Autom Softw Eng (2009) 16: 145–192
DOI 10.1007/s10515-008-0043-7

Parameter reference immutability: formal definition, inference tool, and comparison

Shay Artzi · Adam Kieżun · Jaime Quinonez · Michael D. Ernst

Received: 15 October 2008 / Accepted: 20 October 2008 / Published online: 3 December 2008

Abstract Knowing which method parameters may be mutated during a method's execution is useful for many software engineering tasks. A parameter reference is *immutable* if it cannot be used to modify the state of its referent object during the method's execution. We formally define this notion, in a core object-oriented language. Having the formal definition enables determining correctness and accuracy of tools approximating this definition and unbiased comparison of analyses and tools that approximate similar definitions.

We present Pidasa, a tool for classifying parameter reference immutability. Pidasa combines several lightweight, scalable analyses in stages, with each stage refining the overall result. The resulting analysis is scalable and combines the strengths of its component analyses. As one of the component analyses, we present a novel dynamic mutability analysis and show how its results can be improved by random input generation. Experimental results on programs of up to 185 kLOC show that, compared to previous approaches, Pidasa increases both run-time performance and overall accuracy of immutability inference.

Keywords Readonly · Reference immutability · Definition · Combined analysis

S. Artzi (✉) · A. Kieżun · J. Quinonez · M.D. Ernst
MIT Computer Science and Artificial Intelligence Laboratory, 32 Vassar st., Cambridge, MA 02142, USA
e-mail: artzi@csail.mit.edu

A. Kieżun
e-mail: akiezun@csail.mit.edu

J. Quinonez
e-mail: jaimeq@csail.mit.edu

M.D. Ernst
e-mail: mernst@csail.mit.edu

1 Introduction

Knowing which method parameters are accessed in a read-only way, and which ones may be mutated, is useful in many software engineering tasks, such as modeling (Burdy et al. 2005), verification (Cataño and Huisman 2003; Tkachuk and Dwyer 2003), compiler optimizations (Clausen 1997; Sălcianu 2006), program transformations such as refactoring (Fowler 2000), test input generation (Artzi et al. 2006), regression oracle creation (Mariani and Pezzè 2005; Xie 2006), invariant detection (Ernst et al. 2001), specification mining (Dallmeier et al. 2006), program slicing (Weiser 1984), and program comprehension (Demsky and Rinard 2002; Dolado et al. 2003).

Unintended mutation is a common source of errors in programs. Such errors can be difficult to discover and debug, because their symptoms often appear far from the location of the error, in time or in space. The problem of unintended mutation is exacerbated by the difficulty of expressing the programmer's design intent. A type system or a program annotation system offers an approximation to the truth regarding whether particular mutations can occur. The advantage of such approximations is that they are checkable. However, every statically checkable approximation prevents certain references, that are never used for mutation, from being so annotated because the proof of that fact is beyond the capability of the program analysis. A trivial example is that whether a given mutating statement can be executed is undecidable, but the problem is not a theoretical one and arises frequently in practice (Birka and Ernst 2004; Tschantz and Ernst 2005; Zibin et al. 2007; Papi et al. 2008).

Informally, reference immutability guarantees that a given reference is not used to modify its referent. (An immutable reference is sometimes known as a "read-only reference".) This definition has been the basis of much previous research (Birka and Ernst 2004; Boyland et al. 2001; Dietl and Müller 2005; Hogg 1991; Kniesel and Theisen 2001; Sălcianu and Rinard 2005; Skoglund and Wrigstad 2001; Tschantz 2006; Zibin et al. 2007). The informal description is intuitively understandable, but because it is vague, different people have different intuitions. For example, "used" might refer to the time the reference is in scope (Artzi et al. 2007; Sălcianu and Rinard 2005) or the entire execution (Boyland et al. 2001; Tschantz 2006). The referent might be considered to be only a single object (Stroustrup 2000; Boyland et al. 2001), or also all objects (transitively) referred to by subfields (Tschantz and Ernst 2005; Zibin et al. 2007; Kniesel and Theisen 2001). Modification might refer to the referent's concrete state (Boyland et al. 2001; Sălcianu and Rinard 2005; Artzi et al. 2007) or to its abstract state (Tschantz 2006; Zibin et al. 2007). Modifications via aliases may be ignored (Sălcianu and Rinard 2005) or counted as a mutation (Zibin et al. 2007). (See Sect. 8.2 for further discussion of related work.)

A precise definition of parameter reference mutability is a prerequisite to understanding, evaluating, or verifying an algorithm or tool. Some research (Kniesel and Theisen 2001; Noble et al. 1998; Skoglund and Wrigstad 2001; Dietl and Müller 2005) informally describes reference mutability but does not define the concept precisely. Capabilities (Boyland et al. 2001), Javari (Tschantz 2006), and IGJ (Zibin et al. 2007) use type/annotation systems to formally define different variants of reference immutability. Such formal definitions (e.g., type rules) approximate what mutations can actually occur: some references, that cannot be used to perform mutation, are mutable according to the definition. By contrast, our definition is precise: it does not depend on any particular implementation or approximation. Another difference is that the previous definitions considered mutations in the entire program execution when determining method parameter mutability. In our definition, which is inspired by (Sălcianu and Rinard 2005; Artzi et al. 2007), only mutations during the execution of the containing method are counted when determining parameter mutability. However, our definition easily extends to the definitions used by Javari, Capabilities, and IGJ (Sect. 2.3 discusses the extension).

The lack of a precise, formal definition of reference (im)mutability has made it difficult to determine both *whether* a program analysis tool correctly approximates the ideal, and *how closely* it does so. In addition, previous approaches for computing mutability suffered from scalability problems (static systems) and accuracy problems (dynamic systems). Our research addresses these issues. We formally define parameter (im)mutability; we present Pidasa, an approach to detecting mutability that combines the strengths of both static and dynamic analysis resulting in a system with better run-time performance and accuracy. Finally, we qualitatively and quantitatively compare both our definition and our implementation to existing immutability inference systems.

The Pidasa approach to mutability detection combines the strengths of static and dynamic analyses. Previous work has employed static analysis techniques to detect *immutable* parameters. Computing accurate static analysis approximations threatens scalability, and imprecise approximations can lead to weak results. Dynamic analyses offer an attractive complement to static approaches, both in not using approximations and in detecting *mutable* parameters. In our approach, different analyses are combined in stages, forming a "pipeline", with each stage refining the overall result.

This paper focuses on reference immutability of parameters. Parameter immutability is an important and useful special case of reference immutability, and it is supported by tools against which we can compare our definition. Our definition extends in a straightforward way to general reference immutability.

Parameter reference immutability can be computed per method implementation. A type system for reference immutability may enforce that method overriding preserve mutability of parameters, and type annotations can be easily computed from the per-implementation reference immutability information.

Contributions. This paper makes the following contributions:

- A formalization of a widely-used definition of parameter reference immutability that does not depend on a type or annotation system.
- The first staged analysis approach for discovering parameter mutability. Our staged approach is unusual in that it combines static and dynamic stages and it explicitly represents analysis incompleteness. The framework is sound, but an unsound analysis may be used as a component, and we examine the tradeoffs involved in such a choice.
- Mutability analyses. Our novel dynamic analysis scales well, yields accurate results (it has a sound mode as well as optional heuristics), and complements other

immutability analyses. We extend the dynamic analysis with random input generation, which improves the analysis results by increasing code coverage. We also explore a new point in the space of static techniques with a simple but effective static analysis.

- Evaluation. We have implemented our framework and analyses for Java, in a tool Pidasa. We performed two kinds of experiments.

  The first kind of experiments, investigates the costs and benefits of various sound and unsound techniques, including both Pidasa and that of Sălcianu and Rinard (2005). Our results show that a well-designed collection of fast, simple analyses can outperform a sophisticated analysis in both run-time performance and accuracy.

  The second kind of experiments demonstrates that the our formal definition is useful in exposing similarities and differences between approaches to immutability. We find that the evaluated analyses (Pidasa, JPPA (Sălcianu and Rinard 2005), Javari (Birka and Ernst 2004; Tschantz and Ernst 2005), and JQual (Greenfieldboyce and Foster 2007)) produce results that are very close to our formal definition, which confirms the need for a common formal base.

Outline. The remainder of this paper is organized as follows. Section 2 formally defines parameter reference immutability and illustrates it on an example (Appendix contains a longer example). Section 3 presents our staged mutability analysis framework. Sections 4 and 5 describe the new dynamic and static mutability analyses that we developed as components in the staged analysis. Section 6 experimentally evaluates various instantiations of the staged analysis framework. Section 7 experimentally compares results computed by the Pidasa inference tool and other existing tools to the formal definition. Section 8 surveys related work, and Sect. 9 concludes.

2 Parameter mutability definition

Informally, an object pointed by an immutable parameter reference cannot be changed. However, different interpretations of what is considered a change exist. For instance, the change might be to the set of reachable references from the parameter (deep), or to the references immediately reachable from the parameter (shallow). The change might happen during the method call, or after the method call. A change might happen through an aliasing reference, without even referring to the parameter in the method (reference vs. object immutability). Thus, a formal definition of parameter reference mutability is non-trivial.

Reference immutability differs from object immutability, which states that a given object cannot be mutated through any reference whatsoever. Both immutability variants have their own benefits. Reference immutability is more useful for specifying that a procedure may not modify its arguments, even if the caller reserves the right to do so before or after the procedure call. Object immutability is more useful, for example, when optimizing a program to store constant data in a shared section or read-only memory. Reference immutability is more fine-grained—reference immutability may be combined with an alias and escape analysis to infer object immutability (Birka and Ernst 2004; Sălcianu and Rinard 2005), but not vice-versa.

Section 2.1 informally defines parameter mutability. Section 2.2 demonstrates the intuition on an example. Section 2.3 formalizes parameter mutability. Section 2.4 and Appendix give examples of the formal definition.

2.1 Informal definition

Parameter p of method m is reference-*mutable* if there exists any execution of m during which p is *used* to mutate the state of the object pointed to by p. Parameter p is said to be *used* in a mutation, if the left hand side of the mutating assignment was p or was obtained from p via a series of field accesses and copy operations during the given execution. (Array accesses are analogous. The index expression is not considered to be used; that is, `a[b.f].f = 0` is not a mutation of `b`.) The mutation may occur in m itself or in any method that m transitively calls. The state of an object o is the part of the heap that is reachable from o by following references, and includes the values of reachable primitive fields. Thus, reference immutability is deep—it covers the entire abstract state of an object, which includes the fields of objects reachable from the object.

If no such execution (in all possible well-typed invocations of the method) exists, the parameter p is reference-*immutable*.

For example, in the following method:

```
void f(C c) {
D d = c.d;
E e = d.e;
e.f = null;
}
```

parameter `c` is used in the mutation in the last statement since the statement is equivalent to `c.d.e.f = null`.

The definition presented in this section, as well as its formalism in Sect. 2.3, is perfectly precise, but it is not computable, due to the quantification over all possible executions. Any tool can only infer an approximation of this definition of reference-mutability. (By contrast, some other attempts at defining mutability are based on a computable algorithm, but that does not capture the actual behavior of the program.) Section 7 compares several mutability inference tools, including our Pidasa tool (Sect. 3), to the definition.

2.2 Immutability classification example

In the code in Fig. 1, parameters `p1-p5` are reference-*mutable*, since there exists an execution of their declaring method such that the object pointed to by the parameter reference is modified *via the reference*. Parameters `p6` and `p7` are reference-*immutable*.

Mutable parameters:

- `p1` may be directly modified in `modifyParam1` (line 8).
- `p2` is passed to `modifyParam1`, in which it may be mutated.

```
 1  class C {
 2     public C next;
 3  }
 4
 5  class Main {
 6     void modifyParam1(C p1, boolean doIt) {
 7        if (doIt) {
 8           p1.next = null;
 9        }
10     }
11
12     void modifyParam1Indirectly(C p2, boolean doIt) {
13        modifyParam1(p2, doIt);
14     }
15
16     void modifyAll(C p3, C p4, C p5, boolean doIt) {
17        C c = p3.next;
18        p4.next = p5;
19        c.next = null;
20        modifyParam1Indirectly(p3, doIt);
21     }
22
23     void doNotModifyAnyParam(C p6) {
24        if (p6.next == null)
25           System.out.println("p6.next is null");
26     }
27     void doNotModifyAnyParam2(C p7) {
28        doNotModifyAnyParam(p7);
29     }
30  }
```

Fig. 1 Example code that is used to illustrates parameter immutability. All non-primitive parameters other than `p6` and `p7` are *mutable*

- `p3` is *mutable* because line 19 modifies `p3.next.next`.
- `p4` is directly modified in `modifyAll` (line 18), because the mutation occurs via reference `p4`.
- `p5` is *mutable* because the state of the object passed to `p5` can get modified on line 19 via `p5`. This can happen because `p4` and `p3` might be aliased: for example, in the call `modifyAll(x2,x2,x1,false)`. In this case, the reference to `p5` is copied into `c` and then used to perform a modification on line 19. Neither `p3` nor `p5` is considered reference mutable as a result of line 18 even though they might be aliased to `p4` at run time.

Immutable parameters:

- `p6` and `p7` are reference-*immutable*. No execution of either method `doNotModifyAnyParam` or `doNotModifyAnyParam2` can modify an object passed to `p6` or `p7`.

2.3 Formal definition

We present our formal parameter mutability definition in three steps. The first step defines a core object-oriented language (Sect. 2.3.1). The second step defines the evaluation rules (Sect. 2.3.1) (i.e., the operational semantics) that compute whether a parameter reference is used in a mutation. The third step (Sect. 2.3.3) formally defines parameter reference-mutability (Definition 1) by adding universal quantification over all possible well-typed invocations.

Our definition, described in this section, is quantified over all well-typed expressions. As a consequence, our definition does not depend on any specific execution, although it is defined using operational semantics in Sect. 2.3.1.

2.3.1 Core language

We define reference-mutability in the context of Mut Lightweight Java (MLJ), an augmented version of Lightweight Java (LJ) (Tschantz 2006), a core calculus for Java with mutation. LJ extends Featherweight Java (FJ) (Igarashi et al. 2001) to include field assignment with the `set` construct. To support the field assignment, LJ adds the store (S) that records a mapping from each object to its class and field record. LJ uses the field record ($\mathcal{F}$) to record a mapping from each field to the values they contain.

The syntax of MLJ is presented in Fig. 2. The syntax of MLJ is identical to the syntax of LJ without the parts of LJ that are related to generics and wild-cards. Those parts are irrelevant to reference immutability. Control flow constructs such as `if` only propagate, never introduce, mutation, so adding them to MLJ would only clutter the presentation. Similarly, arrays offer no more insight than field accesses. Other features, such as local variables, can be emulated in MLJ. The definition ignores reflection, but some tools handle it, including the dynamic analyses of our Pidasa tool (Sect. 3).

Mut Lightweight Java allows us to define mutability in the context of the underlying structure of Java, without being overwhelmed by the complexity of the full language. Since the definition is quantified over all possible executions, it does not depend on any specific execution, even though it uses operational semantics mechanism.

To avoid clutter, in the description below we present the operational semantics (evaluation) rules, but omit the typing rules, which are irrelevant to reference immutability.

Figure 3 shows the notations of MLJ that are used in the operational semantics, with the changes from LJ shown shaded. A value $v = (\!| o, \mathcal{M} |\!)$ in MLJ is a pair containing the corresponding value o from LJ, and a mutability set $\mathcal{M}$. If the value v can be modified, then the formal parameters in $(\!| o, \mathcal{M} |\!)$ are parameter reference-mutable. The *ret* expression provides a hook for removing formal parameters from mutability

Fig. 2 Syntax of Mut Lightweight Java: same as Lightweight Java (Tschantz 2006) without generics and wild cards. $\overline{X}$ is a vector of X

$$\begin{array}{lll}
\texttt{Q} ::= \texttt{class C extends C}\,\{\overline{\texttt{C f}};\ \texttt{K}\ \overline{\texttt{M}}\} & & \textit{class declarations} \\
\texttt{K} ::= \texttt{C}(\overline{\texttt{C f}})\,\{\texttt{super}(\overline{\texttt{f}});\ \texttt{this}.\overline{\texttt{f}} = \overline{\texttt{f}};\} & & \textit{constructor declarations} \\
\texttt{M} ::= \texttt{C m}(\overline{\texttt{C x}})\{\ \texttt{return e};\ \} & & \textit{method declarations} \\
\texttt{e} ::= \texttt{x} & & \textit{expressions} \\
\quad\ \mid\ \texttt{e.f} & & \\
\quad\ \mid\ \texttt{e.m}(\overline{\texttt{e}}) & & \\
\quad\ \mid\ \texttt{new C}(\overline{\texttt{e}}) & & \\
\quad\ \mid\ \texttt{set e.f = e then e} & & \\
\texttt{C} \quad \textit{class names} & & \\
\texttt{m} \quad \textit{method names} & & \\
\texttt{f} \quad \textit{field names} & & \\
\texttt{x} \quad \textit{parameter names} & &
\end{array}$$

Fig. 3 Mut Lightweight Java notations used in the operational semantics (Fig. 4). Changes from Lightweight Java (Tschantz 2006) are shaded

r	::=	$ret\ \mathtt{m}^i\ \mathtt{e}$	*return expressions*
$\mathcal{M}$	::=	$\varnothing$	*mutability sets*
	|	$\{C.m^i.x, \ldots\}$	*parameters*
v	::=	$(\!\| o, \mathcal{M} \|\!)$	*values : pairs of object, mutability set*
S	::=	$\varnothing$	*stores*
	|	$S, o = (C, \mathcal{F})$	*object bindings*
$\mathcal{F}$	::=	$\varnothing$	*field records*
	|	$\mathcal{F}, \mathtt{f} = v$	*field bindings*
Ω	::=	$\varnothing$	*mutability stores*
	|	$\{C.m.x, \ldots\}$	*mutable parameters*
o			*objects*
i			*natural numbers*

sets when a method exits. A *ret* expression cannot be written by a user, but is created during the evaluation of a program. The mutability store (Ω) contains the set of parameters that have been discovered to be reference-mutable.

