A}_i$ into the two disjoint sets $\mathcal{V}_i$ and $\mathcal{C}_i$ so that $\mathcal{V}_i$ is the set of all abstract locations from $\mathcal{A}_i$ whose regions are region variables, so that $\mathcal{C}_i$ is the set of all abstract locations from $\mathcal{A}_i$ whose regions are sets of abstract object identifiers or $\top$, and so that $\mathcal{V}_i \cup \mathcal{C}_i = \mathcal{A}_i$. $\mathcal{V}_1 \sqcup \mathcal{V}_2$ is defined simply as the union of the two sets. $\mathcal{C}_1 \sqcup \mathcal{C}_2$ consists of the union of the following:

1. The set of locations whose field identifiers appear in $\mathcal{C}_1$ or $\mathcal{C}_2$, but not both:

$$\{\langle \rho, \kappa.\nu \rangle : (\langle \rho, \kappa.\nu \rangle \in \mathcal{C}_1 \wedge \neg\exists \langle \rho', \kappa.\nu \rangle \in \mathcal{C}_2) \vee (\langle \rho, \kappa.\nu \rangle \in \mathcal{C}_2 \wedge \neg\exists \langle \rho', \kappa.\nu \rangle \in \mathcal{C}_1)\}$$

2. The set of locations formed by unifying the regions of each abstract location whose field identifier appears in $\mathcal{C}_1$ and $\mathcal{C}_2$:

$$\{\langle \rho \cup \rho', \kappa.\nu \rangle : \langle \rho, \kappa.\nu \rangle \in \mathcal{C}_1 \wedge \langle \rho', \kappa.\nu \rangle \in \mathcal{C}_2\}$$

We can then define $\mathcal{A}_1 \sqcup \mathcal{A}_2$ as $\mathcal{V}_1 \cup \mathcal{V}_2 \cup (\mathcal{C}_1 \sqcup \mathcal{C}_2)$.

We present the effects inference rules in Figure 3. The relation rpt relates a local variable to its associated region: a region variable for formal parameters, the global region $\top$ for globals, and the set of abstract objects aliased by the local in other cases. The rules READ and WRITE, which establish lower bounds on the effect annotations for load and store statements, are straightforward. SUMMARY gives the annotation summary for a method body; it is this summary that is instantiated at call sites.

The function pmap transforms effects annotations by substituting regions (or region variables) for region variables in a method summary at the point of a call to that method. pmap replaces every region variable with the region variable or explicit region associated with the local of the corresponding actual parameter, as given by the rpt relation.

$$\textsc{r-formal}\ \frac{\mathsf{formal}(l,i,m)}{\mathsf{rpt}(l,\rho_i)} \qquad \textsc{r-global}\ \frac{}{\mathsf{rpt}(l_\omega,\top)} \qquad \textsc{r-other}\ \frac{l \neq l_\omega \qquad \neg\mathsf{formal}(l,i,m) \qquad \mathsf{pt}(l,\rho)}{\mathsf{rpt}(l,\rho)}$$

$$\textsc{read}\ \frac{\mathsf{load}(s,l,l_h,\kappa.\nu) \qquad \mathsf{rpt}(l_h,\rho)}{\varphi(s) \sqsupseteq \textsc{read} : \{\langle \rho, \kappa.\nu \rangle\}\rangle} \qquad \textsc{write}\ \frac{\mathsf{store}(s,l_h,\kappa.\nu,l) \qquad \mathsf{rpt}(l_h,\rho)}{\varphi(s) \sqsupseteq \textsc{write} : \{\langle \rho, \kappa.\nu \rangle\}\rangle}$$

$$\textsc{summary}\ \frac{s_0,\cdots,s_n \in m}{\varphi(m) \sqsupseteq \varphi(s_0) \sqcup \cdots \sqcup \varphi(s_n)} \qquad \textsc{call}\ \frac{s \rightarrow m'}{\varphi(s) \sqsupseteq \mathsf{pmap}(s, \varphi(m'))}$$

Fig. 3. Lightweight effects inference rules

4 Initializers and Initialization Effects

Type-and-effect systems identify READ and WRITE effects that code may exhibit upon shared state. (Some, but not all, type-and-effect systems also identify additional effects, such as allocation, exception raising, or taking references.) If the goal of effect systems is to identify potentially interfering computational effects, this taxonomy is rather impoverished: it does not identify *initializations*, which are a special kind of WRITE that will not interfere with any other effects.

