CSSE Technical Report UWA-CSSE-14-001

Side effect and purity checking in Java: a review

Arran D. Stewart, Rachel Cardell-Oliver, Rowan Davies
School of Computer Science & Software Engineering
The University of Western Australia
Crawley WA 6009 Australia
email: {arran.stewart, rachel.cardell-oliver, rowan.davies}@uwa.edu.au

March 2014

Abstract

"Side effects" in programming language expressions have long been regarded as making programs hard to understand and prone to error. In object-oriented programming languages, proposals have been made for enforcing "purity" (an absence of side effects) in the methods of objects, thus restricting the use of side effects. We examine the different approaches to defining and checking purity and side effects in the Java programming language, and explain the background out of which the concepts arose.

Keywords: Java, purity, side effect, object-oriented, static analysis.

Contents

Introduction

It has long been recognized that "side effects" in programming language expressions make programs harder to understand and thus more prone to error. Proposals for controlling side effects have included code analyses which ensure that methods are "side-effect free", a property sometimes also called "purity" or "functional purity". We examine proposals for doing so in the Java programming language. These proposals can involve static or dynamic analysis – most typically, both – but we focus primarily on the static aspects of analysis.

In the functional language research community, the concept of "functional purity" has a long history, but its application to object-oriented languages like Java is much more recent.[1] When transplanted to the object-oriented paradigm, it remains an open question how purity is best defined; in particular, some proposed definitions of purity conflict with important tenets of object-oriented languages.

We examine the background to this research, and identify proposals for extending the Java programming language in order to allow pure methods to be statically checked and used in application-level software development. We focus on tools and extensions to the language which allow Java developers to make use of method purity in implementing code, rather than on, for instance, proposals for completely new JVM-based programming languages, or on specification and assertion languages which are intended to be used for system design.

Ierusalimschy and Rodriguez have outlined a number of possible "levels" of side-effect–freeness which methods in an object-oriented language could have, which are useful in classifying the analyses we consider, and which we label "P1" to "P4":

P1 Strict prohibition on side effects – the methods use only a "functional subset" of the language. Assignments, and allocation of memory on the heap, are prohibited.

P2 Methods can make assignments to local variables, but cannot make any change to global state – that is, when the method returns, the heap is in the same state it was before the method executed.

P3 Methods do not make any changes to existing objects on the heap, but can allocate new objects.

P4 No restrictions are placed on side effects, except perhaps by convention.

[1] We consider object-oriented languages as being ones which encourage an object-oriented style of programming, in which a running program is structured into *objects* with behaviour, state and identity, which communicate by passing messages to one another, and in which a message can be responded to in different ways by different classes of objects. See, for instance: Budd (2002, p 15 ff); Scott (2009, p 12); and Meyer (1988), especially pp 41–63. This is different to Cook (2009) and Van Roy and Haridi (2004, p 428), who distinguish objects from ADTs on the basis of which other entities their internal representation is accessible to.

Imperative models of side effects

The related terms *purity*, *side effect* and *computational effect* have been defined and used in various ways in the computer science literature. The term "side effect" arose from researchers and practitioners using imperative languages, whereas the term "purity" arose from the functional language community.

The literature on functional programming provides extensive discussion on possible meanings of "purity" and "side effect" – see, for instance Sabry (1998, 2009) – but these are not always consonant with the way the terms are used in the literature on object-oriented languages, or by the software engineering community.

The term "side effect" was originally applied to programming languages in the context of non–object-oriented, procedural languages such as Algol and Fortran. In such languages, *functions* are procedural abstractions intended to return a result, and *procedures* are procedural abstractions intended to change their environment (see Backus et al., 1962) – for instance, by altering non-local variables, or by performing I/O.

There is typically nothing to stop a programmer from writing a function which also changes its environment. This means that an expression that ostensibly appears to calculate a value using a function call might also change the value of non-local variables. This is undesirable, as it makes the interactions between portions of code more difficult to understand, and such changes were referred to as "side effects" of evaluation of the expression.

Knuth (1967) defined "side effect" as follows:

> "A 'side effect' is conventionally regarded as a change (invoked by a function designator) in the state of some quantities which are *own* variables or which are not local to the function designator. In other words, when a procedure is being called in the midst of some expression, it has side effects if in addition to computing a value it does input or output or changes the value of some variable that is not internal to the procedure."

("Own" variables in Algol are similar to "static" variables in C: they retain their value between calls to a function, and thus provide access to state which is not local to the function.) Wichmann (1973) defined side effects slightly differently:

> "A function has a side-effect if it produces some other effect upon its environment apart from delivering a value. In ALGOL a function can have a side-effect by one of the following means:
>
> (a) Use of own variables
>
> (b) Use of non-local variables (including assignment to parameters)
>
> (c) Failing
>
> (d) Call of a procedure or function which does (a) (b) or (c)."

Compared with Knuth's definition, Wichmann's definition seems to exclude input and output, but includes failure as a "side effect".