2.3.2 Evaluation rules

In MLJ, evaluation maintains a mutability set Ω for each value v. A formal parameter $C.\mathtt{m}.\mathtt{x}$ is added to set $\mathcal{M}$ for a given value v if either v was the value bound to parameter $C.\mathtt{m}.\mathtt{x}$ or if v was a result of a dereference operation from a value that had $C.\mathtt{m}.\mathtt{x}$ in its mutability set. On exit from the ith invocation of $\mathtt{m}$, evaluation removes from all mutability sets, any parameters added on the corresponding method entry. A modification to a value causes the parameters in the value's mutability set to be classified as mutable.

Figure 4 shows the operational semantics rules for MLJ. Each reduction rule is a relationship, $\langle \mathtt{e}, S, \Omega \rangle \longrightarrow \langle \mathtt{e}', S', \Omega' \rangle$, where expression $\mathtt{e}$ with store S and mutability store Ω reduces to expression $\mathtt{e}'$ with store S' and mutability store Ω'. The changes from LJ, in computation and congruence rules, are shaded in Fig. 4. The congruence rules remain essentially unchanged between LJ and MLJ. We describe each computation rule, and the additional computation in MLJ.

[R-FIELD] This rule evaluates field accesses.
LJ: locates the value v_2 stored in the field and returns it.
MLJ: adds the mutability set $\mathcal{M}_{v_1}$, of the dereferenced value v_1, to the mutability set of v_2. The field access causes v_2 to be accessed from the parameters in $\mathcal{M}_{v_1}$, as well as the parameters that were previously used to obtain v_2 (i.e., the mutability set of v_2).

[R-INVK] This rule evaluates method invocations.
LJ: works in two stages. First, it finds the correct method $\mathtt{m}$ (using auxiliary function *mbody* which returns a pair $\overline{\mathtt{x}}.\mathtt{e}$ where $\overline{\mathtt{x}}$ are $\mathtt{m}$'s formal parameters and $\mathtt{e}$ is $\mathtt{m}$'s body). Second, it replaces the call with the body of $\mathtt{m}$ and replaces each formal parameter with the actual parameter.
MLJ: updates the mutability set of each formal parameter with the corresponding superscripted (with number of invocations) formal parameter, e.g., $m^i.x$. This rule also adds a `ret` expression call. The `ret` expression enables the [R-RET] rule to remove the same superscripted (with number of invocations) formal parameters from

Computation:

$$\frac{\mathcal{S}(o_1) = \langle \mathtt{C},\ \mathcal{F} \rangle \qquad \mathcal{F}(\mathtt{f}_i) = v_2 \qquad v_1 = (\!| o_1, \mathcal{M}_{v_1} |\!) \qquad v_2 = (\!| o_2, \mathcal{M}_{v_2} |\!)}{\langle v_1.\mathtt{f}_i,\ \mathcal{S}\ ,\ \Omega \rangle \longrightarrow \langle\ (\!| o_2, \mathcal{M}_{v_1} \cup \mathcal{M}_{v_2} |\!)\ ,\ \mathcal{S}\ ,\ \Omega \rangle} \qquad \text{[R-FIELD]}$$

$$\frac{\mathcal{S}(o) = \langle \mathtt{C},\ \mathcal{F} \rangle \quad \mathit{mbody}(\mathtt{m},\ \mathtt{C}) = \overline{\mathtt{x}}.\mathtt{e}_0 \qquad v = (\!| o, \mathcal{M}_v |\!) \qquad i = \mathit{invocations}(m) + 1 \qquad \overline{v} = (\!| \overline{o}, \mathcal{M}_{\overline{v}} |\!)}{\langle v.\mathtt{m}(\overline{v}),\ \mathcal{S}\ ,\ \Omega \rangle \longrightarrow \langle\ \mathit{ret}\ m^i\ [\ (\!|\ \overline{o}\ ,\overline{\mathcal{M}_v} \cup \mathtt{C.m}^i.\overline{\mathtt{x}} |\!)\ /\overline{\mathtt{x}},\ \ (\!|\ o\ ,\mathcal{M}_v \cup \{C.m^i.\mathtt{this}\} |\!)\ /\mathtt{this}]\mathtt{e}_0,\ \mathcal{S}\ ,\ \Omega \rangle} \qquad \text{[R-INVK]}$$

$$\frac{v = (\!| o, \mathcal{M}_v |\!)}{\langle \mathit{ret}\ \mathtt{m}^i\ v,\ \mathcal{S}\ ,\ \Omega\ \rangle \longrightarrow \langle (\!| o, \mathit{remove}(\mathtt{m}^i, \mathcal{M}_v) |\!),\ \mathit{removeAll}(\mathtt{m}^i, \mathcal{S})\ ,\ \Omega\ \rangle} \qquad \text{[R-RET]}$$

$$\frac{o \notin \mathtt{dom}(\mathcal{S}) \qquad \mathcal{F} = [\overline{\mathtt{f}} \mapsto \overline{v}]}{\langle \mathtt{new}\ \mathtt{C}(\overline{v}),\ \mathcal{S}\ ,\ \Omega \rangle \longrightarrow \langle\ (\!|\ o\ ,\varnothing |\!)\ ,\ \mathcal{S}[o \mapsto \langle \mathtt{C},\ \mathcal{F} \rangle]\ ,\ \Omega \rangle} \qquad \text{[R-NEW]}$$

$$\frac{\mathcal{S}(o_1) = \langle \mathtt{C},\ \mathcal{F} \rangle \qquad v_1 = (\!| o_1, \mathcal{M}_{v_1} |\!)}{\langle \mathtt{set}\ v_1.\mathtt{f}_i = v_2\ \mathtt{then}\ \mathtt{e}_b,\ \mathcal{S}\ ,\ \Omega \rangle \longrightarrow \langle \mathtt{e}_b,\ \mathcal{S}[o_1 \mapsto \langle \mathtt{C},\ \mathcal{F}[\mathtt{f}_i \mapsto v_2] \rangle]\ ,\ \Omega \cup \mathit{removeSuperscript}(\mathcal{M}_{v_1})\ \rangle} \qquad \text{[R-SET]}$$

Congruence:

$$\frac{\langle \mathtt{e}_0,\ \mathcal{S}\ ,\ \Omega \rangle \longrightarrow \langle \mathtt{e}'_0,\ \mathcal{S}'\ ,\ \Omega' \rangle}{\langle \mathtt{e}_0.\mathtt{f},\ \mathcal{S}\ ,\ \Omega \rangle \longrightarrow \langle \mathtt{e}'_0.\mathtt{f},\ \mathcal{S}'\ ,\ \Omega' \rangle} \qquad \text{[RC-FIELD]}$$

$$\frac{\langle \mathtt{e}_0,\ \mathcal{S}\ ,\ \Omega \rangle \longrightarrow \langle \mathtt{e}'_0,\ \mathcal{S}'\ ,\ \Omega' \rangle}{\langle \mathtt{e}_0.\mathtt{m}(\mathtt{e}),\ \mathcal{S}\ ,\ \Omega \rangle \longrightarrow \langle \mathtt{e}'_0.\mathtt{m}(\mathtt{e}),\ \mathcal{S}'\ ,\ \Omega' \rangle} \qquad \text{[RC-INVK-RECV]}$$

$$\frac{\langle \mathtt{e}_i,\ \mathcal{S}\ ,\ \Omega \rangle \longrightarrow \langle \mathtt{e}'_i,\ \mathcal{S}'\ ,\ \Omega' \rangle}{\langle v.\mathtt{m}(\overline{v}, \mathtt{e}_i, \overline{\mathtt{e}}),\ \mathcal{S}\ ,\ \Omega \rangle \longrightarrow \langle v.m(\overline{v}, \mathtt{e}'_i, \overline{\mathtt{e}}),\ \mathcal{S}'\ ,\ \Omega' \rangle} \qquad \text{[RC-INVK-ARG]}$$

$$\frac{\langle \mathtt{e}_i,\ \mathcal{S}\ ,\ \Omega \rangle \longrightarrow \langle \mathtt{e}'_i,\ \mathcal{S}'\ ,\ \Omega' \rangle}{\langle \mathtt{new}\ \mathtt{C}(\overline{v}, \mathtt{e}_i, \overline{\mathtt{e}}),\ \mathcal{S}\ ,\ \Omega \rangle \longrightarrow \langle \mathtt{new}\ \mathtt{C}(\overline{v}, \mathtt{e}'_i, \overline{\mathtt{e}}),\ \mathcal{S}'\ ,\ \Omega' \rangle} \qquad \text{[RC-NEW-ARG]}$$

$$\frac{\langle \mathtt{e}_0,\ \mathcal{S}\ ,\ \Omega \rangle \longrightarrow \langle \mathtt{e}'_0,\ \mathcal{S}'\ ,\ \Omega' \rangle}{\langle \mathtt{set}\ \mathtt{e}_0.f = \mathtt{e}_v\ \mathtt{then}\ \mathtt{e}_b,\ \mathcal{S}\ ,\ \Omega \rangle \longrightarrow \langle \mathtt{set}\ \mathtt{e}'_0.f = \mathtt{e}_v\ \mathtt{then}\ \mathtt{e}_b,\ \mathcal{S}'\ ,\ \Omega' \rangle} \qquad \text{[RC-SET-LHS]}$$

$$\frac{\langle \mathtt{e}_v,\ \mathcal{S}\ ,\ \Omega \rangle \longrightarrow \langle \mathtt{e}'_v,\ \mathcal{S}'\ ,\ \Omega' \rangle}{\langle \mathtt{set}\ v.f = \mathtt{e}_v\ \mathtt{then}\ \mathtt{e}_b,\ \mathcal{S}\ ,\ \Omega \rangle \longrightarrow \langle \mathtt{set}\ v.f = \mathtt{e}'_v\ \mathtt{then}\ \mathtt{e}_b,\ \mathcal{S}'\ ,\ \Omega' \rangle} \qquad \text{[RC-SET-RHS]}$$

$$\frac{\langle \mathtt{e}_v,\ \mathcal{S}\ ,\ \Omega \rangle \longrightarrow \langle \mathtt{e}'_v,\ \mathcal{S}'\ ,\ \Omega' \rangle}{\langle \mathit{ret}\ \mathtt{m}^i\ \mathtt{e}_v,\ \mathcal{S}\ ,\ \Omega \rangle \longrightarrow \langle \mathit{ret}\ \mathtt{m}^i\ \mathtt{e}'_v,\ \mathcal{S}'\ ,\ \Omega' \rangle} \qquad \text{[RC-RET]}$$

Fig. 4 Operational semantics (evaluation rules) for Mut Lightweight Java. Changes from Lightweight Java (Tschantz 2006) are shaded

all the relevant mutability sets when the method exits. This rule uses the auxiliary method *invocations* which returns the number of times the method `m` was invoked (also ensures that i is fresh).

[R-RET] This rule evaluates `ret` expressions and it does not exist in LJ.

MLJ: removes superscripted (see rule [R-INVK]) method formal parameters from all mutability sets. The function *remove* removes the parameters of $\mathtt{m}^i$ from the mutability set of the returned value v. The function *removeAll* does the same for all other values in the store.

[R-NEW] This rule evaluates object allocations.

LJ: creates a newly-allocated object.

MLJ: sets the mutability set of the newly-allocated object o to the empty mutability set.

[R-SET] This rule evaluates `set` expressions, i.e., field writes.

LJ: updates the value stored in a field for a given object.

MLJ: adds all the un-superscripted parameters (auxiliary method `removeSuperscript`) from the mutability set of the modified object to the mutability store Ω.

2.3.3 Reference-immutability definition

We define reference-immutable and reference-mutable parameters:

Definition 1 (Parameter reference mutability) Parameter `x` of method `m` of class C is *reference-mutable* if there exists an expression `e` such that $C.\mathtt{m}.\mathtt{x} \in \Omega$ at the end of the evaluation of `e`. Otherwise, p is *reference-immutable*.

To simplify the presentation in the rest of this section, we will refer to $C.m.x$ as x and to $C.m$ as m. Theorem 1 demonstrates the properties of Definition 1 and its equivalence to the intuitive informal definition. An auxiliary Definition 2 formalizes the notion of a series of dereferences needed for Theorem 1.

Let $\phi(i, m, e)$ be the evaluation of the ith invocation (dynamically) of m in e. We write it as ϕ when the parameters can be inferred from context.

Let $v_x^\phi = (\!| o, \mathcal{M}_v \cup \mathtt{m}^i.\mathtt{x} |\!)$ be the value passed to `x` at the start of ϕ (Rule [R-INVK]).

Definition 2 We inductively define $\delta_j^\phi(x)$, the executed dereference set of x during the first j evaluation steps of ϕ.

$$\begin{aligned}
&\delta_1^\phi(x) = \{v_x^\phi\}, \\
&\delta_j^\phi(x) = \delta_{j-1}^\phi(x) \cup (\!| o_2, \mathcal{M}_{v_1} \cup \mathcal{M}_{v_2} |\!) && \text{if the } j\text{th step is [R-FIELD]} \\
& && \text{and } v_1 \in \delta_{j-1}^\phi(x), \\
&\delta_j^\phi(x) = \delta_{j-1}^\phi(x) && \text{otherwise.}
\end{aligned}$$

Let $\delta^\phi(x)$ be the set of executed dereferences at the end of ϕ.

Theorem 1 *A parameter x is reference-mutable iff there exist v, i, m, e, j such that $v \in \delta_j^\phi(x)$ and* [R-SET] *is evaluated on $v.f$ during ϕ.*

Before proving Theorem 1 we present two auxiliary lemmas. Lemma 1 states that $m^i.x$ can exist in a mutability set only during ϕ. Lemma 2 states that if a value v is in the executed dereferences of a parameter x, then v's mutability set contains x.

Lemma 1 *Suppose that $v = (\!| o, \mathcal{M}_v |\!)$ and $m^i.x \in \mathcal{M}_v$. Such a value v can exist only during ϕ.*

Proof $m^i.x$ is created and added to a mutability set in rule [R-INVK] when starting the evaluation ϕ. [R-RET], which evaluates at the end of ϕ, removes all parameters of the form $m^i.x$ from all mutability sets. □

Lemma 2 presents the equality of the mutability set and the set of "used" references.

Lemma 2 *Let* $v = (\!| o, \mathcal{M}_v |\!)$. *Then* $m^i.\mathrm{x} \in \mathcal{M}_v$ *iff* $v \in \delta^\phi(x)$.

Proof $\longleftarrow$ Proof by induction on j where $v = v^j$ such that $v^j \in \delta^\phi_j(x) \wedge v^j \notin \delta^\phi_{j-1}(x)$.

If $j = 1$ then $v = v^\phi_x$ and $\mathtt{m}^i.x \in v^\phi_x$ by definition.

Otherwise, from Definition 2

$$v^j = (\!| o_2, \mathcal{M}_{v_1} \cup \mathcal{M}_{v_2} |\!) \wedge v_1 \in \delta^\phi_{j-1}(x).$$

By the induction assumption $\mathtt{m}^i.\mathrm{x} \in \mathcal{M}_{v_1}$. Thus $\mathtt{m}^i.\mathrm{x} \in \mathcal{M}_{v^j}$.

$\longrightarrow$ From $m^i.\mathrm{x} \in \mathcal{M}_v$ it follows that $\mathtt{m}^i.\mathrm{x}$ was added to $\mathcal{M}_v$ in either rule [R-INVK] or rule [R-FIELD] (the only two rules that augment a mutability set).

If $\mathtt{m}^i.\mathrm{x}$ was added to $\mathcal{M}_v$ in rule [R-INVK] then $v \in \delta^\phi_1(x)$.

If $\mathtt{m}^i.\mathrm{x}$ was added to $\mathcal{M}_v$ in rule [R-FIELD] then

$$v = (\!| o_2, \mathcal{M}_{v_1} \cup \mathcal{M}_{v_2} |\!) \quad \text{and} \quad \mathtt{m}^i.\mathrm{x} \in \mathcal{M}_{v_1} \quad \text{or} \quad \mathtt{m}^i.\mathrm{x} \in \mathcal{M}_{v_2}.$$

The rest of the proof follows by induction on the length of the executed [R-FIELD] from a value that contains $\mathtt{m}^i.\mathrm{x}$ in its mutability set. □

The proof of Theorem 1 follows directly from Lemmas 2 and 1.

Proof $\longrightarrow$ By Definition 1,

$$\exists \mathtt{e} \text{ such that } \mathtt{m.x} \in \Omega \text{ during the evaluation of } \mathtt{e}.$$

Since rule [R-SET] is the only rule creating a larger Ω it follows that

$$\exists v = (\!| o, \mathcal{M}_v |\!) \text{ such that } \mathtt{m}^i.\mathrm{x} \in \mathcal{M}_v \text{ and [R-SET] is evaluated on } v.f.$$

By Lemma 1, v can only exist during ϕ. The proof now follows directly from Lemma 2.

$\longleftarrow$ By $v \in \delta^\phi_j(x)$ and Lemma 2,

$$m^i.\mathrm{x} \in \mathcal{M}_v.$$

Since [R-SET] is evaluated on $v.f$ during ϕ, we get $\mathtt{m}^i.\mathrm{x} \in \Omega$. By Definition 1, x is mutable. □

Our definition accounts for mutations that occur during the dynamic extent of a method invocation. In some other mutability definitions (Boyland et al. 2001; Tschantz 2006; Zibin et al. 2007), if a parameter's value escapes to another context in which the value is later mutated, then the parameter is mutable in the original method (e.g., a getter method). Our framework accommodates both varieties of definition: removing rule [R-RET] converts Definition 1 into the other variant. In the revised definition, a reference is not removed from the mutability set when its method exits.

The rest of this paper uses *mutable* and *immutable* to refer to parameter reference mutable and parameter reference immutable, respectively.