4.1 Background and Definitions

We will introduce the notion of initializations with a simple example, but first we provide some background on some properties of Java programs and objects.

Java objects are created via the `new` operator, which performs three tasks before returning a reference to the newly-allocated object: memory allocation, zero-filling object fields, and constructor method invocation. Constructors may invoke other constructors declared in the same class (via the `this()` syntax) or in superclasses (implicitly or explicitly via the `super()` syntax), but there is no way to invoke a constructor on an object after the dynamic lifetime of its constructor invocation completes. There is also no way to create and use an object without invoking its constructor. (This is the case in Java source because `new`, which is the only way to create an object, includes both object allocation and constructor invocation. These tasks correspond to distinct Java bytecode instructions – `new` and `invokespecial` – but the Java Virtual Machine will signal an error if code attempts to access an object that has been allocated but not constructed.) As a consequence, each object will be constructed exactly once before it is accessible to the code that created it.

Consider the `String` class in the Java standard library. `String` is an *immutable* class; once an instance of `String` has been created, its contents cannot be modified. A constructor for the `String` class, in setting up the state of an individual instance, will exhibit WRITE effects on that object's fields. However, these WRITE effects

will never interfere with other effects, since the only WRITE effects on a `String` will occur during its constructor and the code that creates a `String` will not be able to read its state until after the constructor completes.

Immutable classes present an extreme example, but WRITE effects on an object — even a mutable one — by its constructor will not interfere with other effects on that object that occur after the constructor completes. Classical type-and-effect systems do not discriminate between writes that occur to an object during its constructor and writes that occur after an object creation has completed. Such a system may spuriously identify WRITE effects occurring on an object during its creation as interfering with WRITE effects occurring on that object (or on other objects) that have already been created.

We will present a way to discriminate between WRITE effects to objects that have been created and initializations, which are writes that occur to an object while it is being constructed. However, we will first introduce the notion of an *initializer method* and present a algorithm for identifying which methods are initializers for given objects.

4.2 Initializer Methods

Informally, an *initializer method* (or simply an *initializer*) on some object o is a method that executes on o during the dynamic lifetime of its constructor. Since we would like to use the notion of initializer methods to identify WRITE effects that are guaranteed to occur on an object while it is being constructed, we are not interested in any method that merely may initialize part of an object's state; rather, we are interested in methods that may *only* execute on an object during the dynamic lifetime of its constructor.

If we can assume a closed world, we can identify such methods with a simple extension to a conservative static approximation of the program's call graph. We define the *receiver-sensitive call graph* (RSCG) as a set M of nodes corresponding to method bodies, a distinguished *start node* $m_{\mathsf{main}} \in M$, and a set C of labeled call-site edges. An edge is of the form $m \rightarrow_\rho m'$, indicating that m contains a call site that may transfer control to the beginning of m' with a receiver of ρ, which is either this, indicating that m' is an instance method that is invoked on the same object as m, or $\top$, indicating that m' may be invoked on some other method or is not an instance method.

We can thus define a conservative overapproximation of the initializer methods in a program inductively as follows:

1. m is an initializer on o if m is a constructor that may be executed on o (that is, a constructor declared in the class of o or in one of its superclasses).
2. m is an initializer on o if every edge to m in the RSCG is this-labeled and originates from an initializer on o.

In the remainder of this paper we will assume a closed world. We note, however, that this technique is still applicable in an open-world situation — that is, in which the entire program and libraries are not available to be analyzed. It

is still possible to identify initializers in an open world as long as the RSCG is constructed in such a way as to include conservative, sound assumptions about open parts of the program. For example, `private` methods could still soundly be identified as initializers even in an open world, since they can only be invoked from within their declaring class.

