The Left Hand of Equals

James Noble
Victoria University of Wellington
New Zealand
kjx@ecs.vuw.ac.nz

Andrew P. Black
Portland State University
U.S.A.
black@cs.pdx.edu

Kim B. Bruce
Pomona College
U.S.A
kim@cs.pomona.edu

Michael Homer
Victoria University of Wellington
New Zealand
mwh@ecs.vuw.ac.nz

Mark S. Miller
Google Inc.
U.S.A.
erights@google.com

Abstract

When is one object equal to another object? While object *identity* is fundamental to object-oriented systems, object *equality*, although tightly intertwined with identity, is harder to pin down. The distinction between identity and equality is reflected in object-oriented languages, almost all of which provide two variants of "equality", while some provide many more. Programmers can usually override at least one of these forms of equality, and can always define their own methods to distinguish their own objects.

This essay takes a reflexive journey through fifty years of identity and equality in object-oriented languages, and ends somewhere we did not expect: a "left-handed" equality relying on trust and grace.

Categories and Subject Descriptors D.3.3 [*Programming Languages*]: Language Constructors and Features—Classes and objects.

Keywords equality, identity, abstraction, object-orientation

Introduction

> *I'll make my report as if I told a story, for I was taught as a child on my homeworld that Truth is a matter of the imagination. The soundest fact may fail or prevail in the story of its telling.*

Ursula Le Guin, The Left Hand of Darkness, (LeGuin 1969)

We began with Simula. This is hard to say now, for all of us who came of age in the golden years of programming language design feel in our bones that the world began with Smalltalk. Even though we know it's not so, we cherish the memories of the dusty underground shelf where the library hid the Smalltalk books, of the Tektronix 4404 Smalltalk machine, equipped with a "cat" as well as a "mouse", and of loading Smalltalk-80 off the Apple-branded floppy disks onto a Lisa. So much romance! Meanwhile, down in the basement machine room, Simula had been chugging along happily on the DECSYSTEM-10 since 1975. That Simula system lacked the sexy graphics of the Lisa and the 4404, but did offer an online debugging facility with breakpoints that has evolved but slightly into the debuggers of today.

We finish with Grace. Or perhaps: we hope to finish with grace, to finish gracefully. Much of our recent professional lives have been occupied with the design of a new object-oriented language — Grace — intended be useful in education (Black et al. 2012). Grace follows in the tradition of Simula, Smalltalk, Self, Basic, and Pascal, mixing in Java, Ruby, Python, Newspeak and many other languages. If this essay has a question, a motivation, or a destination, it is: can we find a Graceful definition of equality, simple enough to explain to novice programmers, but general enough to provide the foundations for serious programming. What should it mean for for two Grace objects to be equal, and what "equals" operators should Grace provide?

Simula

> *Associated with an object there is a unique "object reference" which identifies the object. . . . Two object references X and Y are said to be "identical" if they refer to the same object.*

SIMULA-67 Common Base Standard (1970) (Dahl et al. 1970).

Simula-67 introduced the idea of "object identity", supported by two "reference comparators", == and =/=. We will mostly call this comparison **reference equality** because it tests if two references actually refer to the same object, although we may lapse into using the synonym **object identity**. Simula also includes the Algol-like operators = and /=, which compare primitive objects such as numbers or characters by looking at the values that they contain. We will call this kind of comparison **value equality**. Thus, Simula has two distinct families of equality operators.

Reference Equality

Do two references refer to the same object?

Value Equality

Do two objects contain equal values?

By a **family of operators** we mean a set of operators that use the same concept of equality. Each family consists of an equality operator (such as =) and a inequality operator (such as /=). Simula's reference equality operators (== and =/=) are one family, and its value equality operators (= and <> or /=) are another family. Families get bigger over time: many languages include a unary `hash` operation as a third member of each family. We will pretty much ignore the other operators, and talk about the family in terms of the equality operator and its semantics.

Simula actually goes one step further and distinguishes assignments, using Algol's := for value assignment, and its own :– for reference assignment. Famously, Simula assignments are legal if there is *any* subtype relationship between the left- and right-hand sides: if the type of the right-hand side is a supertype of that of the left, the assignment is checked dynamically.