2.4 Examples

We illustrate Definition 1 on two example functions in Figs. 5 and 6. Each example contains a program in MLJ and a table with the evaluation of MLJ operational semantics on the program. Each line in the table presents an expression to be evaluated, the state of the stores corresponding to the expression, and the next rule to apply. Similarly to Featherweight Java (Pierce 2002), we treat Obj as a distinguished class name whose definition does not appear in the class table. Obj has an empty constructor, no methods, and no fields. In addition, irrelevant (to mutability) details such as calls to `super` in constructors and the `extends` keyword are omitted.

Example (Classifying a modified receiver as mutable, Fig. 5) The parameter `this` of the function `m` in Fig. 5 is *mutable* because line 4 modifies `this`. The first two lines in the MLJ evaluation show the evaluation of the [R-NEW] rules. These rules creates the initial state of the store, and all the values created in them have empty mutability sets. The rule [R-INVK] is applied to the expression in step 3. Since it is the first invocation of the method m, parameters $\mathtt{m}^1$ and $m^1.this$ are added to

```
1   class B {
2     Obj f;
3     B(){ this.f = new Obj() }
4     Obj m(Obj p){ return set this.f = p then p }
5   }
6   new B().m(new Obj())
```

	expression	S	Ω	next rule
1	$new\ B().m(new\ Obj())$	$\varnothing$	$\varnothing$	[R-NEW]
2	$\mathtt{new}\ B().m(\llparenthesis o_1, \varnothing \rrparenthesis)$	$\{(o_1, \langle Obj, \varnothing\rangle)$	$\varnothing$	[R-NEW]
3	$\llparenthesis o_3, \varnothing \rrparenthesis.m(\llparenthesis o_1, \varnothing \rrparenthesis)$	$\{(o_1, \langle Obj, \varnothing\rangle), (o_2, \langle Obj, \varnothing\rangle),$ $(o_3, \langle B, \{f : \llparenthesis o_2, \varnothing \rrparenthesis\}\rangle)\}$	$\varnothing$	[R-INVK]
4	$ret\ \mathtt{m}^1\ set\ \llparenthesis o_3, \{m^1.this\} \rrparenthesis.f = \llparenthesis o_1, \{m^1.p\} \rrparenthesis\ then\ \llparenthesis o_1, \{m^1.p\} \rrparenthesis$	...	$\varnothing$	[R-SET]
5	$ret\ \mathtt{m}^1\ \llparenthesis o_1, \{m^1.p\} \rrparenthesis$	$\{(o_1, \langle Obj, \varnothing\rangle), (o_2, \langle Obj, \varnothing\rangle),$ $(o_3, \langle B, \{f : \llparenthesis o_1, \{m^1.p\} \rrparenthesis\}\rangle)\}$	$\{m.this\}$	[R-RET]
6	$\llparenthesis o_1, \varnothing \rrparenthesis$	$\{(o_1, \langle Obj, \varnothing\rangle), (o_2, \langle Obj, \varnothing\rangle),$ $(o_3, \langle B, \{f : \llparenthesis o_1, \varnothing \rrparenthesis\}\rangle)\}$	$\{m.this\}$	*Done*

Fig. 5 Classifying a mutable parameter. The Mut Lightweight Java program calls a method that updates a field in an object. The figure shows the evaluation of the program using the rules of Fig. 4. After the rule [R-SET] is applied (step 4), the parameter `m.this` is classified as mutable. Symbol ... means that a store does not change between evaluation steps

```
1   class B{
2     Obj f;
3     B(){ this.f = new Obj() }
4     Obj n(){ return this.m(this, this) }
5     B m(B p1, B p2){ return set p1.f = new Obj() then this }
6   }

7   new B().n()
```

	expression	S	Ω	next rule
1	$new\ B().n()$	$\emptyset$	$\emptyset$	[R-NEW]
2	$\llparenthesis o_2, \emptyset \rrparenthesis.n())$	$\{(o_1, \langle Obj, \emptyset\rangle), (o_2, \langle B, \{f : \llparenthesis o_1, \emptyset \rrparenthesis\}\rangle)\}$	$\emptyset$	[R-INVK]
3	$ret\ n^1\ \llparenthesis o_2, \{n^1.this\}\rrparenthesis.m(\llparenthesis o_2, \{n^1.this\}\rrparenthesis, \llparenthesis o_2, \{n^1.this\}\rrparenthesis)$	...	$\emptyset$	[R-INVK]
4	$ret\ n^1\ ret\ m^1\ set\ \llparenthesis o_2, \{n^1.this, m^1.p_1\}\rrparenthesis.f =$ $new\ Obj()\ then\ \llparenthesis o_2, \{n^1.this, m^1.this\}\rrparenthesis$	...	$\emptyset$	[R-NEW]
5	$ret\ n^1\ ret\ m^1\ set\ \llparenthesis o_2, \{n^1.this, m^1.p_1\}\rrparenthesis.f =$ $\llparenthesis o_3, \emptyset \rrparenthesis\ then\ \llparenthesis o_2, \{n^1.this, m^1.this\}\rrparenthesis,$	$\{(o_1, \langle Obj, \emptyset\rangle), (o_2, \langle B, \{f : \llparenthesis o_1, \emptyset \rrparenthesis\}\rangle)$ $(o_3, \langle Obj, \emptyset\rangle)\}$	$\emptyset$	[R-SET]
6	$ret\ n^1\ ret\ m^1\ \llparenthesis o_2, \{n^1.this, m^1.this\}\rrparenthesis$	...	$n.this,\ m.p_1$	[R-RET]
7	$ret\ n^1\ \llparenthesis o_2, \{n^1.this\}\rrparenthesis$	...	$n.this,\ m.p_1$	[R-RET]
8	$\llparenthesis o_2, \emptyset \rrparenthesis$	...	$n.this,\ m.p_1$	

Fig. 6 Classifying a mutable parameter while leaving another parameter, that points to a modified object, non mutable. Symbol ... means that a store does not change between evaluation steps. For the lack of local variables in the language, the utility method n sends the same object (this) as both parameters of the method m. The figure demonstrate how 2 parameters are classified as mutable, using the rules of Fig. 4. After the rule [R-SET] is applied to the expression in line 5, the parameters $n.this$ and $m.p_1$ are classified as mutable. Parameter $m.p_2$ can not be in the modified set of any parameter in the body of method m and thus there is no execution that will modify $m.p_2$, and so $m.p_2$ is defined as immutable

the mutability sets of the newly created values. The rule [R-SET] is applied to the expression in step 4. Since the object o_3 is modified, the parameters in its mutability set $\{m^1.this\}$ are added to the store of mutable parameters. Finally, the evaluation of the rule [R-RET] signals the exit from method m^1 and thus it removes all parameters superscripted by 1 from the mutability sets.

Example (Classifying aliased parameters, Fig. 6) In function `m` (Fig. 6), reference `p`$_1$ is *mutable* due to the modification in line 5. However, reference `p`$_2$ is reference-*immutable*—it is never used to make any modification to an object during the execution of `m`. The evaluation table in Fig. 6 demonstrates that parameter `p`$_2$ of function `m` is not reference-*mutable* in the call `m(o, o)` (i.e., even when parameters are aliased). When the execution finishes, $m.p_2 \notin \Omega$ and thus it is not classified as mutable during the evaluation of the call `new B().n()` or any other call for that matter.

Our definition is concerned with *reference* mutability, which, together with aliasing information, may be used to compute object mutability. In the example of function `m` in Fig. 6, the information that parameter `p`$_2$ is reference-immutable can be combined with information about `p`$_1$ and `p`$_2$ being aliased in the call `m(o, o)` to determine that, in that call, both objects may be modified.

Appendix contains an additional, more involved, example of an evaluation of MLJ.

3 Staged mutability analysis

The goal of any parameter-reference-mutability analysis is the classification of each method parameter (including the receiver) as either reference-mutable or reference-immutable.

In Pidasa approach, mutability analyses are combined in stages, forming a "pipeline". The input to the first stage is the initial classification of all parameters (typically, all *unknown*, though parameters declared in the standard libraries may be pre-classified). Each stage of the pipeline refines the results computed by the previous stage by classifying some *unknown* parameters. Once a parameter is classified as *mutable* or *immutable*, further stages do not change the classification. The output of the last stage is the final classification, in which some parameters may remain *unknown*.

Throughout this paper, two objects are *aliased* if the intersection of their states contains at least one non-primitive object (the same object is reachable from both of them).

Our dynamic and static analyses complement each other to classify parameters in Fig. 1 into *mutable* and *immutable*, in the following steps:

1. Initially, all parameters are *unknown*.
2. A flow-insensitive, intra-procedural static analysis classifies `p1`, `p4`, and `p5` as *mutable*. The analysis classifies `p6` as *immutable*—there is no direct mutation in the method and the parameter does not escape.
3. An inter-procedural static analysis propagates the current classification along the call-graph. It classifies `p2` as *mutable* since it is passed to an already known mutable parameter, `p1`. It also classifies parameter `p7` as *immutable* since it can only be passed to *immutable* parameters.
4. A dynamic analysis classification of `p3` depends on the given example execution. The dynamic analysis classifies `p3` as *mutable* if a method (similar to the `main` method below)

   ```
   void main() {
   modifyAll(x1, x2, x2, false);
   }
   ```

   is supplied or generated (see Sect. 4.4). Otherwise, the dynamic analysis classifies `p3` as *unknown*.

Our staged analysis correctly classifies all parameters in Fig. 1. However, this example poses difficulties for purely static or purely dynamic techniques. On the one hand, static techniques have difficulties correctly classifying `p3`. This is because, to avoid over-conservatism, static analyses often assume that on entry to a method all parameters are fully un-aliased, i.e., point to disjoint parts of the heap. In our example, this assumption may lead such analyses to incorrectly classify `p3` as *immutable* (in fact, Sălcianu uses a similar example to illustrate the unsoundness of his analysis, Sălcianu 2006, p. 78). On the other hand, dynamic analyses are limited to a specific execution and only consider modifications that happen during that execution. In our example, a purely dynamic technique may incorrectly classify `p2` as *immutable* if during the execution, `p2` is not modified.

Fig. 7 The static and dynamic component analyses used in our experiments. "✓*" means the algorithm is trivially sound, by never outputting the given classification. "✓†" means the algorithm is sound but our implementation is not

Analysis	Name	Section	i-sound	m-sound
dynamic	D	4.2	✓*	✓
dynamic heuristic **A**	DA	4.3	-	✓
dynamic heuristic **B**	DB	4.3	✓*	-
dynamic heuristic **C**	DC	4.3	✓*	-
dynamic heuristics **A**,**B**,**C**	DH	4.3	-	-
static intraprocedural	S	5.2	✓	-
static intraprocedural heuristic	SH	5.2.1	-	-
static interproc. propagation	P	5.3	✓	✓†
JPPA [52]	J	6.2	-	✓*
JPPA + `main`	JM	6.2	-	✓*
JPPA + `main` + heuristic	JMH	6.2	-	-

Combining mutability analyses can yield an analysis that has better accuracy than any of the components. For example, a static analysis can analyze an entire program and can prove the absence of a mutation, while a dynamic analysis can avoid analysis approximations and can prove the presence of a mutation.

Combining analyses in a pipeline also has performance benefits—a component analysis in a pipeline may ignore previously classified parameters. This can permit the use of techniques that would be too computationally expensive if applied to an entire program.

The problem of mutability inference is undecidable, so no analysis can be both sound and complete. An analysis is *i-sound* if it never classifies a *mutable* parameter as *immutable*. An analysis is *m-sound* if it never classifies an *immutable* parameter as *mutable*. An analysis is *complete* if it classifies every parameter as either *mutable* or *immutable*.

In our staged approach, analyses may explicitly represent their incompleteness using the *unknown* classification. Thus, an analysis result classifies parameters into three groups: *mutable*, *immutable*, and *unknown*. Previous work that used only two output classifications (Rountev and Ryder 2001; Rountev 2004) loses information by conflating parameters/methods that are known to be mutable with those where analysis approximations prevent definitive classification.

Some tasks, such as many compiler optimizations (Clausen 1997; Sălcianu 2006) require i-sound results (unless the results are treated as hints or are used online for only the current execution, Xu et al. 2007). Therefore, we have i-sound versions of our static and our dynamic analyses. However, other tasks, such as test input generation (Artzi et al. 2006), can benefit from more complete immutability classification while tolerating i-unsoundness. For this reason, we have devised several unsound approximations to increase the completeness (recall) of the analyses. Clients of the analysis can create an i-sound analysis by combining only i-sound components. Other clients, desiring more complete information, can use i-unsound components as well. Figure 7 summarizes the soundness characteristics of the analyses presented in this paper.

4 Dynamic mutability analysis

Our dynamic mutability analysis observes the program's execution and classifies as *mutable* those method parameters that are used to mutate objects. Our analysis does not implement the formal rules of Definition 1. It implements a simpler version of the formal rules that is designed for optimization.

The algorithm is m-sound: it classifies a parameter as *mutable* only when the parameter is mutated. The algorithm is also i-sound: it classifies all remaining parameters as *unknown*. Section 4.1 gives the idea behind the algorithm, and Sect. 4.2 describes an optimized implementation.

To improve the analysis results, we developed several heuristics (Sect. 4.3). Each heuristic carries a different risk of unsoundness. However, most are shown to be accurate in our experiments. The analysis has an iterative variation with random input generation (Sect. 4.4) that improves analysis precision and run-time.

4.1 Conceptual algorithm

The conceptual algorithm is based on Definition 1. The algorithm maintains the mutability set for each reference during the program execution. The mutability set is the set of all formal parameters (from any method invocation on the call stack) whose fields were directly or indirectly accessed to obtain the reference. When a reference `x` is side-effected (i.e., used in `x.f = y`), all formal parameters in `x`'s mutability set are classified as mutable. The algorithm implements the following set of data-flow rules based on the evaluation rules in Sect. 2.3.2.

1. On method entry, the algorithm adds each formal parameter (that is classified as *unknown*) to the parameter set of the corresponding actual parameter reference.
2. On method exit, the algorithm removes all parameters for the current invocation from the parameter sets of all references in the program.
3. Assignments, including pseudo-assignments for parameter passing and return values, propagate the parameter sets unchanged.
4. Field accesses also propagate the sets unchanged: the set of parameters for `x.f` is the same as that of `x`.
5. For a field write `x.f = v`, the algorithm classifies as *mutable* all parameters in the parameter set of `x`.

The next section presents an alternative algorithm that we implemented.

4.2 Optimized dynamic analysis algorithm

Maintaining mutability sets for all references, as required by the algorithm of Sect. 4.1, is computationally expensive. To improve performance, we developed an alternative algorithm that does not maintain mutability sets. The alternative algorithm is i-sound and m-sound, but is less complete—it classifies fewer parameters. In the alternative algorithm, parameter p of method m is classified as *mutable* if: (i) the transitive state of the object that p points to changes during the execution of m, and (ii) p is not aliased to any other parameter of m. Without part (ii), the algorithm would not be m-sound—*immutable* parameters that are aliased to a *mutable* parameter during the execution might be wrongly classified as *mutable*.

The example code in Fig. 8 illustrates the difference between the conceptual algorithm presented in Sect. 4.1 and the alternative algorithm presented in this section.

Fig. 8 Example code that is used to illustrates the limitation of the alternative algorithm in Sect. 4.2

```
class Main {
   void m1(C p1, C p2) {
     p1.x = null;
   }
   void m2(C p3, C p4) {
     p3.x = null;
     p4.x = null;
   }

   main(){
      o = new C();
      m1(o,o);
      m2(o,o);
   }
}
```

By definition 1 parameters $p1$, $p3$, $p4$ are mutable. The conceptual algorithm will classify them correctly when observing the execution of the method `main`. However, the alternative algorithm leaves all parameters as *unknown* since these parameters are aliased (in fact in this example they refer to the same object)—when the modification occurs. Note that the intra-procedural static analysis (Sect. 5.1) compensates for the incompleteness of the dynamic analysis in this case and correctly classifies `p1`, `p3`, `p4` as *mutable*.

The algorithm permits an efficient implementation: when method m is called during the program's execution, the analysis computes the set $\mathit{reach}(m, p)$ of objects that are transitively reachable from each parameter p via field references. When the program writes to a field in object o, the analysis finds all parameters p of methods that are currently on the call stack. For each such parameter p, if $o \in \mathit{reach}(m, p)$ and p is not aliased to other parameters of m, then the analysis classifies p as *mutable*. The algorithm checks aliasing by verifying emptiness of intersection of reachable sub-heaps (ignoring immutable objects, such as boxed primitives, which may be shared).

The implementation instruments the analyzed code at load time. The analysis works online, i.e., in tandem with the target program, without creating a trace file. Our implementation includes the following three optimizations, which together improve the run time by over $30\times$: (a) the analysis determines object reachability by maintaining and traversing its own data structure that mirrors the heap, which is faster than using reflection; (b) the analysis computes the set of reachable objects lazily, when a modification occurs; and (c) the analysis caches the set of objects transitively reachable from every object, invalidating it when one of the objects in the set is modified.

4.3 Dynamic analysis heuristics

The dynamic analysis algorithm described in Sects. 4.1 and 4.2 is m-sound—a parameter is classified as *mutable* only if it is modified during execution. Heuristics can improve the completeness, or recall (see Sect. 6), of the algorithm. The heuristics take advantage of the *absence* of parameter modifications and of the classification results computed by previous stages in the analysis pipeline. Using the heuristics may potentially introduce i-unsoundness or m-unsoundness to the analysis results, but in practice, they cause few misclassifications (see Sect. 6.3.5).

(**A**) *Classifying parameters as* immutable *at the end of the analysis*. This heuristic classifies as *immutable* all (*unknown*) parameters that satisfy conditions that are set by the client of the analysis. In our framework, the heuristic classifies as *immutable* a parameter p declared in method m if p was not modified, m was executed at least N times, and the executions achieved block coverage of at least $t\%$. Higher values of the threshold N or t increase i-soundness but decrease completeness.