4.3 Initialization effects

An *initialization effect* is a write to an object's state that occurs during the dynamic lifetime of its constructor. Since we have already defined an initializer on some object o as a method that is only transitively invoked through zero or more this-edges in the RSCG from a constructor on o, we can identify initialization effects rather straightforwardly: An initialization effect is a WRITE effect that occurs from within an initializer and on some field of its receiver. We denote sets of initialization effects as an INIT set in an effects annotation and present updated inference rules for initializer methods and for WRITE and INIT effects in Figure 4. (The WRITE rule from Figure 4 supercedes that from Figure 3.)

$$\textsc{imeth-immed}\ \frac{m \text{ is a constructor}}{\mathsf{imeth}(m)} \qquad \textsc{imeth-trans}\ \frac{(\forall m', \rho) m' \rightarrow_\rho m \models \mathsf{imeth}(m') \wedge \rho = \rho_{\mathsf{this}}}{\mathsf{imeth}(m)}$$

$$\textsc{write}\ \frac{\mathsf{store}(s, l_h, \kappa.\nu, l) \quad \mathsf{rpt}(l_h, \rho) \quad s \in m \quad \neg(\rho = \rho_{\mathsf{this}} \wedge \mathsf{imeth}(m)))}{\varphi(s) \sqsupseteq \textsc{write} : \{\langle \rho, \kappa.\nu \rangle\}\rangle} \qquad \textsc{init}\ \frac{\mathsf{store}(s, l_h, \kappa.\nu, l) \quad s \in m \quad \mathsf{imeth}(m) \quad \mathsf{rpt}(l_h, \rho_{\mathsf{this}})}{\varphi(s) \sqsupseteq \textsc{init} : \{\langle \rho_{\mathsf{this}}, \kappa.\nu \rangle\}\rangle}$$

Fig. 4. Inference rules for initialization methods and initializer effects

If we can assume that an object will not be used until after the dynamic lifetime of its constructor, then we can guarantee that a method exhibiting INIT effects in some region ρ will not interfere with methods exhibiting other effects in ρ. This assumption, that uses of an object o by code outside of its constructor will come strictly after the dynamic lifetime of its constructor, is sound only if a reference to o cannot leak to code (say, in another thread) that could effect o before it is fully created. Such leaks are rare (and unidiomatic), so the unsound assumption is perhaps justifiable as a practical matter. For the sake of completeness, though, we briefly sketch a sound treatment of self-leaks:

A value-flow analysis could be used to indicate those constructors that might leak a reference to the constructed object. (In fact, some effect systems track reference leaking explicitly.) The classes containing such constructors could then be considered to not have initializers; as a consequence, WRITE effects occurring during the dynamic lifetime of a constructor on an object of such a class would

be conservatively (and soundly) regarded as potentially interfering with `write` effects that occur strictly after the completion of an object's constructor.

Initialization effects are a useful addition to the expressivity of object-oriented effects systems. Since the initializations of a field during an object's creation will not interfere with any reads conducted after the dynamic lifetime of the object's constructor, initialization effects allow effect systems to statically identify a greater range of static effects as noninterfering. As we shall see, inferring initialization effects also enables us to identify *quiescing fields*.

5 Quiescing Fields

Some storage is mutable for its entire lifetime, but the lifetimes of many locations can be divided into two phases: an initialization phase, in which the contents of a location are mutable, and a read-only phase, in which the contents of a location will not change. We call such fields *quiescing fields* when the phase transition happens at a statically identifiable and semantically useful place. In this section, we introduce the concept of quiescing fields, explain how we can identify them, and describe why they are useful; compare quiescing fields to Java's `final` fields; and identify the static and dynamic prevalence of quiescing fields in the Java programs from the DaCapo benchmark suite.

5.1 Quiescing Fields Defined and Identified

We define a *quiescing field* as an instance field (i.e. an object member) that is mutable while the object it contains is being constructed but that is immutable for the entire period of program execution strictly after the dynamic lifetime of its containing object's constructor. As a consequence, a quiescing field will have the same value for the entire period that the object containing the field is accessible to the code that created it (and to the rest of the program, modulo the no-leaks assumption of the previous section).