Simula keeps things mostly straightforward because "There is no value assignment operation for objects" (Dahl et al. 1970). Objects in Simula are accessed by references, so objects always use the reference family operators (with reference equality semantics), while numbers and characters use the value family (with value equality semantics). We say that Simula is "mostly" straightforward, because strings are more complicated, as they are represented by a mutable reference type `text`. Texts behave pretty much like objects, but support *both* reference equality, to distinguish different text objects, and value equality, to determine if two text objects contain the same characters. If `T` and `U` are texts, then "the relations `T=U` and `T=/=U` may both have the value true" (Dahl et al. 1970).

Here is the worm in our garden of Eden.

Smalltalk and Self

> *Equivalence (==) is the test of whether two objects are the same object. Equality (=) is the test of whether two objects represent the same component. The decision as to what it means to be "represent the same component" [sic] is made by the receiver of the message.*
>
> Adele Goldberg and David Robson, Smalltalk: The Language and its Implementation (Goldberg and Robson 1983).

Conceptually, Smalltalk starts from Simula's design, with two families of equality operators. It even writes the core operators the same way: == and =. The operator == is reference equality, testing whether its arguments "are the same object" (Goldberg and Robson 1983). Smalltalk's operator =, though, is not value equality — although it may appear to be so at first glance.

Rather, Smalltalk's = is a dynamically-dispatched method request, so its semantics depends critically on the = method in the receiving object. The semantics of = should be one part of the overall semantics of the abstraction represented by the object, so for this reason we'll call this kind of operation **abstract equality**. Whereas reference equality involves only two objects, abstract equality (and value equality) can involve many more: the objects being compared, and any other objects that are part of those objects' implementations. "(I am large, I contain multitudes.)" (Whitman 1891).

Abstract Equality

Do two objects represent the same abstract value?

Both = and == are nominally implemented as Smalltalk primitives that invoke operations directly: in fact, Smalltalk cheats on ==, compiling it directly as reference equality, so == cannot be redefined in Smalltalk.

Smalltalk makes two significant conceptual advances over Simula. The first is the depth of its object-orientation. Simula is a hybrid language, defined by adding object-oriented features to Algol 60, and retaining many features (control structures, numbers, array, procedures, texts) that are not object-oriented. In contrast, pretty much everything in Smalltalk is an object, and all computation is carried out by dynamically dispatched messages. Second, Smalltalk introduces an encapsulation boundary around each object: an object's fields (instance variables) can be accessed only from within that object.

Self (Ungar and Smith 1991) is a descendant of Smalltalk that eschews classes in favour of cloning and delegation. Like Smalltalk, everything in Self is an object, and all computation proceeds by dynamic dispatch. Also like Smalltalk, Self's encapsulation boundaries surround individual objects (at least in early versions of Self; later versions kept the vis-

ibility definitions but did not enforce them). Self's comparison operators are the == reference equality and = abstract equality of Smalltalk: but unlike Smalltalk, in Self both are dynamically dispatched. Indeed, a major goal of Self's implementation was to demonstrate that a language could perform well even when the lowest level operations (reference equality, control structures, and even arithmetic) are dynamically dispatched (Chambers et al. 1989).

Self's == operation is implemented by a low-level primitive function _Eq: that takes two arguments. Here is the flaw in Self's design: Self's reference equality primitive _Eq: can be applied to any pair of objects at any time, although doing so would be considered extremely bad style.

Lisp and EGAL

Lisps systems have a large number of different equality operations — Common Lisp has at least eight (Steele 1990). Henry Baker addresses the proliferation of Lisp equality operators in his seminal 1993 paper by proposing one equality operator to unify them all:

> *We define a single, computable, primitive equality predicate called EGAL which we show is consistent with the notion of "operational identity" of data structures . . .*
>
> *Our model for object identity distinguishes mutable objects from immutable objects, and mutable components of aggregate objects from immutable components.*
>
> Henry Baker, Equal Rights for Functional Objects . . . (Baker 1993).