The intuition behind this heuristic is that, if a method executed multiple times, and the executions covered most of the method, and the parameter was not modified during any of those executions, then the parameter may be *immutable*. This heuristic is m-sound but i-unsound. In our experiments, this heuristic greatly improved recall and was not a significant source of mistakes (Sect. 6.3.5).

(**B**) *Using current mutability classification*. This heuristic classifies a parameter as *mutable* if the object to which the parameter points is passed in a method invocation to a formal parameter that is already classified as *mutable* (by a previous or the current analysis). That is, the heuristic does not wait for the actual modification of the object but assumes that the object will be modified if it is passed to a *mutable* position. The heuristic improves analysis performance by not tracking the object in the new method invocation.

The intuition behind this heuristic is that if an object is passed as an argument to a parameter that is known to be *mutable*, then it is likely that the object will be modified during the call. The heuristic is i-sound but m-unsound. In our experiments, this heuristic improved recall and run time of the analysis and caused few misclassifications (see Sect. 6.3.5).

(**C**) *Classifying aliased mutated parameters*. This heuristic classifies a parameter p as *mutable* if the object that p points to is modified, regardless of whether the modification happened through an alias to p or through the reference p itself. For example, if parameters a and b happen to point to the same object o, and o is modified, then this heuristic will classify both a and b as *mutable*, even if it the modification is only done using the formal parameter's reference to a.

The heuristic is i-sound but m-unsound. In our experiments, using this heuristic improved the results in terms of recall, without causing any misclassifications.

4.4 Using randomly generated inputs

In this section we consider the use of randomly generated sequences of method calls as the required input for the dynamic analysis. Random generation can complement (or even replace) executions provided by a user. For instance, Pacheco et al. (2007) uses feedback-directed random generation to detect previously-unknown errors in widely used (and tested) libraries.

Using randomly-generated execution has benefits for a dynamic analysis. First, the user need not provide a sample execution. Second, random executions may explore parts of the program that the user-supplied executions do not reach. Third, each of the generated random inputs may be executed immediately—this allows the client of the analysis to stop generating inputs when the client is satisfied with the results of the analysis computed so far. Fourth, the client of the analysis may focus the input generator on methods with unclassified parameters.

Our generator gives a higher selection probability to methods with *unknown* parameters and methods that have not yet been executed by other dynamic analyses in the pipeline. Generation of random inputs is iterative. After the dynamic analysis has classified some parameters, it makes sense to propagate that information (see Sect. 5.3) and to re-focus random input generation on the remaining *unknown* parameters. Such re-focusing iterations continue as long as each iteration classifies at least 1% of the remaining *unknown* parameters (the threshold is user-settable).

By default, the number of generated method calls per iteration is *max*(*5000*, *#methodsInProgram*). The randomly generated inputs are executed in a safe way (Pacheco et al. 2007), using a Java security manager.

5 Static mutability analysis

This section describes a simple, scalable static mutability analysis. It consists of two phases: S, an intraprocedural analysis that classifies as (*im*)*mutable* parameters (never) affected by field writes within the procedure itself (Sect. 5.2), and P, an interprocedural analysis that propagates mutability information between method parameters (Sect. 5.3). P may be executed at any point in an analysis pipeline after S has been run, and may be run multiple times, interleaving with other analyses. S and P both rely on an intraprocedural pointer analysis that calculates the parameters pointed to by each local variable (Sect. 5.1).

5.1 Intraprocedural points-to analysis

To determine which parameters can be pointed to by each expression, we use an intraprocedural, context-insensitive, flow-insensitive, 1-level field-sensitive, points-to analysis. As a special case, the analysis is flow-sensitive on the code from the beginning of a method through the first backwards jump target, which includes the entire body of methods without loops. We are not aware of previous work that has explored this point in the design space, which we found to be both scalable and sufficiently precise.

The points-to analysis calculates, for each local variable l, a set $P_0(l)$ of parameters whose state l can point to directly and a set $P(l)$ of parameters whose state l can point to directly or transitively. (Without loss of generality, we assume three-address SSA form and consider only local variables.) The points-to analysis has "overestimate" and "underestimate" varieties; they differ in how method calls are treated (see below).

For each local variable l and parameter p, the analysis calculates a distance map $D(l, p)$ from the fields of object l to a non-negative integer or ∞. $D(l, p)(f)$ represents the number of dereferences that can be applied to l starting with a dereference of the field f to find an object pointed to (possibly indirectly) by p. Each map $D(l, p)$ is either strictly positive everywhere or is zero everywhere. As an example, suppose l directly references p or some object transitively pointed to by p; then $D(l, p)(f) = 0$ for all f. As another example, suppose $l.f.g.h = p.x$; then $D(l, p)(f) = 3$. The distance map D makes the analysis field-sensitive, but only at the first layer of dereferencing; we found this to be important in practice to provide satisfactory results.

The points-to analysis computes $D(l, p)$ via a fixpoint computation on each method. At the beginning of the computation, $D(p, p)(f) = 0$, and $D(l, p)(f) = \infty$ for all $l \neq p$. The dataflow rules are straightforward, so we give their flavor with a few examples:

- A field dereference $l_1 = l_2.f$ updates

$$\begin{aligned}\forall g : D(l_1, p)(g) &\leftarrow \min(D(l_1, p)(g), D(l_2, p)(f) - 1),\\ D(l_2, p)(f) &\leftarrow \min(D(l_2, p)(f), \min_g D(l_1, p)(g) + 1).\end{aligned}$$

- A field assignment $l_1.f = l_2$ updates

$$\begin{aligned}D(l_1, p)(f) &\leftarrow \min(D(l_1, p)(f), \min_g D(l_2, p)(g) + 1),\\ \forall g : D(l_2, p)(g) &\leftarrow \min(D(l_2, p)(g), D(l_1, p)(f) - 1).\end{aligned}$$

- Method calls are handled either by assuming they create no aliasing (creating an underestimate of the true points-to sets) or by assuming they might alias all of their parameters together (creating an overestimate). If an underestimate is desired, no values of $D(l, p)(f)$ are updated. For an overestimate, let S be the set of all locals used in the statement (including receiver and return value); for each $l \in S$ and each parameter p, set $D(l, p)(f) \leftarrow \min_{l' \in S} \min_{f'} D(l', p)(f')$.

After the computation reaches a fixpoint, it sets

$$\begin{aligned}P(l) &= \{p \mid \exists f : D(l, p)(f) \neq \infty\},\\ P_0(l) &= \{p \mid \forall f : D(l, p)(f) = 0\}.\end{aligned}$$

5.2 Intraprocedural phase: S

The static analysis S works in four steps. First, S performs the "overestimate" points-to analysis (Sect. 5.1). Second, the analysis marks as *mutable* some parameters that are currently marked as *unknown*: for each mutation $l_1.f = l_2$, the analysis marks all elements of $P_0(l_1)$ as *mutable*. Third, the analysis computes a "leaked set" L of locals, consisting of all arguments (including receivers) in all method invocations and all locals assigned to a static field (in a statement of the form `Global.field = local`). Fourth, if all the parameters of a method that are not already classified as *immutable* are *unknown* parameters that are not in the set $\bigcup_{l \in L} P(l)$ the analysis marks them as *immutable*.

S is i-sound and m-unsound. To avoid over-conservatism, S assumes that on the entry to the analyzed method all parameters are fully un-aliased, i.e., point to disjoint parts of the heap. This assumption may cause S to miss possible mutations due to aliased parameters; to maintain i-soundness, S never classifies a parameter as *immutable* unless all other parameters to the method can be classified as *immutable*.

For example, S does not detect any mutation to parameter `p5` of the method `modifyAll` in Fig. 1. Since other parameters of `modifyAll` (i.e., `p3` and `p4`) are classified as *mutable*, S conservatively leaves `p5` as *unknown*. In contrast, Sălcianu's static analysis JPPA (Sălcianu and Rinard 2005) incorrectly classifies `p5` as *immutable*.

The m-unsoundness of S is due to infeasible paths (e.g., unreachable code), flow-insensitivity, and the overestimation of the points-to analysis.

5.2.1 Intraprocedural analysis heuristic

We have also implemented a i-unsound heuristic SH that is like S, but it can classify parameters as *immutable* even when other parameters of the same method are not classified as *immutable*. In our experiments, this never caused a misclassification.

5.3 Interprocedural propagation phase: P

The interprocedural propagation phase P refines the current parameter classification by propagating both mutability and immutability information through the call graph. Given an i-sound input classification, the propagation algorithm is i-sound and m-sound. However, our implementation is i-sound and m-unsound.

Because propagation ignores the bodies of methods, the P phase is i-sound only if the method bodies have already been analyzed. It is intended to be run only after the S phase of Sect. 5.1 has already been run. However, it can be run multiple times (with other analyses in between).

Section 5.3.1 describes the binding multi-graph (BMG), and then Sect. 5.3.2 gives the propagation algorithm itself.

5.3.1 Binding multi-graph

The propagation uses a variant of the *binding multi-graph* (BMG) (Cooper and Kennedy 1988); our extension accounts for pointer data structures. Each node is a method parameter `m.p`. An edge from `m1.p1` to `m2.p2` exists iff `m1` calls `m2`, passing as parameter `p2` part of `p1`'s state (either `p1` or an object that may be transitively pointed-to by `p1`).

A BMG is created by generating a call-graph and translating each method call edge into a set of parameter dependency edges, using the sets $P(l)$ described in Sect. 5.1 to tell which parameters correspond to which locals.

The BMG creation algorithm is parameterized by a call-graph construction algorithm. Our experiments used CHA (Dean et al. 1995)—the simplest and least precise call-graph construction algorithm offered by Soot. In the future, we want to investigate using more precise but still scalable algorithms, such as RTA (Bacon and Sweeney 1996) (available in Soot, but containing bugs that prevented us from using it), or those proposed by Tip and Palsberg (2000) (not implemented in Soot).

The true BMG is not computable, because determining perfect aliasing and call information is undecidable. Our analysis uses an under-approximation (i.e., it contains a subset of edges of the ideal graph) and an over-approximation (i.e., it contains a superset of edges of the ideal graph) to the BMG as safe approximations for determining mutable and immutable parameters, respectively. One choice for the over-approximated BMG is the *fully-aliased* BMG, which is created with an overestimating points-to analysis which assumes that method calls introduce aliasings between *all* parameters. One choice for the under-approximated BMG is the *un-aliased* BMG, which is created with an underestimating points-to analysis which assumes that method calls introduce *no* aliasings between parameters. More precise approximations could be computed by a more complex points-to analysis.

To construct the under-approximation of the true BMG, propagation needs a call-graph that is an under-approximation of the real call-graph. However, most existing call-graph construction algorithms (Dean et al. 1995; Diwan et al. 1996; Bacon and Sweeney 1996; Tip and Palsberg 2000) create an over-approximation. Therefore, our implementation uses the same call-graph for building the un- and fully-aliased BMGs. Due to this approximation, our implementation of P is m-unsound. Actually, P is m-unsound even on the under-approximation of the BMG. For example, assume that m1.p1 is unknown, m2.p2 is mutable, and there is an edge between m1.p1 and m2.p2. It is possible that there is an execution of m2.p2 in which p2 is mutated, but for every execution that goes through m1, m2.p2 is immutable. In this case, the algorithm would incorrectly classify m1.p1 as mutable. In our experiments, this approximation caused several misclassifications of *immutable* parameters as *mutable* (see Sect. 6.3.1).

5.3.2 Propagation algorithm

Propagation refines the parameter classification in 2 phases.

The *mutability propagation* classifies as *mutable* all the *unknown* parameters that can reach in the under-approximated BMG (that is, can flow to in the program) a parameter that is classified as *mutable*. Using an over-approximation to the BMG would be unsound because spurious edges may lead propagation to incorrectly classify parameters as mutable.

The *immutability propagation* phase classifies additional parameters as *immutable*. This phase uses a fix-point computation: in each step, the analysis classifies as *immutable* all *unknown* parameters that have no *mutable* or *unknown* successors (callees) in the over-approximated BMG. Using an under-approximation to the BMG would be unsound because if an edge is missing in the BMG, the analysis may classify a parameter as *immutable* even though the parameter is really mutable. This is because the parameter may be missing, in the BMG, a *mutable* successor.

6 Evaluation

We implemented the combined static and dynamic analysis framework in a tool, Pidasa, and experimentally evaluated all sensible combinations (192 in all) of the mutability analyses described above, we compared the results with each other and with the correct classification of parameters as determined by Definition 1. Our results indicate that staged mutability analysis can be accurate, achieve better run-time performance, and are useful.

6.1 Methodology and measurements

We computed mutability for 6 open-source subject programs (see Fig. 9). When an example input was needed (e.g., for a dynamic analysis), we ran each subject program on a single input.

- **jolden**[1] is a benchmark suite of 10 small programs. As the example input, we used the `main` method and arguments that were included with the benchmarks. We included these programs primarily to permit comparison with Sălcianu's evaluation (Sălcianu and Rinard 2005).
- **sat4j**[2] is a SAT solver. We used a file with an unsatisfiable formula as the example input.
- **tinysql**[3] is a minimal SQL engine. We used the program's test suite as the example input.
- **htmlparser**[4] is a real-time parser for HTML. We used our research group's webpage as the example input.
- **ejc**[5] is the Eclipse Java compiler. We used one Java file as the example input.
- **daikon**[6] is an invariant detector. We used the StackAr test case from its distribution as the example input.

As the input to the first analysis in the pipeline, we used a pre-computed classification for all parameters in the Java standard libraries. Callbacks from the library code to the client code (e.g., `toString()`, `hashCode()`) were analyzed under the closed world assumption in which all of the subject programs were included. The pre-computed classification was created once, and reused many times in all the experiments. A benefit of using this classification is that it covers otherwise un-analyzable code, such as native calls.

We measured the results only for non-trivial parameters declared in the application. That is, we did not count parameters with a primitive, boxed primitive, or `String` type, nor parameters declared in external or JDK libraries.

To measure the accuracy of each mutability analysis, we determined the correct classification (*mutable* or *immutable*) for 8,885 parameters: all of jolden and ejc, and 4 randomly-selected classes from each of the other programs. To find the correct classification, we first ran every tool available to us (including our analysis pipelines, Sălcianu's tool, and the Javarifier (Tschantz 2006; Quinonez et al. 2008) type inference tool for Javari). Then, we manually verified the correct classification for every parameter where any two tool results differed, or where only one tool completed successfully. In addition, we verified an additional 200 parameters, chosen at random, where all tools agreed. We found no instances where the tools agreed on the mutability result, but the result was incorrect.

[1] http://www-ali.cs.umass.edu/DaCapo/benchmarks.html

[2] http://www.sat4j.org/

[3] http://sourceforge.net/projects/tinysql

[4] http://htmlparser.sourceforge.net/

[5] http://www.eclipse.org/

[6] http://pag.csail.mit.edu/daikon/

Fig. 9 Subject programs

program	size (LOC)	classes	parameters all	parameters non-trivial	parameters inspected
jolden	6,215	56	705	470	470
sat4j	15,081	122	1,499	1,136	118
tinysql	32,149	119	2,408	1,708	206
htmlparser	64,019	158	2,270	1,738	82
ejc	107,371	320	9,641	7,936	7,936
daikon	185,267	842	16,781	13,319	73
Total	410,102	1,617	33,304	26,307	8,885

Figure 10 and the tables in Sect. 6.3 present precision and recall results, computed as follows:

$$\begin{aligned}
\text{i-precision} &= \frac{ii}{ii + im},\\
\text{i-recall} &= \frac{ii}{ii + ui + mi},\\
\text{m-precision} &= \frac{mm}{mm + mi},\\
\text{m-recall} &= \frac{mm}{mm + um + im},
\end{aligned}$$

where ii is the number of immutable parameters that are correctly classified, and mi is the number of immutable parameters incorrectly classified as *mutable* (similarly, ui). Similarly, for mutable parameters, we have mm, im and um. i-precision is measure of soundness: it counts how often the analysis is correct when it classifies a parameter as *immutable*. i-recall is measure of completeness: it counts how many immutable parameters are marked as such by the analysis. m-precision and m-recall are similarly defined. An i-sound analysis has i-precision of 1.0, and an m-sound analysis has m-precision of 1.0. Ideally, both precision and recall should be 1.0, but this is not feasible: there is always a trade-off between analysis precision and recall.

6.2 Evaluated analyses

Our experiments evaluate pipelines composed of analyses described in Sect. 3. X-Y-Z denotes a staged analysis in which component analysis X is followed by component analysis Y and then by component analysis Z.

Our experiments use the following component analyses:

- S is the sound intraprocedural static analysis (Sect. 5.2).
- SH is the intraprocedural static analysis heuristic (Sect. 5.2.1).
- P is the interprocedural static propagation (Sect. 5.3).
- D is the dynamic analysis (Sect. 4), using the inputs of Sect. 6.1.
- DH is D, augmented with all the heuristics described in Sect. 4.3. DA, DB, and DC are D, augmented with just one of the heuristics.
- DRH is DH enhanced with random input generation (Sect. 4.4); likewise for DRA, etc.