Because a quiescing field is guaranteed not to change after the object that contains it is fully constructed, quiescing fields represent a useful kind of run-time constant. If quiescing fields are prevalent in a program, identifying them can greatly simplify analyses and transformations that require accurate interprocedural data dependence information.

Given sound effects annotations including initialization effects, it is quite straightforward to identify quiescing fields: $\kappa.\nu$ is quiescing if and only if no effect annotation in the whole program contains an abstract location implicating $\kappa.\nu$ (e.g. $\langle \rho, \kappa.\nu \rangle$) in its WRITE set. (If $\kappa.\nu$ is not implicated in the INIT or WRITE sets of any effects annotation, then it is never written after allocation and is trivially a quiescing field.)

Because we need only examine every effect in the whole program once in order to determine which fields are implicated in WRITE effects — and we need not even unify method summaries at call sites in order to do so — quiescing field inference scales linearly with the number of statements in the program.

5.2 Final Fields and Quiescing Fields

The Java language [8] provides the `final` keyword and the semantic guarantee that instance variables declared as `final` will be assigned to exactly once for any given containing object. The `final` keyword thus provides both documentation for programmers and a constraint for use by analyses and transformations.

However, because the guarantee of finality is enforced by a rather coarse flow analysis (identifying "definite assignment," that is, that each final field is on the left-hand side of exactly one assignment along every possible path through each constructor of the object containing it), `final` is of limited applicability. To give one example, since all assignments to `final` fields must occur in the body of a constructor, it is impossible to share initialization code common to several constructors in a `private` instance method.

While it is often possible to restructure the code in a class so that a quiescing field meets the criteria for `final`, such a rewrite may be inconvenient. Furthermore, rewriting code so that a quiescing field is `final` may well obscure the clear meaning of the program for a human reader. Since many programmers will not immediately realize the benefits of having as many fields as possible declared `final`, manual code transformations to expose more fields as `final` are likely to be regarded as insufficiently profitable.

On the contrary, quiescing fields may be written arbitrarily many times during the dynamic lifetime of an object's constructor, not strictly in the static body of the constructor and exactly once along each path of each constructor. Quiescing fields may be read and written freely during the dynamic lifetime of their containing object's constructor, so long as they are not written to after their containing object is fully constructed. Finally, no programmer annotations are necessary to identify quiescing fields, since we present a straightforward and efficient technique for automatically inferring quiescing fields.

5.3 Static and Dynamic Prevalence of Quiescing Fields

We evaluated our definition of quiescing fields on seven of the programs from the DaCapo benchmark suite [2].

We identified the *static prevalence* of `final` and quiescing fields by determining what percentage of all fields implicated in any effect were declared `final` and what percentage were inferred to be quiescing. (Since `final` fields are, by definition, quiescing, counts of quiescing fields include counts of `final` fields.) We also instrumented the Jikes RVM in order to get a trace of all instance field reads from a benchmark execution. From this trace, we derived the percentages of dynamic instance field reads that access `final` and quiescing fields; again, the count of quiescing field reads includes `final` field reads.

Figure 5 gives our complete results; in summary, we found that between 18.7% and 22.3% of fields implicated in any static effects annotation were declared `final`; between 48% and 53% of fields implicated in any static effects annotation were identifiable as quiescing. Between 0.78% and 77.7% of dynamic reads were from `final` fields, and between 24.13% and 78.53% of fields were from quiescing

fields. The authors of the bloat, eclipse, and luindex benchmarks seem to have declared a high percentage of frequently-read quiescing fields as `final`; in the other benchmarks, the disparity between the number of dynamic reads of `final` and quiescing fields is much greater.

		Static		*Dynamic*	
Input	Time	% FF	% QF	% FF	% QF
antlr	3.11	19.89	49.25	3.65	24.13
bloat	3.16	22.30	53.01	64.05	70.05
eclipse	3.23	21.50	51.56	77.69	78.53
hsqldb	3.73	18.67	47.97	20.12	58.75
jython	3.61	18.74	52.99	19.17	50.30
luindex	3.06	20.82	51.06	43.87	47.43
pmd	3.35	19.48	48.47	0.78	24.93

Fig. 5. Static and dynamic prevalence of final and quiescing fields in select DaCapo benchmarks. Time represents analysis time in seconds; *static* numbers show the percentage of fields implicated in at least one static effect that are final (FF) and quiescing (QF); *dynamic* numbers indicate the percentage of dynamic reads in a benchmark execution that are of final (FF) and quiescing (QF) fields.