Many other authors have adopted a similar approach to defining side effects.[2] They tend to agree that for a function to modify global variables is a side effect, and that modifying purely local variables is not. Authors often differ on whether they explicitly consider variables which are neither global or local: such non-local, non-global variables include parameters which are passed by reference, or variables accessible only within a particular module. Authors also differ as to whether they explicitly mention I/O or failure as being a "side effect".

In many languages developed after Fortran and Algol no clear distinction between functions and procedures is made. In C, for instance, all subprograms are referred to as "functions", whether they return a value or not. Thus, some authors adopted a definition in which any subprogram which makes a change to non-local variables causes a "side effect", regardless of whether it is termed a "function" or not.[3] Some languages also have built-in operators which return a value as well as altering the state of variables (for instance, the "++" and "=" operators in C and C++), and authors discussing those languages tend to use the term "side effect" to encompass these changes to state too.[4]

Still broader definitions of side effect can be found in definitions drawn from the programming language theory community. An example of these is Scott (2009), who defines a side effect as follows (Scott, 2009, p 224):

> "In general, a programming language construct is said to have a *side effect* if it influences subsequent computation (and ultimately program output) in any other way than returning a value for use in the surrounding context."

This definition, unlike the previous ones, includes altering program flow. Pierce (2002, p 153) also includes alterations of program flow in his sample list of side effects:

> "[B]esides just yielding results, evaluation of terms ... may assign to mutable variables (reference cells, arrays, mutable record fields, etc.), perform input and output to files, displays, or network connections, make non-local transfers of control via exceptions, jumps, or continuations, engage in inter-process synchronization and communication, and so on. In the literature on programming languages, such 'side effects' of computation are more generally referred to as computational effects."

[2] See, for instance, Bottenbruch and Grau (1962), Wirth (1971, p 39), Popek et al. (1977), Sebesta (1996, p 329), Appleby and VandeKopple (1997, p 85), and Sebesta (2012, p 325).

[3] See, for instance, Banning (1979) and Ralston (2003).

[4] See, for instance, Kernighan and Pike (1999, p 8).

"Benign" side effects

We have seen that the term "side effect" originally arose to describe the undesirable way that evaluation of one expression could affect the evaluation of others. However, not all "side effects" identified by the definitions we have outlined are necessarily harmful in this way.

For instance, consider "own" variables, the use of which Knuth considered to be a "side effect". For an expensive calculation, it might be desirable to memoize the results of the calculation so that it need not be calculated again for the same arguments. In pseudocode:

```
function calculation (n)
    declare own lookup-table
    if n is in lookup-table then
        return lookup-table value for n
    else
        let r := result of expensive calculation
        add (n,r) to lookup-table
        return r
    end if
end function
```

Here, `lookup-table` is a variable which retains its value between calls to the function (an "own" variable in Algol terminology, or a "static" variable in C terminology). In this situation, although one could consider the `calculation` function as having a side effect – namely, altering the non-local variable `lookup-table` – this side effect is not detectable by the caller of the function. This sort of side effect has typically been referred to as "benign". Examples of this approach are provided by Liskov and Guttag (1986) and Meyer (1988), who give explicit definitions of "benign" side effects in relation to the languages CLU and Eiffel, respectively.

In the CLU language (Liskov and Guttag, 1986), objects can be mutable or immutable. A mutable object has operations which can modify its state, and an immutable object does not. However, Liskov and Guttag make it clear that "immutability" refers to the *observable* state of an object – operations can in fact make changes to the state of an immutable object, as long as these are not observable from outside of the object, and such are changed are referred to as "benign side effects" (Liskov and Guttag, 1986, p 77).

Meyer (1988) makes a similar distinction in describing the language Eiffel. Operations on objects in Eiffel are divided syntactically into *commands* (which modify objects, and do not return a value) and *queries* (which return a value). However, this distinction does not clarify whether queries can legitimately modify objects.

Meyer clarifies this by distinguishing *concrete side effects*, which are any modifications of the state of the object, from *abstract side effects* – modifications of the state of the object, which will also have the effect of altering the future result of a query accessible to clients of the object. Side effects which are concrete, but not

abstract, since they are not observable from outside the object, are benign, and may legitimately be incorporated in the code for a query.

The use-case for benign side effects on objects is similar to that for functions: expensive calculations can be memoized or calculated lazily. Liskov and Guttag give the example of an abstract data type for representing rational numbers. As abstract mathematical entities, rational numbers seem best represented as immutable objects. The question then arises, should they be stored in reduced form or not? Liskov and Guttag point out that one could lazily do the reduction as needed, and since this would have no effect on the observable properties of a rational number, it would be a benign side effect.

Strict versus observable purity in Java

When considering the incorporation of purity and side-effect-freeness – both terms are used – into Java, researchers have normally accepted that a complete prohibition on mutating state is far too strict to be a useful concept – although exceptions to this do exist. The side-effect analysis presented by Milanova et al. (2002) aims simply to identify all methods which can mutate objects, regardless of whether this is observable to a caller or not.