EGAL compares mutable objects using reference equality, and "automatically recurse[s] into the components of an immutable object" i.e., compares them with value equality. EGAL is also defined over closures, primarily to support idioms that build objects (Hoyte 2008). The point of EGAL is that, like reference identity, it is a stable comparison: it does not depend on the mutable state of any objects in the program. The key practical difference between EGAL and reference equality is that two immutable objects containing equal values (say, two point objects representing the same location, or two immutable sequences of the same numbers) are indistinguishable under EGAL.

EGAL

Compare mutable objects with reference equality, and immutable objects with value equality.

EGAL does not support abstract equality. Two different representations of the same abstract object (say, the sequence containing (1, 2, 3, 4, 5) and the range 1..5 (Cook 2009)) are not EGAL, whereas Smalltalk's abstract equality could be defined to consider them equal. Baker does suggest that "Systems defining abstract datatypes might consider providing a new "generic" predicate that defaults to EGAL for primitive datatypes, and can be overloaded for user-defined abstract datatypes" but doesn't take this suggestion further.

Two recent languages have adopted EGAL. Clojure (Hickey 2016) uses EGAL for its = operator, along with reference equality (called identical). Pyret (Krishnamurthi et al. 2016) supports three main equality operators: reference equality, written <=> or identical; EGAL written = or equals−always, simplified from Baker's design by eschewing function comparison; and value equality, written =∼ or equals−now). Pyret's design makes clear that EGAL sits between reference and value equality. By default, Pyret objects are compared using EGAL. Programmers can define an _equals method to override the default, but the _equals method is used only when the objects being compared have the same brands, so its ability to implement abstract equality is tightly constrained. We really like Pyret's names though: "equal always" and "equal now" are quite evocative.

Curly Bracket Languages

> *In BETA there is clear distinction between* reference equality *and* value equality . . .
>
> *Note that it is the presence of the symbol* [] *which indicates reference equality instead of value equality*
>
> Ole Lehrmann Madsen, Birger Møller-Pedersen, Kristen Nygaard, Object-Oriented Programming in the BETA Programming Language (Lehrmann Madsen et al. 1993).

C++ was originally designed to combine SIMULA and C, and combines statically dispatched operators with functions and dynamically dispatched methods. C++ chooses between reference equality, value equality, and abstract equality, using its type systems to determine which operation should be invoked (Stroustrup 1991). BETA, Simula's most direct successor, distinguishes reference and value equality based on argument syntax.

Java follows C++ syntactically, although its semantics draws more from Smalltalk. Java's built-in operator == gives reference equality, just like Smalltalk's ==, while an overridable equals(Object) method parallels Smalltalk's abstract =. C# started out by following the essentials of Java, but has become more complicated over time. C# sports an overridable == operator, and an Equals(Object) method, overloaded by an Equals(T) method, which provide abstract equality. A separate ReferenceEquals(x,y) method on a system object tests reference equality.

JavaScript follows Java in syntax, but its semantics are more influenced by Self and Scheme. JavaScript offers the === and Object.is operators that test reference equality for objects, and also a == equality operator that performs

numerous type conversions and is better avoided (Crockford 2008).

Although it is not a curly bracket language syntactically, Python provides primitive reference equality (the **is** operator) and an overridable == operator that can be used to provide abstract equality. Ruby follows Smalltalk's design in many respects. It has an equal? method testing reference equality, and a == method testing abstract equality. Ruby also offers three special purpose comparisons: eql? (used for keys), === (used for case matching), and <=> (for three-way comparison). As in Self, Ruby's reference equality method equal? is a true method request, so it can be overridden, although programmers are advised not to do so!

Scala builds on Java, with a user level abstract equality == defined in terms of an overridable equal method, and a separate eq method for reference equality. Scala also supports the definition of "case classes": objects that are generally immutable and automatically implement equals as value equality.

```
final def == (that: Any): Boolean =
  if (null eq this) null eq that else this equals that
```

Martin Odersky,
The Scala Language Specification Version 2.9

Equality

All animals are equal,
but some animals are more equal than others.