6 Degrees of Purity

Methods may be *pure*. The classic definition identifies a method that exhibits no effects on mutable state as pure. However, this definition fails to admit idempotent methods that create and modify objects in order to complete their work.

A less restrictive definition, due to Leavens et al. [9, 10] and applied for static analysis by Sălcianu and Rinard [11], characterizes a method as pure if and only if it does not modify any state that exists immediately before method entry. This definition of purity captures a notion of method purity as the absence of potential interference with other code: a method may have effects on mutable state that does not exist before it executes. Other definitions of purity are also possible; the concepts we present in this section are generally orthogonal to a base notion of purity and can be straightforwardly adapted to different definitions.

In accepting a definition of purity, we also decide which effects constitute "impure" behavior. Perhaps all side effects are "impure," as in the classical definition. Alternatively, following Leavens et al., we could ignore certain READ or WRITE effects on objects that did not exist at a method's entry. We can then identify some methods as *read-only* – these are methods that may have "impure" READ effects (but not "impure" WRITE effects) on mutable state. (Note that all pure methods are also read-only methods.)

If we are to characterize the purity of methods in typical object-oriented programs, we may wish to characterize instance methods by the effects that they have on mutable state that exists outside of the receiver object.

An *externally-pure* method is one whose "impure" READ or WRITE effects on mutable state occur only to the receiver object (that is, in the ρ_{this} region). Put another way, an externally-pure method is pure, for some definition of "pure," with respect to all state outside of the instance it is operating upon. All pure methods are also externally-pure.

An *externally-read-only* method is one whose "impure" WRITE effects on durable state occur only to the receiver object or to state that did not exist immediately before method entry. Such a method is read-only with respect to all state outside of the instance it is operating upon. All externally-pure methods are also externally-read-only.

We can combine these notions of purity with initialization effects and quiescing fields by masking INIT effects (which represent writes to the state of newly-allocated objects) and masking READ effects on quiescing fields. If we do so, we can identify a vast preponderance of instance methods as externally-pure or externally-read-only, as in Figure 6.

		Externally	
Input	Time	% Pure	% RO
antlr	4.47	79.19	81.16
bloat	4.63	77.05	78.40
eclipse	4.63	79.21	80.73
hsqldb	5.44	76.87	78.26
jython	5.25	77.02	78.31
luindex	4.39	80.30	81.89
pmd	4.88	79.05	80.50

Fig. 6. Percentage of all instance methods that are externally- pure or read-only.

7 Related Work

Work related to our contributions in this paper falls into two broad categories: work on effects systems and work on inferring fields or memory locations that are immutable for at least some part of their lifetime.

7.1 Effects Systems

Lucassen and Gifford's foundational paper on polymorphic effect systems [6] focused on identifying scheduling constraints for execution of implicitly-parallel programs, but later work has established applications of effects systems in region-based memory management [12], and in automatically providing annotations for a model checker or specification language [11]. We note that our contributions, by improving the precision of effects analyses, can also improve the precision of analyses and transformations that depend on effects annotations.

The natural compatibility of effects and objects has led to a great deal of excellent work; as we discussed in Section 3, Greenhouse and Boyland [7] devised an idiomatic, object-oriented treatment of regions and effects, but did not provide an inference algorithm. Bierman and Parkinson [13] extended the work of Greenhouse and Boyland with a semantic treatment of effects and an effects inference algorithm for a subset of Java; their work left region annotations as a responsibility for the programmer. (Effect and region reconstruction for functional languages [14] is a better-studied problem.) Most recently, Cherem and Rugina [15] presented a parameterized framework for compact effect signatures, which allows clients of effect annotations to trade precision for annotation size.