Ierusalimschy and Rodriguez (1995) also discuss the possibility of a complete prohibition on side effects, where "methods use only a 'functional subset' of the language", and any assignments or allocation of new objects are prohibited, but they dismiss this as "much too restrictive". They consider a slightly more liberal degree of purity – where methods can make assignments to local variables, but no other changes – but likewise regard this as too restrictive for object-oriented programming. Their rejection of these very strict levels of purity does seem to accord with the original purpose of identifying side effects as a phenomenon, namely to describe particular functions which would make evaluation of expressions containing them difficult to understand. As we have seen, side effects can mutate non-local state but still be benign. Banning benign effects does not seem to have obvious benefits in an object-oriented language, and would result in dramatic changes to the way programming in such languages was done.

Additionally, banning the use of side effects which are benign, and purely local to a method, would be at odds with a fundamental tenet of data abstraction: the idea that once one has specified a public "contract" for how a data abstraction is to behave, and implementer should be at liberty to implement it in whatever way they want. This is sometimes termed "encapsulation", which has been defined as (Snyder, 1986):

> "a technique for minimizing inter-dependencies among separately-written modules by defining strict external interfaces. The external interface of a module serves as a contract between the module and its clients, and thus between the designer of the module and other designers."

Meyer (1988), likewise, has suggested that classes and methods should have a public "contract", specifying their observable behaviour in terms of invariants.

Adopting this approach, we would conclude that purity is part of the externally observable behaviour of a method, and that it ought therefore to be specified in the methods contract. Standard Java methods do not have an enforceable "contract" of this sort, except insofar as is specified in the type signature; any invariants a method satisfies are normally just given in the form of comments to the methods. Languages such as Eiffel, by contrast, provide extensive facilities to describe the contracts to which classes and methods adhere. Several of the purity-checking systems we will examine provide a facility which falls between the two: they allow methods to be marked with Java annotations which indicate that the method is pure (that is, it adheres to a standard contract about what other objects it will modify).

In considering the approaches to purity checking based on this sort of criterion of "observable purity", some helpful definitions and distinctions are provided by Leavens et al. (2006) in their outline of the JML modelling language.

They consider that a location is "modified" when "it is allocated in both the pre-state of the method, reachable in the post-state, and has a value that is different in these two states" (where the "pre-state" is the state of the program just before the method body has started executing, and the "post-state" the state just after).

They describe a *temporary side effect* as being when a method makes a change to a location, but then changes it back to its original value, and a *benign side effect* as being assignment to a location in such a way that clients of the method cannot observe the change. The definition of Leavens et al. has been widely adopted by researchers in proposals for incorporating purity into Java.

Alternative definitions and approaches

Some authors have argued (for instance, Lippmeier, 2009) that the term "side effect" actually makes little sense in the case of language constructs or procedures which are being executed expressly *for* the purpose of achieving these effects (for instance, calling a mutator method on an object, or calling a routine which performs system output). Indeed, in programming language theory, it is now more common to refer instead to "computational effects", or simply "effects".

Although it is not an approach taken by any side-effect analyses in Java, some authors consider that the lapse of time ought to be considered a side effect (for instance, Sabry, 1998, 2009). Lippmeier (2009) points out, however, that this would entail that if two procedures produce the same result, but take significantly different amounts of time, then they should be considered as having different "side effects" (Lippmeier, 2009, p 33) – they would "mean" different things – which seems counter to normal usage. It would also entail that, when a compiler optimizes a code fragment in order to speed up its execution, the compiler is changing the meaning of the code – yet the very point of optimization is to speed up code *without* altering its meaning or effect.

Lippmeier points out that this is not the case in all languages. In a language intended for hard real-time systems, then a program which does not produce its result in the required time is simply *wrong* (Lippmeier, 2009, p 35) – elapsed time *is* part of the semantics of the language.

We suggest that defining a meaning for "purity", "side effect" or "effect" in a language will depend on what is included in the semantics of the language, and what features can be observed from within the language. In most imperative languages, the lapse of time is not part of the semantics of the language: an implementation of the language that is slower or faster than others is still a valid implementation.

As we have seen, a side effect in a procedure which cannot be detected from within the language by the caller can at the very least be considered benign, and by some authors is not considered an "effect" at all. This approach is adopted, for instance, by Lucassen and Gifford (1988) in their development of type-and-effect systems, in which they define the "effect" of an expression as "a concise summary of the observable side-effects that the expression may have when it is evaluated".

Observable purity analyses for Java

JPPA

Sălcianu and Rinard (2005) describe a purity analysis tool, JPPA (a "Java Pointer and Purity Analysis tool"),[5] and explicitly adopt a definition of purity based on Leavens et al. (2006): a method is pure if it does not mutate any pre-existing object. This means that pure methods can still construct and mutate structures as long as the structures are freshly allocated, and can return such structures as a result.