George Orwell == Eric Arthur Blair,
Animal Farm, (Orwell 1945).

We have avoided until now what is perhaps the most important question of all: what is the *intention* of equality? Equality of opportunity or equality of outcome? Should some or all of the object comparison operations possess the mathematical properties of an equivalence operator? What does it mean if we say that two objects are equal?

The Java specification offers a relatively comprehensive definition: the "relation" induced by *all the overridden equals methods in the program* must be reflexive, symmetric, transitive, consistent, and must handle nulls properly.

> public boolean equals(Object obj)
>
> Indicates whether some other object is "equal to" this one.
>
> The equals method implements an equivalence relation on non-null object references:
>
> - It is *reflexive*: for any non-null reference value x, x.equals(x) should return true.
> - It is *symmetric*: for any non-null reference values x and y, x.equals(y) should return true if and only if y.equals(x) returns true.
> - It is *transitive*: for any non-null reference values x, y, and z, if x.equals(y) returns true and y.equals(z) returns true, then x.equals(z) should return true.
> - It is *consistent*: for any non-null reference values x and y, multiple invocations of x.equals(y) consistently return true or consistently return false, provided no information used in equals comparisons on the objects is modified.
> - For any non-null reference value x, x.equals(null) should return false.
>
> ...
>
> **Returns:**
> true if this object is the same as the obj argument; false otherwise.

The Java Platform (Gosling et al. 2005)
(our underlining)

Using this definition, we can see that, although EGAL meets the Java specification, Java permits two objects to be equals when they are not EGAL. This distinction between equals and EGAL is centred on the clause we have underlined: Java equals can depend on the mutable state of the objects being compared. Java permits abstract equal now semantics for its equals and so results can change if one or both of its argument objects are modified — or rather, if any mutable state in any object that happens to be read by an equals method is modified. Two different mutable objects will never be EGAL, but they can be equals in Java. On the other hand, the results of EGAL comparisons are *always* consistent, irrespective of any state mutation: two EGAL objects are equal always.

This kind of definition gives shape to an equality relation, but does not specify how equality affects the rest of the program: what are the consequences of two objects being equal or identical? What are the pragmatic semantics of equality? Tellingly, the Java specification just says that objects are "equal to" each other (with quotations marks in the original) or that the objects are "the same as" each other (our quotes). But what does that mean? What should it mean? Pragmatically, if a program discovers that two objects are equal, what other assumptions should programmers be able to make about those two objects?

Based on work by Horst Reichel (Reichel 1995) and Bart Jacobs (Jacobs 1996), William Cook argues for object equivalence based on bisimulation: "if two objects simulate each other ... they are equivalent" (Cook 2009). David Ungar has described this more pragmatically, as the desired semantics of Self's identity relation ==:

> *Basically, given two references, A and B, A == B implies that for any message M, you could send M to A or send M to B and there would be no observable change in the future response to messages of the system.*

David Ungar, personal communication.

This is a very strong condition. To repurpose Pyret's terminology: under this semantics, if two objects are equal at

some point in the program, they will always be equal at any other time. This **always equal** semantics can be implemented by reference equality, obviously, but there are other kinds of equality can also meet this condition, notably EGAL, and an abstract equality method could be implemented that satisfies this condition.

On the other hand many kinds of equality do *not* imply bisimulation, including Java's `equals`, Smalltalk and Self's $=$, and Ruby's $==$. These meet a weaker condition: again stealing Pyret's terminology, the objects are **equal now**, but may not be equal in the future (and may not have been equal in the past). In terms of messages, we can say:

> *Given two references, A and B, A is equal now to B implies that if you sent M to A and received R as the result, or if you sent M to B and received S as the result, R and S would be equal now.*

If two objects are always equal, they must also be equal now: the reverse is not the case.

Object-Orientation

> ***O is for Object,***
> *which is the granddaddy of all soap bubbles.*
>
> Brian Alexander, ABC's for object-gifted children, (Alexander 1992)

If we am to talk about equality in object-oriented languages, we must also talk about objects, and, in particular, how object-oriented programming differs from other approaches to programming. There are many definitions of object-orientation, focusing on different aspects of programming and design (Noble 2009). The problem with traditional extensional definitions such as Grady Booch's "*An object has state, behaviour, and identity*" (Booch 1994) is that they beg the question: if identity is built into your definition of object-orientation then there is little left to ruminate about.