Given a notion of effects, it is possible to talk about the purity of functions. Barnett et al. [16] present several definitions of purity in the context of object-language methods that may appear in checkable specifications: observational purity (which admits memoization), strong purity (the classic definition), and weak purity (in which methods may modify newly-allocated state). Sălcianu and Rinard [11] present an analysis to identify weakly-pure methods. Barnett et al. [17] extend the Sălcianu-Rinard analysis to support iterators and the additional features, such as pass-by-reference, of the .NET runtime.

Because it masks INIT effects, our analysis can be used to identify the subset of weakly-pure methods that only exhibit INIT effects on newly-allocated objects; other analyses [11, 17] can identify a broader range of weakly pure methods. Consider, for example, a method that constructs a `StringBuffer` and then invokes its `append` method several times before returning a `String` constructed from the buffer; this is weakly pure, but would not be identified as such by our analysis because it exhibits WRITE effects as well as INIT effects. In addition, our purity analysis does not specifically treat iterators.

In general, the work described in this paper is intended to enhance the expressivity and precision of effect systems and purity analyses. It would certainly be possible to extend the system we present with more expressive features, like the information flow effects of Cherem and Rugina or the weak purity of Sălcianu and Rinard. (In particular, treating iterators demands care and a more expressive system.) In addition, one could use the results of a field uniqueness analysis (like that of Ma and Foster [18]) or object inlining transformation in order to automatically generate *unshared* annotations on object fields. Conversely, we believe that initialization effects, quiescing fields, and external purity can be introduced to an extant effect system as crosscutting concerns.

7.2 Inferring Eventual Immutablity

Several analyses (notably Porat et al. [19]) identify Java fields that are immutable, even if the fields are not declared as `final`. Most directly related to our concept of quiescing fields, however, is the *stationary fields* analysis of Unkel and Lam [20], which we shall focus on in the remainder of this review. Unkel and Lam use a flow- and context-sensitive pointer analysis to identify fields for whom every dynamic read must come after every dynamic write.

While both stationary fields and quiescing fields are capable of identifying eventually-immutable fields that are not declared `final`, and both identify about half of all fields in some set of realistic Java programs as eventually-immutable, there are several interesting differences between our approaches. Our approach is substantially more lightweight: we use a flow- and context- insensitive analysis that completes in seconds; their approach uses a flow- and context-*sensitive* analysis that takes between 7 and 106 minutes to analyze a realistic Java program. However, their approach identifies stationary fields and can also track their referents with greater precision; we merely identify quiescing fields. Therefore, both analyses can be used to improve the precision of side-effect and related analyses; theirs is better-suited to improving the precision of alias analyses.

We note that the definitions of quiescing and stationary fields are subtly incompatible: while it seems most likely that the intersection of the sets of quiescing and stationary fields for any given program would be large, there are quiescing fields that are not stationary fields (e.g. those that might be read in the constructor before a write), and there are stationary fields that are not quiescing fields (e.g. those that are written after the dynamic lifetime of a constructor, but before any use of an object). We believe that investigating the relationships between these kinds of fields — and between the analyses and transformations enabled by identifying each — represents a fruitful avenue for future work.

8 Conclusion

This paper has presented three major contributions that enhance the expressivity and precision of effects systems, purity analyses, and related analyses and transformations: the concepts of initialization effects, quiescing fields, and external method purity, as well as analyses to infer these automatically. In so doing, we have identified great amounts of mostly-functional behavior in the real-world Java programs from the DaCapo benchmark suite. Most notably, our techniques are novel, lightweight, and readily composable with extant systems and analyses.

Acknowledgments We are grateful to Nicholas Kidd and the anonymous referees for helpful comments on an earlier draft of this paper.

References

1. Knight, T.: An architecture for mostly functional languages. In: LFP '86: Proceedings of the 1986 ACM conference on LISP and functional programming, New York, NY, USA, ACM (1986) 105–112
2. Blackburn, S.M., Garner, R., Hoffman, C., Khan, A.M., McKinley, K.S., Bentzur, R., Diwan, A., Feinberg, D., Frampton, D., Guyer, S.Z., Hirzel, M., Hosking, A., Jump, M., Lee, H., Moss, J.E.B., Phansalkar, A., Stefanović, D., VanDrunen, T., von Dincklage, D., Wiedermann, B.: The DaCapo benchmarks: Java benchmarking development and analysis. In: OOPSLA '06: Proceedings of the 21st annual ACM SIGPLAN conference on Object-Oriented Programing, Systems, Languages, and Applications, New York, NY, USA, ACM Press (2006)