JPPA is based on a combined pointer and escape analysis, which analyses the reachability of objects, and propagates properties through the graph of reachable objects. Sălcianu and Rinard report on the results of running their tool on several benchmarks and on Java library code, and compare the static analysis with manual proofs of the purity properties of the code.

One drawback of JPPA is that it performs only whole-program analysis: it requires that a program have a "main" method (and analyses reachability from that method), it requires the entire source code for a program, and it makes the generous assumption that any libraries used do not mutate static fields. Most later analyses aim to be able to cope with isolated fragments of code, rather than entire programs, and in some cases can analyse individual methods on their own.

ReIm and ReImInfer

Huang and Milanova (2012) and Huang et al. (2012) also adopt the JML definition of purity in discussing ReIm, their type system for reference immutability, and

[5] Available from http://jppa.sourceforge.net/, and further described in Salcianu (2006).

ReImInfer, a tool for inferring reference immutability. Huang et al. apply the ReImInfer tool to the task of inferring method purity.

ReIm extends the Java language with additional qualifiers (`mutable`, `readonly` and `polyread`) which can be applied to fields, local variables, formal parameters, and return values.

The associated inference tool, ReImInfer, is reported to scale well to large bodies of code, having $O(n^2)$ worst-case complexity, and in practice, scaling linearly with code size. Unlike JPPA, it can analyse fragments of programs; where the source code is unavailable for particular called methods, ReImInfer conservatively assumes they are impure.

To perform its analysis, ReImInfer first infers when references are "reference immutable" – i.e., when the objects they refer to are not mutated via that reference. It then determines whether the method mutates (directly or indirectly) a static field or any object reachable from a static field. If the parameters and "`self`" object are all "reference immutable", and if no pre-existing static field is mutated, then the method is inferred to be pure.

When Huang et al. tested ReImInfer on a range of Java code-bases, it inferred a significant number of methods as being pure. The proportion of methods inferred as pure ranged from 31% (for JDBM 1.0, a transactional persistence engine for Java) up to 69% (for the code for the `java.lang` package from the Oracle 1.6 Java Development Kit). This is a greater proportion than was identified by the analyses of JPPA (Sălcianu and Rinard, 2005) or JPure (Pearce, 2011), indicating a higher degree of precision than those tools. However, JPure, which we discuss next, has different aims to ReImInfer, and trades precision of analysis for simplicity.

JPure

Like ReIm and ReImInfer, JPure (Pearce, 2011) can perform both inference and checking of pure methods. Purity inference is performed as a source-to-source translation which takes in existing Java code and annotates it with custom Java annotations, such as `@Pure`, when these can be inferred.

Pearce points out that whole-program, interprocedural analysis is time-consuming and thus not suited to day-to-day development. JPure instead uses class hierarchy analysis (CHA, proposed by Dean et al., 1995), a popular and simple method for static analysis of object-oriented languages.

CHA constructs a class inheritance graph for the analysed code, and combines this with the declared types of objects at each point where methods are called on those objects. This allows the analyser to estimate the possible types whose methods could be called, and to construct a conservative estimate of the call graph for the program.

Although not as accurate as some other analyses, CHA is very fast, taking $O(n * t)$ time, where n is the number of call sites, and t is the maximum depth of the class hierarchy.

Once JPure has inferred purity for a body of code – which, although faster than interprocedural analysis, may still be too slow to be run on code which a programmer is making frequent changes to – subsequent checking of altered code can be done very quickly, and in a modular fashion which requires the checker to examine only the body of the method being checked, and the type signatures (including purity annotations) of any methods called.

Like most of the analyses considered thus far, JPure uses a similar notion of "observable purity" to that used in JML by Leavens et al. (2006): it considers a method pure if it does not modify – either directly, or indirectly by calling another method – any pre-existing memory location. Additionally, a pure method is restricted to only being able to call other pure methods (a restriction not imposed by the next system we consider, Joe-E).

Besides the `@Pure` annotation, JPure makes use of two additional annotations in order to correctly identify which objects can be modified: `@Fresh` and `@Local`.

To illustrate the use of the `@Fresh` annotation, consider the situation of using an iterator to iterate over a Java collection. The `next()` method of the iterator is repeatedly called, advancing it over the collection. This call to `next()` is a change to the iterator's state; the `@Fresh` annotation is placed on the method which constructs the iterator, to indicate that the iterator is freshly allocated and can be freely modified.

The `@Local` annotation is a type of ownership indicator: it is used to mark fields which constitute the internal state of an object, and which are never shared with other objects. Any modifications of those objects can therefore be regarded as modifications of the owning object (since no external observer would be able to tell that they are distinct objects), and the semantics of the `@Local` annotation are that if an object is freshly allocated, then the objects it owns should be freshly allocated as well. Locality is transitive: the locality of an object consists of its own primitive fields, and the locality of any objects referenced by a field annotated as `@Local`.