Ralph Johnson has a three-fold definition which captures the semiotic, conceptual, and technical aspects of object-orientation rather well (Johnson 2008, 2007; Noble 2009):

> *I explain three views of OO programming. The Scandinavian view is that an OO system is one whose creators realise that programming is modelling. The mystical view is that an OO system is one that is built out of objects that communicate by sending messages to each other, and computation is the messages flying from object to object. The software engineering view is that an OO system is one that supports data abstraction, polymorphism by late-binding of function calls, and inheritance.*

Rather more recently, William Cook has characterised the key structural feature of object-orientation in terms of the *autognostic principle* (Cook 2009):

> An object can only access other objects through their public interfaces.
>
> *Autognosis means 'self knowledge'. An autognostic object can only have detailed knowledge of itself. All other objects are abstract. The converse is quite useful: any programming model that allows inspection of the representation of more than one abstraction at a time is not object-oriented.*
>
> William Cook, On Understanding Data Abstraction, Revisited, (Cook 2009).

Autognosis takes a per-object encapsulation boundary to its logical conclusion: an object can have detailed knowledge — access to the internal representation — only of itself: other objects can be known only via their public interfaces.

Cook lays out the implications of autognosis: increasing flexibility while preventing optimisations of complex operations that require access to the representations of multiple objects. One object can be replaced by another object so long as the replacement object supports the original object's public interface: Jonathan Aldrich has explained how this allows large systems to be built by combining interoperable extensions (Aldrich 2013).

Cook singles out Microsoft's COM for praise, because there is "no built-in notion of object equality", and advises Java programmers who wish to adopt a purely object-oriented system to avoid reference equality (Java's $==$), `instanceof` tests on classes, and using classes as types, because these idioms couple the uses of an object to its implementation, thus subverting its public interface. Furthermore, because objects' instance fields cannot appear in Java interfaces, adopting this discipline changes the program's encapsulation boundaries from per class (as in C++ and Java) to per object (as in Smalltalk, Self, or Newspeak).

A key argument of Cook's essay is that this difference in encapsulation stems from a primordial difference between objects, (build using procedural abstraction based, modelled by the untyped lambda calculus) and abstract data types (build using type abstraction, modelled by the typed lambda calculus). This distinction has not always been clearly understood. For example, a key part of Bertrand Meyer's definition of object-oriented programming is that objects, classes, and abstract data types are tightly related:

> *A class is a software element describing an abstract data type...*
>
> *An abstract data type is a set of objects defined by the list of operations, or features, applicable to these objects, and the properties of these operations.*
>
> Bertrand Meyer, Object-Oriented Software Construction, (Meyer 1997).

Meyer's Eiffel is one of the few languages that supports both encapsulation boundaries simultaneously: individual method and fields can be accessible to every class in the system, to a specified list of classes, to just their defining class, or restricted to just the current instance. Conceptually, abstract data types or the SIMULA-derived encapsulation boundary make abstract equality easy: objects of the class represent instances of the abstract data type; an abstraction function maps instances' representations into the corresponding abstract values of the type; the semantics of equality over the abstract type is precisely defined (e.g. by reduction to a canonical form); a procedure implementing that equality can be written relying on access to the representations of both instances of the abstract type being compared.

The remainder of this essay tries to answer the complementary question. What can we do to provide an equality operator for a pure, autognostic object-oriented language? What is the most we can do in a world of individually encapsulated objects, or what is the least we can get away with and still write programs? (Spoiler: Grace, which we are attempting to design, has adopted Smalltalk's per-instance encapsulation boundary, rather than Simula's per class encapsulation).

Left-handed Equals

$$\Gamma N\Omega\Theta I \quad \Sigma A\Upsilon TON$$

Know Thyself.
Inscription of Apollo's temple at Delphi.