Fig. 10 Mutability analyses on subject programs. Empty cells mean that the analysis aborted with an error. The abbreviations for the component analysis are described in Sect. 6.2

Prog.	Analysis	i-recall	i-precision	m-recall	m-precision
ejc	S-P-DRBC-P	0.781	1.000	0.915	0.956
	SH-P	0.777	1.000	0.904	0.971
	SH-P-DRH-P	0.928	0.996	0.907	0.971
	J	0.593	0.999	0.000	0.000
	JMH	0.734	0.998	0.691	0.941
	JMH-SH-P-DRH-P	0.939	0.997	0.944	0.951
jolden	S-P-DRBC-P	0.829	1.000	1.000	0.924
	SH-P	0.829	1.000	0.907	1.000
	SH-P-DRH-P	0.973	1.000	1.000	0.970
	J	0.894	1.000	0.000	0.000
	JMH	0.985	1.000	0.660	0.955
	JMH-SH-P-DRH-P	0.989	0.996	0.990	0.970
daikon	S-P-DRBC-P	0.705	1.000	0.931	0.844
	SH-P	0.636	1.000	0.931	0.844
	SH-P-DRH-P	0.750	1.000	0.931	0.844
	J	0.750	1.000	0.000	0.000
	JMH	-	-	-	-
	JMH-SH-P-DRH-P	-	-	-	-
tinysql	S-P-DRBC-P	0.965	1.000	0.655	0.655
	SH-P	0.955	1.000	0.667	0.800
	SH-P-DRH-P	0.980	0.995	0.667	0.667
	J	-	-	-	-
	JMH	-	-	-	-
	JMH-SH-P-DRH-P	-	-	-	-
sat4j	S-P-DRBC-P	0.763	1.000	0.968	0.968
	SH-P	0.763	1.000	0.968	0.968
	SH-P-DRH-P	0.854	0.980	0.968	0.968
	J	-	-	-	-
	JMH	-	-	-	-
	JMH-SH-P-DRH-P	-	-	-	-
htmlparser	S-P-DRBC-P	0.482	1.000	0.962	1.000
	SH-P	0.482	1.000	0.654	1.000
	SH-P-DRH-P	0.978	0.989	0.962	0.976
	J	-	-	-	-
	JMH	-	-	-	-
	JMH-SH-P-DRH-P	-	-	-	-

Additionally, we used JPPA (Sălcianu and Rinard 2005). Its informal mutability definition matches Definition 1, so JPPA is a natural comparison for the above analyses. We included the following additional analyses:

- J is Sălcianu and Rinard's state-of-the-art static analysis JPPA that never classifies parameters as *mutable*—only *immutable* and *unknown*.
- JM is J, augmented to use a `main` method that contains calls to all the public methods in the subject program (Rountev 2004); J only analyzes methods that are reachable from `main`.
- JMH is JM plus an m-unsound heuristic to classify as *mutable* any parameter for which J provides an explanation of a potential modification.

6.3 Results

We experimented with six programs and 192 different analysis pipelines. Figure 10 compares the accuracy of a selected set of mutability analyses among those with which we experimented. S-P-DRBC-P is the best-performing i-sound staged analysis. For clients that do not require i-soundness, the pipeline with the highest sum of precision and recall was SH-P-DRH-P. Compared to Sălcianu's (Sălcianu and Rinard 2005) state-of-the-art analysis J, the staged mutability analysis achieves equal or slightly worse i-precision, better i-recall, and much better m-recall and m-precision. SH-P-DRH-P also achieves better run-time performance (see Sect. 6.4). Sălcianu's analysis augmented with a heuristic for classifying mutable references, followed by our best stage analysis, JMH-SH-P-DRH-P, achieves the highest i-recall.

This section discusses the important observations that stem from the results of our experiments. Each sub-section discusses one observation that is supported by a table listing representative pipelines illustrating the observation. The tables in this section present results for ejc. Results for other programs were similar. However, for smaller programs all analyses did better and the differences in results were not as pronounced.

6.3.1 Interprocedural propagation

Running interprocedural propagation (P in the tables) is always beneficial, as the following table shows on representative pipelines.

Analysis	**i-recall**	**i-precision**	**m-recall**	**m-precision**
SH	0.563	1.000	0.299	0.998
SH-P	0.777	1.000	0.904	0.971
SH-P-DRH	0.922	0.996	0.906	0.971
SH-P-DRH-P	0.928	0.996	0.907	0.971
DRH	0.540	0.715	0.144	0.987
DRH-P	0.940	0.776	0.663	0.988

Propagation may decrease m-precision but, in our experiments, the decrease was never larger than 0.03 (not shown in the above table). In the experiments, propagation always increased all other statistics (sometimes significantly). For example, the table shows that propagation increased i-recall from 0.563 in SH to 0.777 in SH-P and it increased m-recall from 0.299 in SH to 0.904 in SH-P. Moreover, since almost all of the run-time cost of propagation lies in the call-graph construction, only the first execution incurs notable run-time cost on the analysis pipeline; subsequent executions of propagation are fast. Therefore, most pipelines presented in the sequel have P stages executed after each other analysis stage.

6.3.2 Combining static and dynamic analysis

Combining static and dynamic analysis in either order is helpful—the two types of analysis are complementary.

Analysis	i-recall	i-precision	m-recall	m-precision
SH-P	0.777	1.000	0.904	0.971
SH-P-DRH	0.922	0.996	0.906	0.971
SH-P-DRH-SH-P	0.928	0.996	0.907	0.971
DRH	0.540	0.715	0.144	0.987
DRH-SH-P	0.939	0.812	0.722	0.981
DRH-SH-P-DRH	0.943	0.813	0.722	0.981

For best results, the static stage should precede the dynamic stage. Pipeline SH-P-DRH, in which the static stage precedes the dynamic stage, achieved better i-precision and m-recall than DRH-SH-P, with marginally lower (by 0.01–0.02) i-recall and m-precision.

Repeating executions of static or dynamic analyses bring no substantial further improvement. For example, SH-P-DRH-SH-P (i.e., static-dynamic-static) achieves essentially the same results as SH-P-DRH (i.e., static-dynamic). Similarly, DRH-SH-P-DRH (i.e., dynamic-static-dynamic) only marginally improves i-recall over DRH-SH-P (i.e., dynamic-static).

6.3.3 *Comparing static stages*

In a staged mutability analysis, using a more complex static analysis brings little benefit. We experimented with replacing our lightweight interprocedural static analysis with J, Sălcianu's heavyweight static analysis.

Analysis	i-recall	i-precision	m-recall	m-precision
SH-P-DRH-P	0.928	0.996	0.907	0.971
J-DRH-P	0.973	0.787	0.664	0.998
JMH-DRH-P	0.939	0.922	0.878	0.949
JMH-SH-P-DRH-P	0.939	0.997	0.944	0.951

SH-P-DRH-P outperforms JMH-DRH-P with respect to 3 of 4 statistics, including i-precision (see Sect. 6.3.6). Combining the two static analyses improves recall—JMH-SH-P-DRH-P has better i-recall than SH-P-DRH-P and better m-recall than JMH-DRH-P. This shows that the two kinds of static analysis are complementary.

6.3.4 *Randomly generated inputs in dynamic analysis*

Using randomly generated inputs to the dynamic analysis (DRH) achieves better results than using a user-supplied execution (DH), at least for the relatively small user-supplied inputs (described in Sect. 6.1) with which we experimented. Although random generation can outperform or improve the results of a single user supplied input, Future work should evaluated whether random input generation can outperform and or augment the use of an exhaustive test suite.

Analysis	**i-recall**	**i-precision**	**m-recall**	**m-precision**
SH-P-DH	0.827	0.984	0.911	0.961
SH-P-DH-P-DRH	0.917	0.984	0.915	0.958
SH-P-DRH	0.922	0.996	0.906	0.971
SH-P-DRH-P-DH	0.932	0.983	0.912	0.970

Pipeline SH-P-DRH achieves better results than SH-P-DH with respect to i-precision, i-recall and m-precision (with lower m-recall). Using both kinds of executions can have different effects. For instance, SH-P-DH-P-DRH has better results than SH-P-DH, but SH-P-DRH-P-DH has a lower i-precision (due to i-unsoundness of heuristic **A**) with a small gain in i-recall and m-recall over SH-P-DRH.

The surprising finding that randomly generated code is as effective as using an example execution suggests that other dynamic analyses (e.g., race detection (Savage et al. 1997; O'Callahan and Choi 2003), invariant detection (Ernst et al. 2001), inference of abstract types (Guo 2006), and heap type inference (Polishchuk et al. 2007)) might also benefit from replacing example executions with random executions.

6.3.5 Dynamic analysis heuristics

By exhaustive evaluation, we determined that each of the heuristics is beneficial. A pipeline with DRH achieves notably higher i-recall and only slightly lower i-precision than a pipeline with DR (which uses no heuristics). This section indicates the unique contribution of each heuristic, by removing it from the full set (because some heuristics may have overlapping benefits). For consistency with other tables in this section, we present the results for ejc; however, the effects of heuristics were more pronounced on other benchmarks.

Heuristic **A** (evaluated by the DRBC line) has the greatest effect; removing this heuristic lowers i-recall (as compared to SH-P-DRH-P, which includes all heuristics.) However, because the heuristic is i-unsound, using it decreases i-precision, albeit only by 0.004. Heuristic **B** (the DRAC line) increases both i-recall and i-precision, and improves performance by 10%. Heuristic **C** (the DRAB line) is primarily a performance optimization. Including this heuristic results in a 30% performance improvement and a small increase in m-recall.

Analysis	**i-recall**	**i-precision**	**m-recall**	**m-precision**
SH-P-DR-P	0.777	1.000	0.905	0.971
SH-P-DRH-P	0.928	0.996	0.907	0.971
SH-P-DRBC-P	0.777	1.000	0.906	0.971
SH-P-DRAC-P	0.927	0.995	0.905	0.971
SH-P-DRAB-P	0.928	0.996	0.906	0.971

Heuristic **A** is parameterized by a coverage threshold t. Higher values of the threshold classify fewer parameters as *immutable*, increasing i-precision but decreasing i-recall. Figure 11 shows this relation. The heuristic is m-sound, so it has no effect on m-precision. The threshold value may reduce m-recall (if the analysis incorrectly classifies a *mutable* parameter), but, in our experiments, we have not observed this.

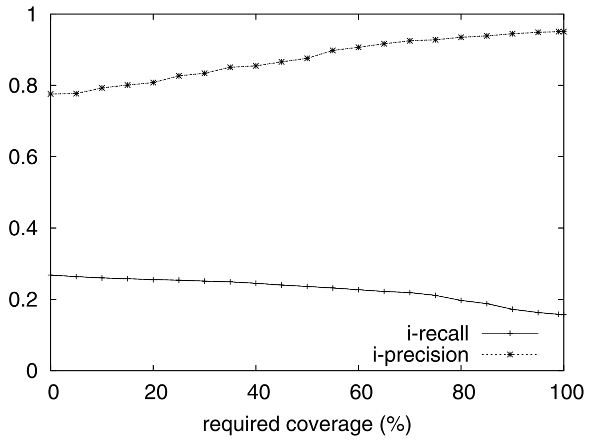

Fig. 11 Relation between i-precision, i-recall, and the coverage threshold in dynamic analysis heuristic **A**. The presented results are for the dynamic analysis DA on the ejc subject program

6.3.6 i-sound analysis pipelines

An i-sound mutability analysis never incorrectly classifies a parameter as *immutable*. All our component analyses have i-sound variations, and composing i-sound analyses yields an i-sound staged analysis.

Analysis	**i-recall**	**i-precision**	**m-recall**	**m-precision**
S	0.454	1.000	0.299	0.998
S-P	0.777	1.000	0.904	0.971
S-P-DRBC-P	0.777	1.000	0.906	0.971
S-P-DBC-P	0.777	1.000	0.912	0.959

S is the i-sound intra-procedural static analysis. Not surprisingly, the i-sound pipelines achieve lower i-recall than the i-unsound pipelines presented in Fig. 10; (Fig. 10 also presents the i-sound results for S-P-DRBC-P for all subjects.) For clients for whom i-soundness is critical, this may be an acceptable trade-off. In contrast to our analyses, J is not i-sound (Sălcianu 2006), although it did achieve very high i-precision (see Fig. 10).

6.4 Run-time performance

Figure 12 shows run times of analyses on daikon (185 kLOC, which is considerably larger than subject programs used in previous evaluations of mutability analyses Rountev and Ryder 2001; Rountev 2004; Sălcianu and Rinard 2005). The experiments were run using a quad-core AMD Opteron 64-bit 4×1.8 GHz machine with 4 GB of RAM, running Debian Linux and Sun HotSpot 64-bit Server VM 1.5.0_09-b01. Experiments were run on single thread and used one core. Staged mutability analysis achieves better run-time performance on large code-bases and runs in about a quarter the time of Sălcianu's analysis (J in Fig. 12).

Figure 12 shows that S-P (Sect. 6.3.6) runs, on daikon, an order of magnitude faster than J (or even better, if differences in call graph construction are discounted).

Fig. 12 The cumulative run time, and the time for the last component analysis in the pipeline, when analyzing the daikon subject program. Time is in seconds. Empty cells indicate that the analysis aborted with an error

Analysis	total time	last component
S	167	167
S-P	564	397
SH	167	167
SH-P	564	397
SH-P-DH	859	295
SH-P-DH-P	869	10
SH-P-DRH	1484	920
SH-P-DRH-P	1493	9
J	5586	5586
JM	-	-
JMH	-	-

Moreover, S-P is i-sound, while J is i-unsound. Finally, S-P has high m-recall and m-precision, while J has 0 m-recall and m-precision.

An optimized implementation of the P and DRH stages could run even faster. First, the major cost of propagation (P) is computing the call graph, which could be reused later in the same pipeline. J's RTA (Bacon and Sweeney 1996) call graph construction algorithm takes seconds, but our tool uses Soot, which takes two orders of magnitude longer to perform CHA (Dean et al. 1995) (a less precise algorithm). Use of a more optimized implementation could greatly reduce the cost of propagation. Second, the DRH step iterates many times, each time performing load-time instrumentation and other tasks that could be cached; without this repeated work, DRH can be much faster than DH. We estimate that these optimizations would save between 50% and 70% of the total SH-P-DRH-P time.

There is a respect in which our implementation is more optimized than J. J is a whole-program analysis that cannot take advantage of pre-computed mutability information for a library such as the JDK. By contrast, our analysis does so by default, and Fig. 12's numbers measure executions that use this pre-computed library mutability information. The number of annotated library methods is less than 10% of the number of methods in daikon.

6.5 Application: test input generation

Section 6.3 evaluated the accuracy of mutability analyses. This section evaluates their utility by measuring how much the computed immutability information helps a client analysis. The client analysis is Palulu (Artzi et al. 2006), a tool for generating regression tests. Palulu combines dynamic analysis with random testing to create legal test inputs, where each input is a sequence of method calls. Palulu works even for programs in which most random sequences of method calls are illegal, and it does not require a formal specification.

Palulu operates in two steps. First, Palulu infers a model that summarizes the sequences of method calls (and their input arguments) observed during the example execution. Palulu generates a model for each class. The model is a directed graph where each edge corresponds to a method call, and each node corresponds to an abstract state of an observed instance. The model contains every sequence of method calls that occurred in the example execution.

Second, Palulu uses the inferred models to guide a feedback-directed random input generator (Pacheco et al. 2007) in creating legal and behaviorally-diverse method

analysis	nodes	ratio	edges	ratio	time (s)	ratio
jolden + ejc + daikon						
no immutability	444,729	1.00	624,767	1.00	6,703	1.00
SH-P-DRH-P	124,601	3.57	201,327	3.10	4,271	1.56
J	131,425	3.83	210,354	2.97	4,626	1.44
htmlparser + tinysql + sat4j						
no immutability	48,529	1.00	68,402	1.00	215	1.00
SH-P-DRH-P	8,254	5.88	13,047	5.24	90	2.38
J	-	-	-	-	-	-

Fig. 13 Palulu (Artzi et al. 2006) model size and model generation time, when assisted by immutability classifications. The numbers are sums over indicated subject programs. Models with fewer nodes and edges are better. Also shown are improvement ratios over no immutability information (the "ratio" columns); larger ratios are better. Empty cells indicate that the analysis aborted with an error

sequences. The test generator uses the model to decide which method should be called and what constants can be used as parameters.

An intermediate method in the sequence of method calls (that is, not the final method call about which some property is asserted) is useful only if it produces a new abstract value that can be used later in the sequence. The new abstract value can come from the method's return value or from side-effecting some existing value. In the absence of other information, Palulu assumes that every method can side-effect every parameter. For example, it would generate both of these method sequences:

```
Date d = new Date();            Date d = new Date();
assert d.getTime() >= 0;        boolean b = d.equals(null);
                                assert d.getTime() >= 0;
```

because the `equals` method might side-effect its receiver.

The model can be pruned (without changing the state space it describes) by removing calls that do not mutate specific parameters, because non-mutating calls are not useful in constructing behaviorally-diverse test inputs. A smaller model permits a systematic test generator to explore the state space more quickly, or a random test generator to explore more of the state space. Per-parameter mutability information permits more model reduction than method-level purity information.

We ran Palulu on our subject programs using no immutability information, and using immutability information computed by J and by SH-P-DRH-P. Since exhaustive model exploration is generally infeasible, Palulu can use unsound immutability information to improve its likelihood of finding errors within a given time bound.

Using the mutability information, Palulu's first step generated smaller models. Using the smaller models, Palulu's second step generated tests that achieved better line and block coverage of the subject programs. Figure 13 shows the number of nodes and edges in the generated model graph, and the time Palulu took to generate the model (not counting the immutability analysis). Our experiment used J rather than JM, in part because JM runs on fewer programs, but primarily because JM's results would be no better than J. Palulu models include only methods called during execution, and J starts from the same main method. JM adds more analysis contexts, but doing so never changes a mutable parameter to immutable, which is the only way to improve Palulu model size.

program	size (LOC)	classes	parameters all ⊇	non-trivial ⊇	inspected ⊇	immutable
jolden	6,215	56	705	470	470	277
tinysql	32,149	119	2,408	1,708	206	200
htmlparser	64,019	158	2,270	1,738	82	56
ejc	107,371	320	9,641	7,936	7,936	3,679
Total	209,754	653	15,033	11,852	8,994	4,212

Fig. 14 Subject program characteristics. The **all** column lists the total number of parameters in the subject program. **non-trivial** parameters are not of primitive type or known immutable type (`java.lang.String` and boxed primitive types from package `java.lang`). Trivial parameters are always immutable. The **inspected** column lists the number of non-trivial parameters that we manually classified in each subject program (Sect. 6.1 describes how the parameters were selected for classification). Finally, the **immutable** column lists the number of inspected parameters that we manually determined to be immutable

7 Tool comparison

Definition 1 is useful for evaluating both annotation-based systems expressing reference immutability and inference-based systems inferring reference immutability. We experimentally evaluated four tools that each infer an approximation to a different mutability definition, by comparing the tool results to Definition 1. We used the Pidasa and JMH tools described in Sect. 6, as well as Javari (Birka and Ernst 2004; Tschantz and Ernst 2005) and JQual (Greenfieldboyce and Foster 2007), both of which are static inference tools.