The JPure inference algorithm adopts a "greedy" approach. It starts with the assumption that all methods are pure, and that all fields are local; as methods are inspected, these assumptions are "downgraded" if the inferrer encounters something (for instance, modifications of static state) which invalidates them.

As a test of the JPure system, Pearce analysed the Java standard library, and found that significant portions of the library were pure according to its analysis. It found that 61% of the `java.lang` package was pure.

ReImInfer, on the other hand, was able to identify 69% of the methods as pure; ReImInfer can therefore be seen to be more precise in its identification of pure methods. However, an advantage of JPure's system is its simplicity; this is important in an annotation-based system such as JPure's, since programmers are expected to understand and update the annotations themselves. The use of JPure's three annotations is simple to describe, and the checker operates in a similar manner to the Java compiler, a tool Java programmers are familiar with.

When used to analyse programs other than very small ones, however, Huang and Milanova (2012) report that they found JPure to be fragile.

Joe-E

Joe-E is a security-oriented subset of the Java language created by Mettler et al. (2010).[6] The language was modified by Finifter et al. (2008) to make it "deterministic" and allow it to statically enforce the use of pure methods.

The Joe-E approach to purity differs from the systems previously described. We saw that most researchers have adopted the position that a pure method is one that causes no observable changes in the state of a system: if a Java method modifies a pre-existing static field, then it is not pure. The systems we have described track accesses to pre-existing static fields, in order to ensure they are not modified.

Joe-E, however, requires that pure methods be not only "side-effect free", but also "deterministic", in the sense that their outputs should depend only on their inputs: other than its parameters, or objects it creates itself, a pure method should not be able to observe any data which varies between executions of the program. This prohibits pure methods from accessing mutable static fields, and Joe-E enforces this by requiring all static fields to be declared `final` and made immutable.

The definition of side-effect–freeness is similar to previous approaches, however. It means that a method makes changes only to objects allocated within the method itself – the pre-state of the method is never changed. The Joe-E specification clarifies that this includes performing system output such as writing to disk (Mettler and Wagner, 2008).

Compared to the previous analyses we have considered, the strong version of purity used by Joe-E approaches more closely the functional programming notion of purity. The strong restrictions placed on pure methods arise out of the fact that Joe-E adopts the *object-capability model* of computation (Miller et al., 2001),[7] which imposes stricter constraints than a typical object-oriented language.

The object-capability model was originally developed to assist in designing secure operating systems, but has been generalized for use in secure programming language design. A *capability* in this model is an unforgeable entity that grants its holder the right to perform certain actions. Capabilities are distinguished from *data*, which is immutable (in Java, this would include primitive types such as `int` or `double`). The ability to create side effects, to perform system input or output, or to observe changes in state, are all regarded as capabilities. In an object-capability programming language, the capabilities held by an object are represented by references to other objects.

Several restrictions are imposed on object-capability languages, which distinguish them from more permissive object-oriented languages. References must not

[6] Joe-E is described in more detail in Mettler and Wagner (2008) and Mettler and Wagner (2010).

[7] The object-capability model is described in more detail in Miller (2006) and Miller et al. (2005). A brief summary of the model is also given by Van Roy and Haridi (2004, p 209).

be forgeable, and must not be allowed to point to invalid areas of memory. While these two conditions are standard for memory-safe object-oriented languages such as Java, they are not true of, for instance, C++. In C++, it is quite possible to acquire a pointer to invalid memory (for instance, through a "dangling pointer" to memory which has been de-allocated).

The model requires that an object only be able to acquire a capability by being constructed already holding the capability, or by being passed it by another object. Capabilities are prohibited from being "ambient" (that is, accessible at a scope greater than that of an individual object).

Applying the object-capability model to Java, the result is that Java objects which are immutable, `final`, and which cannot be used to obtain access to any mutable objects are considered merely "data", and it is permissible for this sort of object to be made "ambiently" accessible in static fields. Mutable objects, however, may not be stored in static fields, since this would allow an object to use them without being passed them (Mettler et al., 2010). Joe-E does not permit "benign" side effects in immutable objects.

Although the restrictions imposed by the object-capability model may seem onerous, they do mean that programmers working in such a language have strong guarantees about what individual objects can do. Finifter et al. discuss some of the advantages of applying the object-capability model to a programming language.

Firstly, it allows systems to be designed in accordance with the "Principle of Least Authority". *Authority*, for the purposes of such languages (Mettler and Wagner, 2010),

> "refers to any effects that running code can have other than to perform side-effect-free computations or to throw a virtual machine error due to resource exhaustion. Authority includes not only effects on external resources such as files or network sockets, but also on mutable data structures that are shared with other parts of the program."

Mettler et al. (2010) suggest that "the fewer privileges a component has, the less harm it can do if it misbehaves or runs amok". If an attacker somehow compromises component, they are limited in the amount of damage they can cause.

Secondly, a statically checked object-capability system like Joe-E allows programmers to reason "modularly" about purity, side effects and determinism. Finifter et al. use the term "modular" in a similar sense to Pearce (2011): in order to determine what side effects or indeterminism a method can make use of, a programmer need only inspect the body of the method, and the signatures of any methods it calls.