At this point we return to Cook's definition: an autognostic object can only have detailed knowledge of itself, while treating every other object as abstract, so their representations cannot be inspected. If identity is part of an object's representation — or at least, the representations of the abstractions modelled by the object — then one object cannot inspect the identity of any other object. Certainly there can be no mandate for a third-party object to be able to compare two objects' identity.

This is why Cook's rules for pure object-oriented programming in Java rule out Java's == reference equality operator. As Andrew Black puts it: "*An application . . . may have its own notion of identity that differs from the underlying system*" (Black 1993). To see why, consider the example of a simple string object being used to e.g. represent an access key in a distributed object-oriented system. An abstract equality could compare strings by looking at their contents, presumably character by character: so clients would be unable to determine if two strings were actually identical or two different string objects that were abstractly equal. A reference equality comparison (such as Java's third-party identity test) would allow clients to distinguish between identical and abstractly equal strings, so allowing the distinction to leak through the public interface of the abstraction. This distinction could potentially open a covert channel e.g. allowing clients to detect whether a key is to a local or a remote resource. A runtime system may need to take extra effort to preserve reference equality semantics, e.g. ensuring that strings from remote machines are always accessed via remote proxies, rather than just copying their contents.

Exposing object identity also makes transparent proxies more difficult to build. Encapsulators (Pascoe 1986) and transparent forwarders (McCullough 1987), first developed in Smalltalk, can monitor objects, check invariants, or provide access control (Gamma et al. 1994) without modifying the underlying object. Racket's chaperones and impersonators (Strickland et al. 2012), and Javascript's proxies (Cutsem and Miller 2010, 2013) offer similar facilities. All these techniques depend on an object's clients being unable to distinguish between the identity of the naked underlying object, and that object wrapped within an encapsulator (proxy, chaperone, impersonator). Direct access to the identity (or the class) of another object allows encapsulators to be detected directly.

If autognosis is "self-knowledge" then it seems, however, that there is one special case of reference equality which can be admitted in an object-oriented system, and that does not break abstractions: when an object is compared with *itself*. For the Java inclined, imagine that you can only write a primitive reference equality with this on the left-hand side "this == other" . In Smalltalk, comparing an object with itself looks even better "self == other" (although in Self we have the inscrutable "== other").

Self Reference Equality

Does another reference refer to self?

Arguably the single most important use of reference equality in Java code is the definition of the equals method in java.lang.Object, defining the default value comparison for objects in terms of reference equality. Here, at least, the call to == is autognostic:

```
public boolean equals(Object obj) {
    return (this == obj);
}
```

exactly because "this == other" is a self-reference check: an object determining if some other reference refers to itself (in Java, to "`this`"). Any invocation of Java's "==" which is not testing against a literal "`this`" is not autognostic.

We will call the (autognostic) self reference equality test "refEqualSelf". In Java, refEqualSelf can be defined with exactly the same code as above:

```
public boolean refEqualsSelf(Object obj) {
    return (this == obj);
}
```

the crucial point being that the binary == is converted to a unary operator, effectively by currying.

There are no doubt other uses of == in Java for testing the equality of value types: primitive ints, floats, and so on. These objects are primitive immutable values: here == embodies primitive value equality, rather than reference equality. The key question is: what about *other* uses of == for reference equality of objects, in particular non-autognostic uses of ==, where the left argument *isn't* self?

From Autognosis to Equals

By analogy with the default definition of equals in Java, we can lift an autognostic reference equality test (i.e. refEqualSelf) into an object's interface by defining a leftEquals method as follows:

```
method leftEquals(other : Object) {refEqualSelf(other)}
```

Clients would call leftEquals method instead of using == on objects. Will this do, or do we need a full-strength identity comparison predicate?

We think that this *will* do, or rather, that it will be just enough. For a start, Self's and Ruby's reference equality operators are defined in pretty much this way. Ruby's equal? runs a primitive method, and although Self's definition bottoms out in the _Eq: primitive, that primitive is only ever used autognostically. In fact it's only used once, in the beautifully gnomic definition of the nonprimitive, dynamically dispatched == method:

```
== x = ( _Eq: x )
```

Couldn't something like leftEquals work just as well in Java or other object-oriented languages? The big difference between leftEquals and built in primitives like Java's ==, Python's is or indeed Self's _Eq: is that the built in primitives are *trustworthy*. Being supported by the language implementation, primitives are guaranteed to give a result that accords with reality, or at least with the conceptual model and semantics of the language. Once we switch to using a method like any other, we lose this guarantee.