Our results indicate that the evaluated tools produce results that are similar to the formal definition we propose. This finding supports the practical utility of a common formal definition. Our results also highlight the differences between the expressiveness of the evaluated tools, i.e., which immutable references each tool identifies.

Section 7.1 summarizes the four tools we compared against Definition 1. Section 7.2 *quantitatively* compares the formal definition of reference immutability to each of the four evaluated tools. Section 7.3 presents *qualitative* comparisons that illustrate the major conceptual differences between our definition and the definitions implemented by the four tools.

7.1 Tools compared

We evaluated four tools:

- Pidasa is the tool that implements the static and dynamic analysis described in Sect. 6. We used the two best combinations: S-P-DRBC-P and SH-P-DRH-P. The tool for combination S-P-DRBC-P, denoted Pidasa$_{snd}$, is i-sound. The tool for combination SH-P-DRH-P, denoted Pidasa$_{uns}$, is i-unsound, but achieves better recall. Pidasa may also leave parameters unclassified; in the evaluation, we conservatively treated such parameters as mutable.
- JMH is a static analysis tool for computing reference immutability (Sălcianu and Rinard 2005), described in Sect. 6.2.
- Javari is a type-annotation-based Java extension for specifying and enforcing reference immutability (Birka and Ernst 2004; Tschantz and Ernst 2005). The Javarifier

tool (Correa Jr. et al. 2007; Quinonez et al. 2008) inferred Javari annotations for the subject programs. To match the assumptions of the other tools, we ran Javarifier with the assumption that all fields should be considered part of the abstract state of an object, even though Javari and Javarifier support excluding specific fields from the abstract state of an object. Then, we conservatively changed all inferred `@Polyread` annotations (parametric polymorphism over mutability) to mutable (15 for jolden, 39 for htmlparser, 70 for tinysql, and 112 for eclipsec). We did not change the Javarifier output in any other way. The resulting annotated programs typecheck in accordance to the Javari definition.

- JQual is a static analysis tool for computing reference immutability (Greenfieldboyce and Foster 2007). We used the version downloaded from http://www.cs.umd.edu/projects/PL/jqual (10 December, 2007). We ran JQual in the context-insensitive, field-insensitive mode, because in any other mode it is unscalable (Greenfieldboyce and Foster 2007) and could process none of the subject programs. Thus, JQual did not infer its parametric polymorphism over mutability.

Regrettably, we did not find more programs on which all of the tools run to completion—bugs in the tools make them terminate with errors.

7.2 Quantitative comparison

Figure 15 presents the results of the quantitative analysis. In this section, we take precision and recall to mean i-precision and i-recall, respectively. Precision and recall of 1.0 would indicate perfect agreement with the definition. This evaluation uses precision and recall *not* as a measure of analysis quality (i.e., how well an analysis classifies parameters according to its own definition), but rather as a measure of how the mutability results of existing tools match our formal definition.

Fig. 15 Mutability analyses on subject programs. The **# params** columns list the numbers of parameters that we manually inspected for mutability (see Fig. 14). Precision and recall are computed only for the set of inspected parameters (Sect. 7.2 describes the details). Empty cells mean that the analysis aborted with an error

Prog	**# params**	**analysis**	**i-recall**	**i-precision**
jolden	470	Pidasa$_{uns}$	0.978	1.000
		Pidasa$_{snd}$	0.829	1.000
		JMH	0.985	1.000
		Javari	0.968	1.000
		JQual	0.696	0.965
tinysql	206	Pidasa$_{uns}$	0.980	0.995
		Pidasa$_{snd}$	0.965	1.000
		JMH	-	-
		Javari	0.956	1.000
		JQual	-	-
htmlparser	82	Pidasa$_{uns}$	0.978	0.989
		Pidasa$_{snd}$	0.482	1.000
		JMH	-	-
		Javari	0.756	1.000
		JQual	-	-
ejc	7936	Pidasa$_{uns}$	0.928	0.996
		Pidasa$_{snd}$	0.781	1.000
		JMH	0.734	0.998
		Javari	0.986	1.000
		JQual	-	-

All four evaluated analyses are flow-insensitive, so none has perfect recall. Pidasa$_{uns}$ achieves highest recall by sacrificing some precision. Javari and Pidasa$_{snd}$ aim for perfect precision and accept low recall. JMH also has nearly perfect precision (the imprecision is due to an implementation bug), but the conservative static analysis may lead to low recall (e.g., in the largest subject program, ejc). JQual also has nearly perfect precision, and its low recall is due to a difference in its definition of field mutability, discussed in Sect. 7.3.4.

7.3 Qualitative comparison

This Section presents a qualitative analysis describing more closely how each tool's results compare to Definition 1.

7.3.1 Pidasa

We examined all parameters on which Pidasa disagreed with the classification according to Definition 1. All differences can be attributed to imprecision in Pidasa's implementation. Thus, our examination showed that the definition of reference immutability used in Pidasa's technique, agrees with our formal definition.

Pidasa$_{uns}$ misclassified 23 mutable parameters as immutable due to an unsound heuristic in its dynamic component. The heuristic classifies a method parameter as immutable if no mutation occurred during the dynamic analysis phase and the block coverage of the method is above a certain threshold (Pidasa$_{uns}$ uses a 85% threshold).

Pidasa$_{uns}$ and Pidasa$_{snd}$ classified a parameter as `mutable` in several cases where the formal definition classifies the parameter as `immutable`, because no mutation can occur at run time. The discrepancies are due to the following analysis imprecision:

- Flow-insensitivity of the static analysis component. Flow-insensitive analysis does not consider the order of statements in the method. This contrasts with Definition 1, which is flow-sensitive. Flow-insensitivity may lead Pidasa to classify some types to be mutable even though no mutation will ever occur at run-time.

  For example, in this code:

  ```
  void foo(Date d) {
  d = new Date();
  d.setHour(12);
  }
  ```

  a flow-insensitive analysis would classify parameter `d` as mutable because the variable `d` can be mutated, even though the value passed as the parameter cannot be mutated.
- Dynamic component failing to generate inputs that exercise a method. This results in the parameters being left unclassified, which, in this evaluation, we conservatively treated as `mutable`.
- Call-graph approximations in the static component. Because the precise caller-callee relationship cannot always be computed in an object-oriented language, the static component of Pidasa uses a conservative approximation of the call-graph.

- Heuristics in the dynamic component. For example, in the code

  ```
  void m(Object p1, Object p2){
  p1.field = ...;
  }
  ```

  if, at run-time, `p1` and `p2` happen to be aliased (i.e., there is an overlap between the parts of the heap reachable from `p1` and `p2`), then the dynamic analysis heuristic incorrectly classifies `p2` as mutable.

7.3.2 JMH

We examined all parameters on which the JMH results disagreed with the classification according to the formal Definition 1. All differences are due to either bugs, or imprecision in the inference tool. Thus, our examination showed that the (implicit) definition used by JMH agrees with our proposed formal definition of reference immutability.

In ejc, JMH misclassified 5 mutable parameters as *immutable*. More precisely, JMH classified as *immutable* 5 parameters that are *mutable* according to the proposed formal definition. This misclassification is due to what seems to be an incorrect treatment of the native method `System.arraycopy()`. `System.arraycopy()` copies objects from a source array to a destination array. JMH seems to treat the destination array as immutable, which is incorrect.

JMH fails to correctly classify immutable parameters (i.e., the computed result disagrees with the proposed formal definition, with JMH reporting *mutable* when the parameter cannot be mutated at run time) due to:

- Failure to analyze package-private constructors that are called in static field initializers.
- Failure to analyze non-abstract methods in abstract classes.
- Conservative pointer analysis. For example, the method `CompilationResult.-computePriority()` in ejc calls the method `HashMap.get()` on a field. This receiver is classified as *mutable* by JMH, since `LinkedHashMap.get()` can mutate its receiver. However, the specific field can only be instantiated with `HashMap` (super-class of `LinkedHashMap`) for which `get()` is a side-effect-free method. Since JMH is a closed-world analysis, in theory it could compute the correct results in this case.
- Call-graph approximations similar to Pidasa (Sect. 7.3.1).
- Flow-insensitivity of the analysis similar to Pidasa (Sect. 7.3.1).

7.3.3 Javari

The Javarifier type inference tool infers the reference mutability of every parameter according to the Javari mutability definition. Figure 16 tabulates the differences between the formal Definition 1 and Javari. Javari is perfectly i-precise: the only kind of difference is when Javarifier inferred a mutable type that Definition 1 classifies as an immutable type. Javarifier disagreed with Definition 1 on the mutability of a parameter for three reasons:

Program	Inspected parameters	Arrays	Flow insensitivity	Dynamic scope	Total differences
jolden	470	12	3	0	15
tinysql	206	9	0	0	9
htmlparser	82	6	12	2	20
ejc	7,936	17	1	31	49

Fig. 16 Discrepancies where Javarifier classified a parameter reference as mutable, when it's classification by Definition 1 is immutable. For jolden, we manually analyzed all classes. For tinysql, htmlparser and ejc, we manually analyzed four classes selected at random. The columns Arrays, Flow insensitive, and Dynamic indicate difference sources for the discrepancies as explained in Sect. 7.3.3

Fig. 17 The formal definition considers parameter `newData` to be immutable because `update()` does not itself mutate either `data` or `newData`. Javari considers `newData` to be mutable because `update()` stores `newData` for possible later mutation

```
class Client {
  public static void main(String[] args) {
    Client c = new Client();
    Data d = new Data();
    c.update(d);
    c.mutateData();
  }

  private Data data;

  // Formal definition: parameter is immutable
  // Javari:            parameter is mutable
  public void update(Data newData) {
    data = newData;
  }

  public void mutateData() {
    data.mutate();
  }
}
```

Arrays. In Javari immutability is not deep with respect to arrays (or generics)—a client can control the mutability of each level explicitly. Therefore, Javari allows different mutabilities for arrays and their elements, while our definition does not. When Javarifier inferred an immutable array of mutable elements for a parameter, for the purpose of comparison to Definition 1, we treat Javarifier's classification as a mutable parameter. Definition 1, however, classified the parameter as immutable.

Flow-insensitivity. Javari is a flow-insensitive type system (like most type systems, e.g., Java's), and thus can misclassify immutable parameters as mutable, similarly to Pidasa (Sect. 7.3.1).

Dynamic scope. The formal definition accounts for mutations that occur during the execution of a method. By contrast, Javari marks a parameter as mutable if the actual argument may later be mutated, even after the method has exited, as a result of being passed into the method. For example, consider Fig. 17. As noted in Sect. 2.3.3, the formal definition can be modified to account for mutations even after a method has exited, simply by removing rule [R-RET] of Fig. 4. Future work could compare this modified formal definition to Javari.

Similarly, certain uses of parametric polymorphism force the Javari classification of some parameters to be mutable, while Definition 1 classifies them as immutable. Javari uses the `@PolyRead` type qualifier to express parametric polymorphism over

```
class DateScanner {
  private List<@ReadOnly Date> allDates;

  // Formal definition: receiver is mutable, parameter is immutable
  // Javari:            receiver is mutable, parameter is mutable
  void addDate(Date date) {
    Date scannedDate = scanDate(date);
    allDates.add(scannedDate);
  }

  // Formal definition: receiver is immutable, parameter is immutable
  // Javari:            receiver is polyread, parameter is polyread
  // The polyread keyword after the parameter list annotates the
  // receiver of the method as polyread.
  void @PolyRead Date scanDate(@PolyRead Date newDate) @PolyRead {
      ...
    }
  }
}
```

Fig. 18 A program in which the parameter of `addDate` is passed into method `scanDate`, which takes a `@Polyread` parameter and receiver. Javari type rules require that parameter `date` of `addDate` be declared as mutable

mutability type qualifiers.[7] Definition 1 is unable to express this polymorphism. In Javari, `@PolyRead` parameters cannot be mutated explicitly in a method, so Definition 1 classifies these parameters as immutable.

For example, in Fig. 18, the method `addDate` has a mutable receiver because it adds to the list `allDates`. Javari type-correctness requires the receiver types to be compatible when `addDate` calls `scanDate`. Since `addDate`'s receiver is mutable, it uses the version of `scanDate` with a mutable receiver. But that version has a mutable parameter, and so `date` is marked as mutable, even though `addDate` does not mutate it (and no later mutation can occur when the Date is extracted from the list, because its type is `@Readonly Date`).

7.3.4 JQual

In the jolden subject program, JQual inferred 7 parameters to be readonly that Definition 1 states are mutable because it cannot be mutated at run-time.

- In one case, JQual was more expressive when it inferred a parameter to be a readonly array of mutable objects, which the proposed formal definition classifies as a mutable array of mutable objects.
- In four cases, JQual incorrectly classified the receiver of a modifying method. Figure 19 demonstrates this problem when a mutating method is called on a reference that is passed through another accessor method.

[7] A method with `@PolyRead` parameters can be viewed as having two signatures that are resolved via overloading: in one version, all instances of `@PolyRead` are replaced by `@Mutable`, and in the other version, they are replaced by `@ReadOnly`. Almost every accessor method has its receiver and return type classified as `@PolyRead`. This means that when an accessor method is called on a `@Mutable` reference, the returned reference is `@Mutable`; when an accessor method is called on a `@ReadOnly` reference, the returned reference is `@ReadOnly`.

- In 2 cases, JQual incorrectly classified the receiver of a method because the method was a member of an inner class.

In 84 other cases, JQual inferred a parameter to be mutable that is immutable according to Definition 1. We manually analyzed ten of these parameters, selected at random. All the differences were due to a definitional difference with regard to field mutability. In JQual, every method that reads a mutable field must have a mutable receiver, even if the method only reads the field and does not mutate the program state. This restriction makes JQual overly conservative in declaring mutable references, but helps ensure soundness in its type system. Figure 20 illustrates the problem.

In order to resolve this issue, JQual would need to run in a field-sensitive and context-sensitive mode. Running in a field-sensitive mode, JQual infers a separate type for each instance of a field, rather than a single static type for the field. Running in a context-sensitive mode, JQual can treat methods as polymorphic over mutability, similar to Javari's `@Polyread`. (Even though JQual cannot express this polymorphism over mutability in its final output, the method can be treated as polymorphic over mutability during the inference step.) In a field-sensitive and context-sensitive mode, JQual's inference is similar to Javari's inference. Specifically, in the class in Fig. 20, the fact that the `data` field can be mutated in `resetFirst()` does not require reading the field in `first()` to be considered a mutation. As noted in Sect. 7.1,

Fig. 19 Example in which the proposed formal definition specifies that the receiver of `resetHead()` is mutable, while JQual classifies the receiver as immutable

```
public class Counter {
  int count;

  public Counter head() {
    return this;
  }

  // Formal definition: receiver mutable
  // JQual:             receiver immutable
  public void resetHead() {
    head().reset();
  }

  public void reset() {
    count = 0;
  }
}
```

Fig. 20 Example in which the proposed formal definition specifies that the receiver of `first()` is reference immutable, while JQual classifies the receiver as mutable. Method `first()` reads `this.data`, so JQual requires that `this` have at least the same mutability as `data` (which is mutable)

```
public class Info {
  private int[] data;

  // Formal definition: receiver immutable
  // JQual:             receiver mutable
  public int first() {
    return data[0];
  }

  public void resetFirst() {
    data[0] = 0;
  }
}
```

we were unable to run JQual in field-sensitive and context-sensitive mode, because JQual does not scale in this mode.

8 Related work

Section 8.1 discusses previous work that discovers immutability (for example, determines when a parameter is never modified during execution). Section 8.2 discusses previous work that checks or enforces mutability annotations written by the programmer (or inserted by a tool).

8.1 Discovering mutability

Early work (Banning 1979; Cooper and Kennedy 1988) on analyzing programs to determine what mutations may occur considered only pointer-free languages, such as Fortran. In such a language, aliases are induced only by reference parameter passing, and aliases persist only until the procedure returns. Analyses that compute MOD set (modified parameters) determine which of the reference parameters, and which global variables, are assigned by the body of a procedure. Our static analysis extends this work to handle pointers and object-oriented programs, and incorporates field-sensitivity.

Subsequent research, often called side-effect analysis, addressed aliasing in languages containing pointers. An update `r.f` $=$ `v` has the potential to modify any object that might be referred to by `r`. An alias analysis can determine the possible referents of pointers and thus the possible side effects. (An alias or class analysis also aids in call graph construction for object-oriented programs, by indicating the type of receivers and so disambiguating virtual calls.) This work indicates which aliased locations might also be mutated (Landi et al. 1992)—often reporting results in terms of the number of locations (typically, an allocation site in the program) that may be referenced—but less often indicates what other variables in the program might also refer to that site. More relevantly, it does not answer reference immutability questions regarding what references might be used to perform a mutation; ours is the first analysis to do so. A follow-on alias or escape analysis can be used to strengthen reference immutability into object immutability (Birka and Ernst 2004).

New alias/class analyses yield improved side-effect analyses (Ryder et al. 2001; Rountev 2004). Landi et al. (1993) improve the precision of previous work by using program-point-specific aliasing information. Ryder et al. (2001) compare the flow-sensitive algorithm (Landi et al. 1993) with a flow-insensitive one that yields a single alias result that is valid throughout the program. The flow-sensitive version is more precise but slower and unscalable, and the flow-insensitive version provides adequate precision for certain applications. Milanova et al. (2002) provide a yet more precise algorithm via an object-sensitive, flow-insensitive points-to analysis that analyzes a method separately for each of the objects on which the method is invoked. Object sensitivity outperforms Andersen's context-insensitive analysis (Rountev et al. 2001). Rountev (2004) compares RTA to a context-sensitive points-to analysis for call graph construction, with the goal of improving side-effect analysis. Rountev's experimental results suggest that sophisticated pointer analysis may not be necessary to achieve good results. (This mirrors other work questioning the usefulness of highly complex pointer analysis, Ruf 1995, Hind 2001.) We, too, compared a sophisticated analysis (Sălcianu's) to a simpler one (ours) and found the simpler one competitive.