There are several other differences between Joe-E and the systems considered thus far. Unlike other systems, Joe-E prohibits the use of native code (except in built-in libraries), since native code could cause side effects which the static checker would be unable to analyse. The other systems do not aim to identify or restrict the use of native code.

The analysis used in Joe-E is also much more conservative than other systems: a method is only considered pure if all its parameters are deeply immutable. This means that far fewer methods would be classed as "pure" using Joe-E's analyses compared with previously considered systems. However, the rule for purity is very simple to explain to programmers. As with JPure, Joe-E relies on Java annotations to mark pure methods and immutable objects, and programmers will necessarily have to maintain these. Thus, a less accurate but simple system is preferred over a more accurate but complex one.

Similar to the approach of Leavens et al. in defining JML, purity is considered part of the public contract of a method – the method guarantees that pre-existing objects will not be modified, nor will any system output will be performed, nor any sources of indeterminism observed. Also like Leavens et al., failure due to exhaustion of resources is not considered to be a side effect.

A further point is that in Joe-E, use of the Java "==" and "!=" operators is restricted (Mettler et al., 2010). In the object-capability paradigm, since data is immutable, no distinction need be made between a reference to data and the data itself.

In Java, however, data represented by objects is treated differently from primitive data types, in that the == operator tests for object identity equality, rather than value equality. This can lead to unintuitive results if immutable objects are passed to a pure function, which is supposed always to produce "the same" results, when given "the same" inputs.

What is meant by "the same" is clear for primitive types, which typically represent abstract mathematical objects and are compared by value, as Finifter et al. point out. It is also seems clear that for mutable objects, to say something is "the same" object means that it has the same identity. However, it is less clear what "the same" should mean for immutable object types. Programmers who think of them as value data would expect them to be compared by value, and may be surprised if they are compared for object identity equality.

The solution adopted in Joe-E is to only allow the equality-testing operators to be used on primitive types, in testing whether an object reference is `null`, and on types annotated as "`Equatable`", so that it is clear to programmers when parameters are being treated as values, and when as reference types.

Tschantz and Ernst (2005, p 227) note that statically-checked immutable types are a frequently-requested feature in Java, and more recently a proposal for objects with value semantics has been published as a JDK Enhancement Proposal (Rose, 2012).

Discussion and Conclusion

We have seen that most side-effect and purity analyses for Java adopt an approach similar to level P3 in the categorization of Ierusalimschy and Rodriguez (1995), and we identified reasons for this. Given that the original objection to side effects

is the detrimental effect they have on program comprehension, levels P1 and P2 are too stringent – they prohibit side effects that are in fact benign, and which would prohibit perfectly normal constructs in Java code.

Furthermore, these levels of purity would conflict with the goal, in object-oriented programming, of hiding implementation details and relying solely on the publicly advertised interface of objects.

The approach of most analyses we considered is, therefore, to allow mutation of freshly allocated objects, but prohibit changes to objects on the heap. However, we noted that what constitutes a "change" to an object on the heap in level P3 is not settled. As we discussed, the principles of object-oriented programming suggest that, strictly speaking, only changes to the *observable* state of an object on the heap ought to count as a "change" for the purposes of side effect analysis. Therefore, "benign" side effects should be permissible; but in the interests of simplicity, the analyses which adopted level P3 also prohibited benign changes to objects.

The extent to which "benign" side effects can be permitted in a purity analysis remains an open question. Also open is the question of what effect enforcing purity has on developed code – such as improving comprehensibility, robustness, or security, for instance. Although many authors have argued that enforcing purity is likely to improve these aspects of code, little empirical research in the area has been done.

References

Doris Appleby and Julius J VandeKopple. *Programming languages: paradigm and practice*. McGraw-Hill, New York, 1997.

J. W. Backus, F. L. Bauer, J. Green, C. Katz, J. McCarthy, P. Naur, A. J. Perlis, H. Rutishauser, K. Samelson, B. Vauquois, J. H. Wegstein, A. van Wijngaarden, M. Woodger, and W. L. van der Poel. Revised report on the algorithmic language Algol 60. *Numerische Mathematik*, 4(1):420–453, December 1962. doi: 10.1007/BF01386340. URL http://link.springer.com/article/10.1007/BF01386340.

John P. Banning. An efficient way to find the side effects of procedure calls and the aliases of variables. In *Proceedings of the 6th ACM SIGACT-SIGPLAN symposium on Principles of programming languages*, POPL '79, page 29–41, New York, NY, USA, 1979. ACM. doi: 10.1145/567752.567756. URL http://doi.acm.org/10.1145/567752.567756.

H. H. Bottenbruch and A. A. Grau. On translation of boolean expressions. *Commun. ACM*, 5(7):384–386, July 1962. doi: 10.1145/368273.368414. URL http://doi.acm.org/10.1145/368273.368414.