Java programmers may argue that they could fix the problem with leftEquals by making it a final method on class Object, so every object would be forced to offer this implementation — this is Ruby's convention. Pragmatically, this will be difficult to enforce in an open object-oriented system where we do not control the provenance of every object.

From Cook's perspective, of course, this is no longer object-oriented: it means that every object now knows (and can depend upon) a crucial fact about the *implementation* of every other object: that is, the precise code in the body of the leftEquals method. Conceptually, we're just back again to SIMULA: a special syntax for reference equality baked into the language.

Trust

the truth is not an obstacle for someone such as me,
she said
because you see we all create our own reality
and if a problem should arise
the best thing you can say is
don't worry, be happy, and have a nice day

MC 900 ft Jesus, Truth is Out of Style, (MC 900 Ft. Jesus 1989)

Drossopoulou and Noble's *Logic of Risk and Trust* can be brought to bear on this question (Drossopoulou et al. 2015). Rather than constraining the implementation of leftEquals to behave in a particular way (to meet a particular specification, formal or informal) programmers can reason explicitly about the trust relationships in their programs: which objects are trusted, and which objects are not. A program may trust only the objects that it has created directly, or all the objects in the same process, or all the objects on the same host, or perhaps all the objects used to access a business's suppliers and none of the objects used to access the business's customers. Drossopoulou and Noble write "o **obeys** $\mathcal{S}$" to represent an assumption that an object o conforms to a specification $\mathcal{S}$. Their contribution is that this trust is an assumption, not an assertion: **obeys** does not mean that o conforms to that specification, but that we will proceed as if it did meet the specification. In this way, **obeys** supports hypothetical reasoning about the behaviour of code under different trust assumptions. Moreover, if we are sure that particular object can be trusted, then we can be sure that method requests on that object will meet their specification, whether that request is for leftEquals or any other method.

(This may seem trivial, motherhood-and-apple-pie, especially one baked by Barbara Liskov: but it is not. Liskov's substitution principle aims to ensure that all objects meet their specification: **obeys** handles cases where they do not.)

As Cook might put it: the question of whether an object obeys its specification or not is ultimately a question about *the implementation of that object*: when we encapsulate implementations, we also encapsulate their correctness.

The very asymmetry of object-orientation (Aldrich 2013) is what comes to our rescue: left-handed equals is left-handed for a reason. When we write a.leftEquals(b) what we mean is: if a **obeys** $\mathcal{I}$ (where $\mathcal{I}$ is a specification of the leftEquals operation) then we can trust that the result is accurate. On the other hand, if a doesn't obey the specification, any result whatsoever could be returned. What this means in practice is that if you're trying to compare two objects for identity, make sure the receiver, i.e. the *left hand argument* of leftEquals is trusted, whether or not the right hand argument is.

An immediate consequence of this style of reasoning is that leftEquals does *not* provide an equivalence relation in the presence of just one object that does not obey the

specification. Imagine a perverse object that claims to be equal to any other:

```
def perverse = object {
 method leftEquals(other : Object) → Boolean { true }
}
```

writing perverse.leftEquals(o) will always return true, unlike the symmetric call, o.leftEquals(perverse), which will return false for any trustworthy object o that isn't actually identical to perverse.

If you know (or are willing to assume) that you can trust o, then leftEquals can give you just as much information as a proper symmetric equality comparison: that the two objects are (or are not) equal. If you're not willing to trust o, then you can't expect to learn anything by requesting a method on that object, and in particular you cannot trust the result (so you'd probably choose not to make the request in the first place). If you assume you can trust o and o turns out to be untrustworthy, then your program hit a serious bug, or an attempt to undermine the system.