3. Vallée-Rai, R., Co, P., Gagnon, E., Hendren, L., Lam, P., Sundaresan, V.: Soot - a java bytecode optimization framework. In: Proceedings of the 1999 conference of the Centre for Advanced Studies on Collaborative research, IBM Press (1999) 13
4. Benton, W.C., Fischer, C.N.: Interactive, scalable, declarative program analysis: from prototype to implementation. In: PPDP '07: Proceedings of the 9th ACM SIGPLAN international conference on Principles and practice of declarative programming, New York, NY, USA, ACM (2007) 13–24
5. Rocha, R., Silva, F., Costa, V.S.: On Applying Or-Parallelism and Tabling to Logic Programs. Theory and Practice of Logic Programming Systems **5** (2005) 161–205
6. Lucassen, J.M., Gifford, D.K.: Polymorphic effect systems. In: Proceedings of the 15th ACM SIGPLAN-SIGACT symposium on Principles of programming languages, ACM Press (1988) 47–57
7. Greenhouse, A., Boyland, J.: An object-oriented effects system. In Guerraoui, R., ed.: ECOOP. Volume 1628 of Lecture Notes in Computer Science., Springer (1999) 205–229
8. Gosling, J., Joy, B., Steele, G., Bracha, G.: The Java Language Specification. 2 edn. Addison-Wesley, Boston, Mass. (2000)
9. Leavens, G.T., Baker, A.L., Ruby, C.: Preliminary design of JML: a behavioral interface specification language for Java. Technical Report TR98-06, Department of Computer Science, Iowa State University, Ames, Iowa (1998)
10. Leavens, G.T., Baker, A.L., Ruby, C.: Preliminary design of JML: a behavioral interface specification language for Java. SIGSOFT Softw. Eng. Notes **31** (2006) 1–38
11. Sălcianu, A., Rinard, M.C.: Purity and side effect analysis for Java programs. In Cousot, R., ed.: VMCAI. Volume 3385 of Lecture Notes in Computer Science., Springer (2005) 199–215
12. Tofte, M., Talpin, J.P.: Region-based memory management. Information and Computation (1997)
13. Bierman, G.M., Parkinson, M.J.: Effects and effect inference for a core Java calculus. In Bono, V., Bugliesi, M., eds.: Electronic Notes in Theoretical Computer Science. Volume 82., Elsevier (2003)
14. Talpin, J., Jouvelot, P.: Polymorphic type, region and effect inference. Journal of Functional Programming **2** (1992) 245–271
15. Cherem, S., Rugina, R.: A practical escape and effect analysis for building lightweight method summaries. In: 16th International Conference on Compiler Construction (CC 2007), Braga, Portugal (2007)
16. Barnett, M., Naumann, D.A., Schulte, W., Sun, Q.: 99.44In: ECOOP Workshop on Formal Techniques for Java-like Programs (FTfJP. (2004) 11–19
17. Barnett, M., Fändrich, M., Garbervetsky, D., Logozzo, F.: Annotations for (more) precise points-to analysis. In: IWACO 2007: ECOOP International Workshop on Aliasing, Confinement and Ownership in Object-Oriented Programming. (2007)
18. Ma, K.K., Foster, J.S.: Inferring aliasing and encapsulation properties for java. In: OOPSLA '07: Proceedings of the 22nd annual ACM SIGPLAN conference On Object Oriented Programming Systems and Applications, New York, NY, USA, ACM (2007) 423–440
19. Porat, S., Biberstein, M., Koved, L., Mendelson, B.: Automatic detection of immutable fields in java. In: CASCON '00: Proceedings of the 2000 conference of the Centre for Advanced Studies on Collaborative research, IBM Press (2000) 10
20. Unkel, C., Lam, M.S.: Automatic inference of stationary fields: a generalization of Java's final fields. In Necula, G.C., Wadler, P., eds.: POPL, ACM (2008) 183–195