Timothy Budd. *An Introduction to object-oriented programming*. Addison-Wesley, Boston, 2002.

William R. Cook. On understanding data abstraction, revisited. In *Proceedings of the 24th ACM SIGPLAN Conference on Object Oriented Programming Systems Languages and Applications*, OOPSLA '09, page 557–572, New York, NY, USA, 2009. ACM. ISBN 978-1-60558-766-0. doi: 10.1145/1640089.1640133. URL http://doi.acm.org/10.1145/1640089.1640133.

Jeffrey Dean, David Grove, and Craig Chambers. Optimization of object-oriented programs using static class hierarchy analysis. In Mario Tokoro and Remo Pareschi, editors, *ECOOP'95 — Object-Oriented Programming, 9th European Conference, Åarhus, Denmark, August 7–11, 1995*, number 952 in Lecture Notes in Computer Science, pages 77–101. Springer Berlin Heidelberg, January 1995. URL http://link.springer.com/chapter/10.1007/3-540-49538-X_5.

Matthew Finifter, Adrian Mettler, Naveen Sastry, and David Wagner. Verifiable functional purity in Java. In *Proceedings of the 15th ACM conference on computer and communications security*, CCS '08, pages 161–174, New York, NY, USA, 2008. ACM. doi: 10.1145/1455770.1455793. URL http://doi.acm.org/10.1145/1455770.1455793.

Wei Huang and Ana Milanova. ReImInfer: method purity inference for Java. In *Proceedings of the ACM SIGSOFT 20th International Symposium on the Foundations of Software Engineering*, FSE '12, pages 38:1–38:4, New York, NY, USA, 2012. ACM. doi: 10.1145/2393596.2393640. URL http://doi.acm.org/10.1145/2393596.2393640.

Wei Huang, Ana Milanova, Werner Dietl, and Michael D. Ernst. Reim & ReImInfer: checking and inference of reference immutability and method purity. *SIGPLAN Not.*, 47(10):879–896, October 2012. doi: 10.1145/2398857.2384680. URL http://doi.acm.org/10.1145/2398857.2384680.

R. Ierusalimschy and N. Rodriguez. Side-effect free functions in object-oriented languages. *Computer languages*, 21(3):129–146, 1995. URL http://www.sciencedirect.com/science/article/pii/0096055195000089.

Brian W Kernighan and Rob Pike. *The practice of programming*. Addison-Wesley, Reading, MA, 1999.

Donald E. Knuth. The remaining trouble spots in ALGOL 60. *Commun. ACM*, 10(10):611–618, October 1967. doi: 10.1145/363717.363743. URL http://doi.acm.org/10.1145/363717.363743.

Gary T. Leavens, Albert L. Baker, and Clyde Ruby. Preliminary design of JML: a behavioral interface specification language for Java. *SIGSOFT Softw. Eng. Notes*, 31(3):1–38, May 2006. doi: 10.1145/1127878.1127884. URL http://doi.acm.org/10.1145/1127878.1127884.

Ben Lippmeier. *Type Inference and Optimisation for an Impure World*. PhD, Australian National University, Canberra, ACT, 2009. URL http://cs.anu.edu.au/people/Ben.Lippmeier/project/thesis/thesis-lippmeier-sub.pdf.

B. Liskov and John Guttag. *Abstraction and specification in program development*. MIT Press, 1986.

J. M. Lucassen and D. K. Gifford. Polymorphic effect systems. In *Proceedings of the 15th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '88, page 47–57, New York, NY, USA, 1988. ACM. doi: 10.1145/73560.73564. URL http://doi.acm.org/10.1145/73560.73564.

Adrian Mettler and David Wagner. The Joe-E language specification, version 1.0. Technical Report UCB/EECS-2008-91, EECS Department, University of California, Berkeley, Aug 2008. URL http://www.eecs.berkeley.edu/Pubs/TechRpts/2008/EECS-2008-91.html.

Adrian Mettler and David Wagner. Class properties for security review in an object-capability subset of Java. In *Proceedings of the 5th ACM SIGPLAN Workshop on Programming Languages and Analysis for Security*, PLAS '10, page 7:1–7:7, New York, NY, USA, 2010. ACM. doi: 10.1145/1814217.1814224. URL http://doi.acm.org/10.1145/1814217.1814224.

Adrian Mettler, David Wagner, and Tyler Close. Joe-E: A security-oriented subset of Java. In *Network and Distributed Systems Symposium. Internet Society*, 2010. URL http://www.internetsociety.org/sites/default/files/met.pdf.

Bertrand Meyer. *Object-oriented software construction*. Prentice-Hall International, London, 1988.

Ana Milanova, Atanas Rountev, and Barbara G. Ryder. Parameterized object sensitivity for points-to and side-effect analyses for Java. In *Proceedings of the 2002 ACM SIGSOFT international symposium on Software testing and analysis*, ISSTA '02, page 1–11, New York, NY, USA, 2002. ACM. doi: 10.1145/566172.566174. URL http://doi.acm.org/10.1145/566172.566174.