Side-effect analysis (Choi et al. 1993; Rountev and Ryder 2001; Milanova et al. 2002; Rountev 2004; Sălcianu and Rinard 2005; Sălcianu 2006) originated in the compiler community and has focused on i-sound analyses. Our work investigates other tradeoffs and other uses for the immutability information. Specifically, differently from previous research, our work (1) computes both *mutable* and *immutable* classifications, (2) trades off soundness and precision to improve overall accuracy, (3) combines dynamic and static stages, (4) includes a novel dynamic mutability analysis, and (5) permits an analysis to explicitly represent its incompleteness.

Preliminary results of using side effect analysis for optimization—an application that requires an i-sound analysis—show modest speedups. Le et al. (2005) report speedups of 3–5% for a coarse CHA analysis, and only 1% more for a finer points-to analysis. Clausen (1997) reports an average 4% speedup, using a CHA-like side effect analysis in which each field is marked as side-effected or not. Razafimahefa (1999) reports an average 6% speedup for loop invariant code motion in an inlining JIT, Xu et al. (2007) report slowdowns in a memoization optimization. Le et al. (2005) summarize their own and related work as follows: "Although precision of the underlying analyses tends to have large effects on static counts of optimization opportunities, the effects on dynamic behavior are much smaller; even simple analyses provide most of the improvement."

Rountev (2004) and Sălcianu (Sălcianu and Rinard 2005; Sălcianu 2006) developed static analyses for determining side-effect-free methods. Like our static analysis component, they combine a pointer analysis, an intra-procedural analysis to determine "immediate" side effects, and inter-procedural propagation to determine transitive side effects. Sălcianu defines a side-effect-free method as one that does not modify any heap cell that existed when the method was called. Rountev's definition is more restricted and prohibits a side-effect-free method from creating and returning a new object, or creating and using a temporary object. Sălcianu's analysis can compute per-parameter mutability information in addition to per-method side effect information. (A method is side-effect-free if it modifies neither its parameters nor the global state, which is an implicit parameter.) Rountev's coarser analysis results are one reason that we cannot compare directly to his implementation. Rountev applies his analysis to program fragments by creating an artificial `main` routine that calls all methods of interest; we adopted this approach in augmenting J (see Sect. 6).

Sălcianu's (Sălcianu and Rinard 2005; Sălcianu 2006) analysis uses a complex pointer analysis. Its flow-insensitive method summary represents in a special way objects allocated by the current method invocation, so a side-effect-free method may perform side effects on a newly-allocated objects. Like ours, Sălcianu's analysis handles code that it does not have access to, such as native methods, by using manually prepared annotations. Sălcianu describes an algorithm for computing object immutability and proves it sound, but his implementation computes reference immutability (not object immutability) and contains some minor unsoundness. We evaluated our analyses, which also compute reference immutability, against Sălcianu's implementation (Sect. 5). In the experiments, our staged analyzed achieve comparable or better accuracy and better run-time performance.

Javarifier (Correa Jr. et al. 2007; Quinonez et al. 2008; Quinonez 2008) infers the reference immutability type qualifiers of the Javari extension of Java (Birka and Ernst 2004; Tschantz and Ernst 2005). Starting at field reassignments (the source of all object side-effects), Javarifier flow- and context-sensitively propagates mutation information to all references, including method receivers. Our case studies using Javarifier (Sect. 7.3.3) show that Javari is sometimes more restrictive than our formal definition due to the conservative nature of its type rules. JQual (Greenfieldboyce and Foster 2007) is a framework for inference of type qualifiers. JQual's definition of reference immutability and inference algorithm are similar to those of Javari.

Porat et al. (2000), Biberstein et al. (2001) infer class immutability for global (static) variables in Java's `rt.jar`, thus indicating the extent to which immutability can be found in practice; the work also addresses sealing/encapsulation. Foster et al. (1999) developed an inference algorithm for `const` annotations using Cqual, a tool for adding type qualifiers to C programs. Their algorithm does not handle aliasing. Foster et al. also present a polymorphic version of `const` inference, in which a single reference may have zero or more annotations, depending on the context.

Other researchers have also explored the idea of dynamic side-effect analysis. Dallmeier and Zeller developed the JDynPur tool (http://www.st.cs.uni-sb.de/models/jdynpur) for offline dynamic side-effect analysis (not parameter mutability) but provide no description of the algorithm or experimental results. Xu et al. (2007) developed dynamic analyses for detecting side-effect-free methods. Their work differs significantly from ours. Xu et al. consider only the method's receiver, while our analyses are more fine-grained and produce results for all formal parameters, including the receiver. Xu et al. examine only one analysis at a time. In contrast, our framework combines the strengths of static and dynamic analyses. Xu et al. do not present an evaluation of the effectiveness of their analyses in terms of precision and recall, they only report the percentage of methods identified as pure by their analyses. In contrast, we established the immutability of more than 8800 method parameters by manual inspection and report the results of our 192 analysis combinations with respect to the established ground truth. Finally, Xu et al.'s dynamic analysis is unsound. In contrast, our analysis framework is sound and we provide sound analyses, both static and dynamic, to use in the framework.

8.2 Specifying and checking mutability

A verification approach enforces reference immutability annotations written in the source code. Soundness requires that any cannot modify an object that was annotated as immutable. Since the problem is uncomputable, any static, sound system for checking annotations rejects some programs that cannot actually violate the immutability specifications at run-time.

Annotation-based approaches include Islands (Hogg 1991), Flexible Alias Protection (Noble et al. 1998), C++ `const` (Stroustrup 2000), ModeJava (Skoglund and Wrigstad 2001), JAC (Kniesel and Theisen 2001), Capabilities (Boyland et al. 2001), Javari (Birka and Ernst 2004; Tschantz and Ernst 2005), Universes (Dietl and Müller 2005), Relation types (Vaziri et al. 2007), and IGJ (Zibin et al. 2007). Many of these systems provide features beyond mutability annotations; for example, problems of ownership and aliasing can contribute to mutation errors, so a type system may also address those issues.

All the above approaches, except for Capabilities and C++'s `const` (which provide only non-transitive reference immutability), state as their goal the reference immutability described informally in the Introduction. Boyland (2005) also noticed those similarities. In JAC (Kniesel and Theisen 2001), "the declarative definition of read-only types is: for an expression of a read-only type that evaluates to an object reference `r`, the full state of the referenced object is protected against changes performed via `r`." In ModeJava (Skoglund and Wrigstad 2001), "a read reference is a reference that is never used for modification of its referenced object (including retrieval of write references that may in turn be used for modification)." In Javari (Birka and Ernst 2004), "A read-only reference is a reference that cannot be used to modify the object to which it refers." In Universes (Dietl and Müller 2005), "references [...] must not be used to modify the referenced object since the reference is not guaranteed to come from the owner or a peer object of the modified object. Hence, we call these references readonly references."

However, while some of these descriptions are formal by having type rules, none specifies precisely what it means that a modification happens through the reference. Without a formal definition of reference immutability, it is not possible to compare different systems for inferring or checking it, nor is it possible to evaluate the systems' trade-offs between expressiveness and checkability. Capabilities (Boyland et al. 2001), and Javari (Birka and Ernst 2004), do provide a formal definition for their systems, but do so using the programming language. In Javari (Capabilities are similar), a reference is read-only if it is possible to annotate it with readonly (i.e., the type system issues no error). In contrast, this paper provides a formal definition that is independent of how it is calculated.

Object immutability is a stronger property than reference immutability: it guarantees that a particular value is never modified, even through aliased parameters. Reference immutability, together with an alias or escape analysis, is enough to establish object immutability (Birka and Ernst 2004). Pechtchanski and Sarkar (2002) allows the user to annotate his code with object immutability annotations and employs a combination of static and dynamic analysis to detect where those annotations are violated. The IGJ language (Zibin et al. 2007) supports both reference and object immutability via a type system based on Java generics.

Side-effect analysis is different from parameter reference immutability, which is our focus. Side-effect analysis concerns methods and whether the heap can be modified during the method's execution. Parameter reference immutability concerns references to method parameters and whether they can be used to modify the state of objects. Except for very strict definitions of purity (such as strong purity, Xu et al. 2007), method purity can often be computed from parameter reference immutability information (combined with analysis of globals).

9 Conclusion

We formally define parameter reference immutability. Previous work relied mostly on informal descriptions and type systems, both for inferring immutable references, and for checking annotations. The formal definition in this paper encompasses those informal notions and enables unbiased comparisons between different inference and type-annotation approaches.

We have described Pidasa, a staged mutability analysis framework for Java, along with a set of component analyses that can be plugged into the analysis. The framework permits combinations of mutability analyses, including static and dynamic techniques. The framework explicitly represents analysis incompleteness and reports both immutable and mutable parameters. Our component analyses take advantage of this feature of the framework.

Our dynamic analysis is novel, to the best of our knowledge; at run time, it marks parameters as mutable based on mutations of objects. We presented a series of heuristics, optimizations, and enhancements that make it practical. For example, iterative random test input generation appears competitive with user-supplied sample executions. Our static analysis reports both *immutable* and *mutable* parameters, and it demonstrates that a simple, scalable analysis can perform at a par with much more heavyweight and sophisticated static analyses. Combining the lightweight static and dynamic analyses yields a combined analysis with many of the positive features of both, including both run-time performance and accuracy.

Our evaluation of Pidasa, includes many different combinations of staged analysis, in both sound and unsound varieties. This evaluation sheds insight into both the complexity of the problem and the sorts of analyses that can be effectively applied to it. We also show how the results of the mutability analysis can improve a client analysis.

We compared parameter immutability in four systems (a type system Javari, and three analysis tools: JMH, Pidasa and JQual), on a large set of parameters in several programs. We then compared the results to the classification based on the formal definition, and analyzed the discrepancies. The results provide insight into the trade-offs each system makes between expressibility and verifiability. We observed that different systems vary in their approach to that trade-off. Javari is a type system and its type rules are conservative with respect to immutability, which leads Javari to mark some parameter references as mutable when in fact they cannot be used in a mutation. JMH is able to infer more immutable parameters than Javari, but it also is conservative and thus under-approximates the set of immutable references. Pidasa's inference combines static and dynamic analyses and offers both a conservative variant and a non-conservative variant that achieves the highest recall, with a small loss in precision.

Our case studies show that existing systems for expressing and inferring reference immutability approximate the formal definition we present in this paper. We hope that future research in this area finds the formal definition useful as a point of reference for verifying correctness and assessing expressibility.

This paper extends our previous work (Artzi et al. 2007) in the following way: Formal parameter mutability definition (Sect. 2) and a comparison of different tools implementing similar mutability definitions (Sect. 7).

Acknowledgements We thank Yoav Zibin for his help with the formalization and comments on the paper, Carlos Pacheco for his help with the random input generator, and Alexandru Sălcianu for assistance with JPPA.

Appendix: Additional example of reference immutability

This section presents an additional example of Definition 1. We apply Definition 1 to the method `modifyAll` of Fig. 1 (which is problematic for JPPA).

The left hand side of Fig. 21 contains the partial Java code for a simplified version of the method `modifyAll` of Fig. 1. The right hand side contains the same method after in MLJ.

It is obvious that parameters $\mathtt{p}_2$ and $\mathtt{p}_3$ of method `m` are reference-*mutable* (line 2 modifies $\mathtt{p}_2$, and line 4 modifies $\mathtt{p}_3$`.f`). It is harder to see that parameter $\mathtt{p}_1$ is also reference-*mutable*. $\mathtt{p}_1$ is mutated when, for example, parameters $\mathtt{p}_2$ and $\mathtt{p}_3$ are aliased (like in the call `m(x, y, y)`). In this case, the state of the object passed to $\mathtt{p}_1$ is modified on line 4 using a series of dereferences from $\mathtt{p}_1$. Sălcianu (2006) presents this method as an example for unsoundness in his static analysis. Sălcianu's tool, JMH, wrongly classifies parameter $\mathtt{p}_1$ as immutable.

Figure 22 presents the application of Definition 1 to method `m` by evaluating `m` in MLJ. Step 2 contains the expression and the stores that are the results of evaluating all the initial constructors. Step 3 is the result of invoking the method `n`. Step 4 is the result of invoking the method `m`. Notably, $\mathtt{p}_2$ is replaced with $(\!|o_6, \{n^1.p_2, m^1.p_2\}|\!)$ and $\mathtt{p}_3$ is replaced with the same object (o_6) but with a different value $(\!|o_6, \{n^1.p_2, m^1.p_3\}|\!)$. After evaluating the first `set` expression (corresponding to $\mathtt{p}_2.\mathtt{f} = \mathtt{p}_1$), the parameters $\mathtt{n.p}_2$ and $\mathtt{m.p}_2$ in the mutability set of

```
1    void m(B p1, C p2, C p3) {
2      p2.f = p1;
3      B l = p3.f;
4      l.f = this;
5    }
```

```
1    class A extends Object {
2      A(){ super(); }
3      A n(B p1, C p2){ return this.m(p1, p2, p2) }
4      A m(B p1, C p2, C p3){ set p2.f = p1 then q(p3.f) }
5      A q(B l){ return set l.f = this then this }
6    }
7    class B extends Object {
8      A f;
9      B(A f){ super(); this.f = f }
10   }

11   class C extends Object {
12     B f;
13     C(B f){ super(); this.f = f }
14   }

     new A().n(new B(new A()), new C(new B(new A())))
```

Fig. 21 The *left-hand side* contains a simplified version of method `modifyAll` from Fig. 1. Parameters $\mathtt{p}_2$ and $\mathtt{p}_3$ are mutable since they are mutated in lines 2 and 4. Parameter p_1 is also mutable. It will be mutated when parameters $\mathtt{p}_2$ and $\mathtt{p}_3$ are aliased, for example in the call `m(x, y, y)`. The *right-hand side* contains the same method and the call `m(x, y, y)`, converted to MLJ. Since MLJ has no local variables, we have converted the local variable `l` into a method's parameter (method `q`). Method `n` passes the same parameter twice to method `m`, used to implement the Java call `m(x, y, y)`

	expression	S	Ω	next rule
1	$new\ A().n(new\ B(new\ A()), new\ C(new\ B(\mathtt{new}\ A())))$	$\varnothing$	$\varnothing$	[R-NEW]
2	$\llparenthesis o_1, \varnothing\rrparenthesis.n(\llparenthesis o_4, \varnothing\rrparenthesis, \llparenthesis o_6, \varnothing\rrparenthesis)$	$\{(o_1, \langle A, \varnothing\rangle), (o_2, \langle A, \varnothing\rangle), (o_3, \langle A, \varnothing\rangle)$ $(o_4, \langle B, \{f : \llparenthesis o_2, \varnothing\rrparenthesis\}\rangle), (o_5, \langle B, \{f : \llparenthesis o_3, \varnothing\rrparenthesis\}\rangle)$ $(o_6, \langle B, \{f : \llparenthesis o_5, \varnothing\rrparenthesis\}\rangle)\}$	$\varnothing$	[R-INVK]
3	$ret\ n^1\ \llparenthesis o_1, \{n^1.this\}\rrparenthesis.m(\llparenthesis o_4, \{n^1.p_1\}\rrparenthesis, \llparenthesis o_6, \{n^1.p_2\}\rrparenthesis, \llparenthesis o_6, \{n^1.p_2\}\rrparenthesis)$	...	$\varnothing$	[R-INVK]
4	$ret\ n^1\ ret\ m^1\ set\ \llparenthesis o_6, \{n^1.p_2, m^1.p_2\}\rrparenthesis.f =$	...	$\varnothing$	[R-SET]
	$\llparenthesis o_4, \{n^1.p_1, m^1.p_1\}\rrparenthesis\ then\ this.q(\llparenthesis o_6, \{n^1.p_2, m^1.p_3\}\rrparenthesis.f)$		$\varnothing$	
5	$ret\ n^1\ ret\ m^1\ this.q(\llparenthesis o_6, \{n^1.p_2, m^1.p_3\}\rrparenthesis.f)$	$\{(o_1, \langle A, \varnothing\rangle), (o_2, \langle A, \varnothing\rangle), (o_3, \langle A, \varnothing\rangle)$ $(o_4, \langle B, \{f : \llparenthesis o_2, \varnothing\rrparenthesis\}\rangle), (o_5, \langle B, \{f : \llparenthesis o_3, \varnothing\rrparenthesis\}\rangle)$ $(o_6, \langle B, \{f : \llparenthesis o_4, \{n^1.p_1, m^1.p_1\}\rrparenthesis\}\rangle)\}$	$n.p_2, m.p_2$	[R-FIELD]
6	$ret\ n^1\ ret\ m^1\ this.q(\llparenthesis o_4, \{n^1.p_2, m^1.p_3, n^1.p_1, m^1.p_1\}\rrparenthesis)$	...	...	[R-INVK]
7	$ret\ n^1\ ret\ m^1\ ret\ q^1\ set\ \llparenthesis o_4, \{n^1.p_2, m^1.p_3, n^1.p_1, m^1.p_1, q^1.p_1\}\rrparenthesis.f =$	...	...	[R-SET]
	$\llparenthesis o_1, \{n^1.this, m^1.this, q^1.this\}\rrparenthesis\ then\ \llparenthesis o_1, \{n^1.this, m^1.this, q^1.this\}\rrparenthesis$			
8	$ret\ n^1\ ret\ m^1\ ret\ q^1\ \llparenthesis o_1, \{n^1.this, m^1.this, q^1.this\}\rrparenthesis$	$\{(o_1, \langle A, \varnothing\rangle), (o_2, \langle A, \varnothing\rangle), (o_3, \langle A, \varnothing\rangle)$ $(o_4, \langle B, \{f : \llparenthesis o_1, \{n^1.this, m^1.this, q^1.this\}\rrparenthesis\}\rangle)$ $(o_5, \langle B, \{f : \llparenthesis o_3, \varnothing\rrparenthesis\}\rangle)$ $(o_6, \langle B, \{f : \llparenthesis o_4, \{n^1.p_1, m^1.p_1\}\rrparenthesis\}\rangle)\}$	$n.p_2, m.p_2, m.p_3$ $n.p_1, m.p_1, q.p_1$	[R-RET]
9	$ret\ n^1\ ret\ m^1\ \llparenthesis o_1, \{n^1.this, m^1.this\}\rrparenthesis$	$\{(o_1, \langle A, \varnothing\rangle), (o_2, \langle A, \varnothing\rangle), (o_3, \langle A, \varnothing\rangle)$ $(o_4, \langle B, \{f : \llparenthesis o_1, \{n^1.this, m^1.this\}\rrparenthesis\}\rangle)$ $(o_5, \langle B, \{f : \llparenthesis o_3, \varnothing\rrparenthesis\}\rangle)$ $(o_6, \langle B, \{f : \llparenthesis o_4, \{n^1.p_1, m^1.p_1\}\rrparenthesis\}\rangle)\}$	$n.p_2, m.p_2, m.p_3$ $n.p_1, m.p_1, q.p_1$	[R-RET]
10	$ret\ n^1\ \llparenthesis o_1, \{n^1.this\}\rrparenthesis$	$\{(o_1, \langle A, \varnothing\rangle), (o_2, \langle A, \varnothing\rangle), (o_3, \langle A, \varnothing\rangle)$ $(o_4, \langle B, \{f : \llparenthesis o_1, \{n^1.this\}\rrparenthesis\}\rangle)$ $(o_5, \langle B, \{f : \llparenthesis o_3, \varnothing\rrparenthesis\}\rangle)$ $(o_6, \langle B, \{f : \llparenthesis o_4, \{n^1.p_1\}\rrparenthesis\}\rangle)\}$	$n.p_2, m.p_2, m.p_3$ $n.p_1, m.p_1, q.p_1$	[R-RET]
11	$\llparenthesis o_1, \varnothing\rrparenthesis$	$\{(o_1, \langle A, \varnothing\rangle), (o_2, \langle A, \varnothing\rangle), (o_3, \langle A, \varnothing\rangle)$ $(o_4, \langle B, \{f : \llparenthesis o_1, \varnothing\rrparenthesis\}\rangle)$ $(o_5, \langle B, \{f : \llparenthesis o_3, \varnothing\rrparenthesis\}\rangle)$ $(o_6, \langle B, \{f : \llparenthesis o_4, \varnothing\rrparenthesis\}\rangle)\}$	$n.p_2, m.p_2, m.p_3$ $n.p_1, m.p_1, q.p_1$	*Done*

Fig. 22 Applying Definition 1 to the code in Fig. 21. The figure shows the evaluation using the rules of Fig. 4. Symbol ... means that a store does not change between evaluation steps. After the rule [R-SET] is applied in step 7, the parameter $m.p_1$ is classified as mutable. The mutation to this parameter is not obvious and it depends on the aliasing between the other two parameters p_2 and p_3

$(\!|o_6, \{n^1.p_2, m^1.p_2\}|\!)$ are classified as mutable. Step 6 presents the result of evaluating the field access. The resulting object o_4 has the combined mutability set of both its previous value $(\!|o_4, \{n^1.p_1, m^1.p_1\}|\!)$ (from the store) those of the dereferenced value $(\!|o_6, \{n^1.p_2, m^1.p_3\}|\!)$. When that value is modified in step 7, all the parameters in its mutability set $(n^1.p_1, m^1.p_1, n^1.p_2, m^1.p_3)$ are classified as mutable. Thus definition recognizes that parameter $m.p_1$ is mutable.

References

Artzi, S., Ernst, M.D., Kieżun, A., Pacheco, C., Perkins, J.H.: Finding the needles in the haystack: Generating legal test inputs for object-oriented programs. In: M-TOOS 2006: 1st Workshop on Model-Based Testing and Object-Oriented Systems. Portland, OR, USA (2006)

Artzi, S., Kieżun, A., Glasser, D., Ernst, M.D.: Combined static and dynamic mutability analysis. In: ASE 2007: Proceedings of the 22nd Annual International Conference on Automated Software Engineering, pp. 104–113. Atlanta, GA, USA (2007)

Bacon, D.F., Sweeney, P.F.: Fast static analysis of C++ virtual function calls. In: Conference on Object-Oriented Programming, Systems, Languages, and Applications, pp. 324–341. San Jose, CA, USA (1996). http://doi.acm.org/10.1145/236337.236371

Banning, J.P.: An efficient way to find the side effects of procedure calls and the aliases of variables. In: Proceedings of the Sixth Annual ACM Symposium on Principles of Programming Languages, pp. 29–41 (1979)

Biberstein, M., Gil, J., Porat, S.: Sealing, encapsulation, and mutability. In: ECOOP 2001—Object-Oriented Programming, 15th European Conference, pp. 28–52. Budapest, Hungary (2001)

Birka, A., Ernst, M.D.: A practical type system and language for reference immutability. In: Object-Oriented Programming Systems, Languages, and Applications (OOPSLA 2004), pp. 35–49. Vancouver, BC, Canada (2004)

Boyland, J.: Why we should not add `readonly` to Java (yet). In: FTfJP'2005: 7th Workshop on Formal Techniques for Java-like Programs. Glasgow, Scotland (2005)

Boyland, J., Noble, J., Retert, W.: Capabilities for sharing: A generalisation of uniqueness and read-only. In: ECOOP 2001—Object-Oriented Programming, 15th European Conference, pp. 2–27. Budapest, Hungary (2001)

Burdy, L., Cheon, Y., Cok, D., Ernst, M.D., Kiniry, J., Leavens, G.T., Leino, K.R.M., Poll, E.: An overview of JML tools and applications. Int. J. Softw. Tools Technol. Transf. **7**(3), 212–232 (2005)

Cataño, N., Huisman, M.: Chase: a static checker for JML's assignable clause. In: VMCAI'03, Fourth International Conference on Verification, Model Checking and Abstract Interpretation, pp. 26–40. New York, New York (2003)

Choi, J.D., Burke, M., Carini, P.: Efficient flow-sensitive interprocedural computation of pointer-induced aliases and side effects. In: Proceedings of the Twentieth Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, pp. 232–245. Charleston, SC (1993)

Clausen, L.R.: A Java bytecode optimizer using side-effect analysis. Concurr. Pract. Exp. **9**(11), 1031–1045 (1997)

Cooper, K.D., Kennedy, K.: Interprocedural side-effect analysis in linear time. In: PLDI 1988, Proceedings of the SIGPLAN'88 Conference on Programming Language Design and Implementation, pp. 57–66. Atlanta, GA, USA (1988)

Correa Jr., T.L., Quinonez, J., Ernst, M.D.: Tools for enforcing and inferring reference immutability in Java. In: Companion to Object-Oriented Programming Systems, Languages, and Applications (OOPSLA 2007), pp. 866–867. Montréal, Canada (2007)

Dallmeier, V., Lindig, C., Wasylkowski, A., Zeller, A.: Mining object behavior with ADABU. In: WODA 2006: Workshop on Dynamic Analysis, pp. 17–24. Shanghai, China (2006)

Dean, J., Grove, D., Chambers, C.: Optimization of object-oriented programs using static class hierarchy analysis. In: ECOOP'95, the 9th European Conference on Object-Oriented Programming, pp. 77–101. Åarhus, Denmark (1995)

Demsky, B., Rinard, M.: Role-based exploration of object-oriented programs. In: ICSE'02, Proceedings of the 24th International Conference on Software Engineering, pp. 313–324. Orlando, Florida (2002)

Dietl, W., Müller, P.: Universes: Lightweight ownership for JML. J. Object Technol. **4**(8), 5–32 (2005)

Diwan, A., Moss, J.E.B., McKinley, K.S.: Simple and effective analysis of statically-typed object-oriented programs. In: Conference on Object-Oriented Programming, Systems, Languages, and Applications, pp. 292–305. San Jose, CA, USA (1996)

Dolado, J.J., Harman, M., Otero, M.C., Hu, L.: An empirical investigation of the influence of a type of side effects on program comprehension. IEEE Trans. Softw. Eng. **29**(7), 665–670 (2003)

Ernst, M.D., Cockrell, J., Griswold, W.G., Notkin, D.: Dynamically discovering likely program invariants to support program evolution. IEEE Trans. Softw. Eng. **27**(2), 99–123 (2001). A previous version appeared in ICSE'99, Proceedings of the 21st International Conference on Software Engineering, pp. 213–224. Los Angeles, CA, USA, 19–21 May 1999

Foster, J.S., Fähndrich, M., Aiken, A.: A theory of type qualifiers. In: PLDI 1999, Proceedings of the ACM SIGPLAN'99 Conference on Programming Language Design and Implementation, pp. 192–203. Atlanta, GA, USA (1999)

Fowler, M.: Refactoring: Improving the Design of Existing Code. Addison-Wesley, Reading (2000)

Greenfieldboyce, D., Foster, J.S.: Type qualifier inference for Java. In: Object-Oriented Programming Systems, Languages, and Applications (OOPSLA 2007), pp. 321–336. Montréal, Canada (2007)

Guo, P.J.: A scalable mixed-level approach to dynamic analysis of C and C++ programs. Master's thesis, MIT Department of Electrical Engineering and Computer Science, Cambridge, MA (2006)

Hind, M.: Pointer analysis: Haven't we solved this problem yet? In: ACM SIGPLAN/SIGSOFT Workshop on Program Analysis for Software Tools and Engineering (PASTE'01), pp. 54–61. Snowbird, Utah, USA (2001)

Hogg, J.: Islands: Aliasing protection in object-oriented languages. In: Conference on Object-Oriented Programming, Systems, Languages, and Applications, pp. 271–285. Phoeniz, AZ, USA (1991)

Igarashi, A., Pierce, B.C., Wadler, P.: Featherweight Java: a minimal core calculus for Java and GJ. ACM Trans. Program. Lang. Syst. **23**(3), 396–450 (2001)

Kniesel, G., Theisen, D.: JAC — access right based encapsulation for Java. Softw. Pract. Exp. **31**(6), 555–576 (2001)

Landi, W., Ryder, B.G.: A safe approximate algorithm for interprocedural pointer aliasing. In: PLDI 1992, Proceedings of the SIGPLAN'92 Conference on Programming Language Design and Implementation, pp. 235–248. San Francisco, Calif. (1992)

Landi, W., Ryder, B.G., Zhang, S.: Interprocedural modification side effect analysis with pointer aliasing. In: PLDI 1993, Proceedings of the SIGPLAN'93 Conference on Programming Language Design and Implementation, pp. 56–67. Albuquerque, NM, USA (1993)

Le, A., Lhoták, O., Hendren, L.: Using inter-procedural side-effect information in JIT optimizations. In: Compiler Construction: 14th International Conference, CC 2005, pp. 287–304. Edinburgh, Scotland (2005)

Mariani, L., Pezzè, M.: Behavior capture and test: Automated analysis of component integration. In: International Conference on Engineering of Complex Computer Systems, pp. 292–301. Shanghai, China (2005)

Milanova, A., Rountev, A., Ryder, B.G.: Parameterized object sensitivity for points-to and side-effect analyses for Java. In: ISSTA 2002, Proceedings of the 2002 International Symposium on Software Testing and Analysis, pp. 1–11. Rome, Italy (2002)

Noble, J., Vitek, J., Potter, J.: Flexible alias protection. In: ECOOP'98, the 12th European Conference on Object-Oriented Programming, pp. 158–185. Brussels, Belgium (1998)

O'Callahan, R., Choi, J.D.: Hybrid dynamic data race detection. In: Proceedings of the 9th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming, pp. 167–178. San Diego, CA, USA (2003). http://doi.acm.org/10.1145/781498.781528

Pacheco, C., Lahiri, S.K., Ernst, M.D., Ball, T.: Feedback-directed random test generation. In: ICSE'07, Proceedings of the 29th International Conference on Software Engineering, pp. 75–84. Minneapolis, MN, USA (2007)

Papi, M.M., Ali, M., Correa Jr., T.L., Perkins, J.H., Ernst, M.D.: Practical pluggable types for Java. In: ISSTA 2008, Proceedings of the 2008 International Symposium on Software Testing and Analysis, pp. 201–212. Seattle, WA, USA (2008)

Pechtchanski, I., Sarkar, V.: Immutability specification and its applications. In: Joint ACM-ISCOPE Java Grande Conference, pp. 202–211. Seattle, WA (2002)

Pierce, B.C.: Types and Programming Languages. MIT Press, Cambridge (2002)

Polishchuk, M., Liblit, B., Schulze, C.: Dynamic heap type inference for program understanding and debugging. In: Proceedings of the 34rd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, pp. 39–46. Nice, France (2007)

Porat, S., Biberstein, M., Koved, L., Mendelson, B.: Automatic detection of immutable fields in Java. In: CASCON. Mississauga, Ontario, Canada (2000)

Quinonez, J.: Inference of reference immutability in Java. Master's thesis, MIT Department of Electrical Engineering and Computer Science, Cambridge, MA (2008)

Quinonez, J., Tschantz, M.S., Ernst, M.D.: Inference of reference immutability. In: ECOOP 2008—Object-Oriented Programming, 22nd European Conference, pp. 616–641. Paphos, Cyprus (2008)

Razafimahefa, C.: A study of side-effect analyses for Java. Master's thesis, School of Computer Science, McGill University, Montreal, Canada (1999)

Rountev, A.: Precise identification of side-effect-free methods in Java. In: ICSM 2004, Proceedings of the International Conference on Software Maintenance, pp. 82–91. Chicago, Illinois (2004)

Rountev, A., Ryder, B.G.: Points-to and side-effect analyses for programs built with precompiled libraries. In: Compiler Construction: 10th International Conference, CC 2001, pp. 20–36. Genova, Italy (2001)

Rountev, A., Milanova, A., Ryder, B.G.: Points-to analysis for Java based on annotated constraints. In: Object-Oriented Programming Systems, Languages, and Applications (OOPSLA 2001), pp. 43–55. Tampa Bay, FL, USA (2001)

Ruf, E.: Context-insensitive alias analysis reconsidered. In: PLDI 1995, Proceedings of the SIGPLAN'95 Conference on Programming Language Design and Implementation, pp. 13–22. La Jolla, CA, USA (1995)

Ryder, B.G., Landi, W.A., Stocks, P.A., Zhang, S., Altucher, R.: A schema for interprocedural modification side-effect analysis with pointer aliasing. ACM Trans. Program. Lang. Syst. **23**(2), 105–186 (2001)

Sălcianu, A.: Pointer analysis for Java programs: Novel techniques and applications. Ph.D. thesis, MIT Department of Electrical Engineering and Computer Science, Cambridge, MA (2006)

Sălcianu, A., Rinard, M.C.: Purity and side-effect analysis for Java programs. In: VMCAI'05, Sixth International Conference on Verification, Model Checking and Abstract Interpretation, pp. 199–215. Paris, France (2005)

Savage, S., Burrows, M., Nelson, G., Sobalvarro, P., Anderson, T.: Eraser: A dynamic data race detector for multi-threaded programs. In: Proceedings of the 16th ACM Symposium on Operating Systems Principles, pp. 27–37. St. Malo, France (1997)

Skoglund, M., Wrigstad, T.: A mode system for read-only references in Java. In: FTfJP'2001: 3rd Workshop on Formal Techniques for Java-like Programs. Glasgow, Scotland (2001)

Stroustrup, B.: The C++ Programming Language, special edn. Addison-Wesley, Reading, (2000)

Tip, F., Palsberg, J.: Scalable propagation-based call graph construction algorithms. In: Object-Oriented Programming Systems, Languages, and Applications (OOPSLA 2000), pp. 281–293. Minneapolis, MN, USA (2000)

Tkachuk, O., Dwyer, M.B.: Adapting side effects analysis for modular program model checking. In: ESEC/FSE 2003: Proceedings of the 9th European Software Engineering Conference and the 11th ACM SIGSOFT Symposium on the Foundations of Software Engineering, pp. 188–197. Helsinki, Finland (2003)

Tschantz, M.S.: Javari: Adding reference immutability to Java. Master's thesis, MIT Department of Electrical Engineering and Computer Science, Cambridge, MA (2006)

Tschantz, M.S., Ernst, M.D.: Javari: Adding reference immutability to Java. In: Object-Oriented Programming Systems, Languages, and Applications (OOPSLA 2005), pp. 211–230. San Diego, CA, USA (2005)

Vaziri, M., Tip, F., Fink, S., Dolby, J.: Declarative object identity using relation types. In: ECOOP 2007—Object-Oriented Programming, 21st European Conference, pp. 54–78. Berlin, Germany (2007)

Weiser, M.: Program slicing. IEEE Trans. Softw. Eng. **SE-10**(4), 352–357 (1984)

Xie, T.: Augmenting automatically generated unit-test suites with regression oracle checking. In: ECOOP 2006—Object-Oriented Programming, 20th European Conference, pp. 380–403. Nantes, France (2006)

Xu, H., Pickett, C.J.F., Verbrugge, C.: Dynamic purity analysis for Java programs. In: ACM SIGPLAN/SIGSOFT Workshop on Program Analysis for Software Tools and Engineering (PASTE 2007), pp. 75–82. San Diego, CA, USA (2007)

Zibin, Y., Potanin, A., Ali, M., Artzi, S., Kieżun, A., Ernst, M.D.: Object and reference immutability using Java generics. In: ESEC/FSE 2007: Proceedings of the 11th European Software Engineering Conference and the 15th ACM SIGSOFT Symposium on the Foundations of Software Engineering, pp. 75–84. Dubrovnik, Croatia (2007)