Mark S. Miller, Chip Morningstar, and Bill Frantz. Capability-based financial instruments. In Yair Frankel, editor, *Financial Cryptography*, number 1962 in Lecture Notes in Computer Science, pages 349–378. Springer Berlin Heidelberg, January 2001. URL http://link.springer.com/chapter/10.1007/3-540-45472-1_24.

Mark S. Miller, Bill Tulloh, and Jonathan S. Shapiro. The structure of authority: Why security is not a separable concern. In Peter Van Roy, editor, *Multi-paradigm Programming in Mozart/Oz*, number 3389 in Lecture Notes in Computer Science, pages 2–20. Springer Berlin Heidelberg, January 2005. URL http://link.springer.com/chapter/10.1007/978-3-540-31845-3_2.

Mark Samuel Miller. *Robust Composition: Towards a Unified Approach to Access Control and Concurrency Control*. PhD, Johns Hopkins University, 2006. URL https://www.cypherpunks.to/erights/talks/thesis/submitted/markm-thesis.pdf.

David Pearce. JPure: a modular purity system for Java. In Jens Knoop, editor, *Compiler Construction*, volume 6601 of *Lecture Notes in Computer Science*, pages 104–123. Springer Berlin/Heidelberg, 2011. URL http://dx.doi.org/10.1007/978-3-642-19861-8_7.

Benjamin C. Pierce. *Types and Programming Languages*. The MIT Press, 1 edition, February 2002.

G. J. Popek, J. J. Horning, B. W. Lampson, J. G. Mitchell, and R. L. London. Notes on the design of euclid. In *Proceedings of an ACM Conference on Language Design for Reliable Software*, page 11–18, New York, NY, USA, 1977. ACM. doi: 10.1145/800022.808307. URL http://doi.acm.org/10.1145/800022.808307.

Anthony Ralston. Side effect. In *Encyclopedia of Computer Science*, page 1573–1574. John Wiley and Sons Ltd., Chichester, UK, 4th edition, 2003. URL http://dl.acm.org/citation.cfm?id=1074100.1074789.

John Rose. JEP 169: Value Objects, December 2012. URL http://openjdk.java.net/jeps/169.

Amr Sabry. What is a purely functional language? *Journal of Functional Programming*, 8(01):1–22, 1998.

Amr Sabry. Side effects. In *Wiley Encyclopedia of Computer Science and Engineering*. John Wiley & Sons, Inc., March 2009. URL http://onlinelibrary.wiley.com/doi/10.1002/9780470050118.ecse370/abstract. DOI: 10.1002/9780470050118.ecse370.

Alexandru Doru Salcianu. *Pointer analysis for Java programs: Novel techniques and applications*. PhD, Massachusetts Institute of Technology, 2006. URL http://dspace.mit.edu/handle/1721.1/38311.

Michael L. Scott. *Programming Language Pragmatics*. Morgan Kaufmann, 3rd edition, 2009. URL http://www.amazon.com/Programming-Language-Pragmatics-Third-Michael/dp/0123745144.

Robert W Sebesta. *Concepts of programming languages*. Benjamin/Cummings, Redwood City, Calif., 3rd edition, 1996.

Robert W Sebesta. *Concepts of programming languages*. Pearson, Boston, 10th edition, 2012.

Alan Snyder. Encapsulation and inheritance in object-oriented programming languages. In *Conference Proceedings on Object-oriented Programming Systems, Languages and Applications*, OOPLSA '86, page 38–45, New York, NY, USA, 1986. ACM. ISBN 0-89791-204-7. doi: 10.1145/28697.28702. URL http://doi.acm.org/10.1145/28697.28702.

Alexandru Sălcianu and Martin Rinard. Purity and side effect analysis for Java programs. In Radhia Cousot, editor, *Verification, Model Checking, and Abstract Interpretation*, volume 3385 of *Lecture Notes in Computer Science*, pages 199–215. Springer Berlin Heidelberg, Berlin, Heidelberg, 2005. doi: 10.1007/978-3-540-30579-8_14.

Matthew S. Tschantz and Michael D. Ernst. Javari: adding reference immutability to Java. *SIGPLAN Not.*, 40(10):211–230, October 2005. doi: 10.1145/1103845.1094828. URL http://doi.acm.org/10.1145/1103845.1094828.

Peter Van Roy and Seif Haridi. *Concepts, techniques, and models of computer programming*. MIT Press, Cambridge, Mass., 2004. ISBN 0262220695 9780262220699.

Brian A. Wichmann. *ALGOL 60 compilation and assessment*. Academic Press, London, 1973. URL http://sw.ccs.bcs.org/CCs/KDF9/Wichmann/book.pdf.

N. Wirth. The programming language Pascal. *Acta Informatica*, 1(1):35–63, March 1971. doi: 10.1007/BF00264291. URL http://link.springer.com/article/10.1007/BF00264291.