Drossopoulou and Noble go on to show how a combination of the **obeys** predicate and conditional reasoning can be used to reason further about trust. We can write a specification such that if "a.leftEquals(b)" does return true, we choose to trust b as much as we trust a (they'd write something like "a **obeys** $\mathcal{I} \longrightarrow$ b **obeys** $\mathcal{I}$"). It's important you are not blind about the objects upon which you request methods.

Alternatively we can recover limited equivalence relations by limiting the domain. Rather than building one relation over *all* objects ("any non-null object values" in the Java Specification) we consider only a given set of trustworthy objects: within that set, leftEquals can be an equivalence.

Points and ColouredPoints

It's well known that inheritance and subtyping can also cause an equality relation to break symmetry (Odersky et al. 2009). Consider the paradigmatic Point type:

```
type Point = interface {
 x → Number
 y → Number
 leftEquals(o : Object) → Boolean
}
```

and implementing class:

```
class point(x′, y′) → Point {
   method x { x′ }
   method y { y′ }
   method leftEquals(other : Object) {
     match (other)
       case { p : Point →
              (x.leftEquals(p.x)) && (y.leftEquals(p.y)) }
       case { _ → false }
   }
}
```

the leftEquals method first tests to see if its argument's public interface conforms to the Point type. If the object is at least of the type Point the method then compares coordinates, and this comparison is type safe because we now know that the other is a Point. If the type test fails, we return false. This type test does not breach Cook's autognosis principle, precisely because it inspects only the public interface of the other object — and a program may access any number of objects via their public interfaces. (Testing another object's class would expose the other object's implementation, breaking instance encapsulation and reducing polymorphism.) Here, leftEquals over objects created from the point class will be an equivalence relation.

Then, because this is the traditional point example, we consider a subtype and a subclass:

```
type ColouredPoint = Point & interface {
 colour → Colour
}


class colouredPoint(x′, y′, c′) → ColouredPoint {
   inherit point(x′, y′)
   method colour { c′ }
   method leftEquals(other : Object) {
     match (other)
       case { cp : ColouredPoint →
                (x.leftEquals(cp.x)) &&
                (y.leftEquals(cp.y)) &&
                (colour.leftEquals(cp.colour))}
       case { _ → false }
   }
}
```

where the subtype just requires an additional colour method and the subclass implements that method, and overrides leftEquals to compare colours. Again, leftEquals over objects created by colouredPoint will be an equivalence relation, but mixing points and coloured points will break symmetry: a point can consider itself equal to a coloured point at the same coordinates, but a coloured point will never consider itself equal to a point.

There are at least two ways to recover symmetry here should that prove necessary. If colours aren't important, then colouredPoint can just inherit the equality method from point, unchanged, so the Point type, that is, the Point public interface is enough to determine equality. Alternatively we could add a default colour into the point class and rely only on the ColouredPoint public interface. Both of these options are preferable to a third alternative: adding some kind of publicly visible tag into the Point interface and making a choice based on the return value of the tag method.

Collection

Let's consider a slightly larger example: some kind of collection, with just two requests add(element) which adds an element to the collection, and contains(other) which checks if

the collection contains that element. That specification looks something like this:

```
type Collection = interface {
  method contains(o : Object) → Boolean //for any o
  method add(a : Object) → Done //must trust a
}
```

The point here is that the arguments to the add and contains methods, although the same type, have different trust assumptions. Add (obviously) adds element to the collection: if the collection is to be trustworthy we must be able to trust the elements we store inside it, at least enough to trust their equals methods (o **obeys** $\mathcal{I}$). On the other hand, we do **not** need to trust that any other object handed into to the collection obeys the specification.

Looking at a potential implementation of this method explains why the asymmetry in the specification is feasible:

```
method contains (other : Object ) → Boolean {
  for (1 .. size) do { index →
      if (contents.at(index).leftEquals(other))
        then {return true} }
  return false
}
```

We iterate through a backing collection contents, calling leftEquals *only* on the objects in the collection, that is, objects we have already assumed we can trust. We never call methods on the other objects because we don't trust them — although collections elements may call other methods on the other objects as part of implementing leftEquals. It is the responsibility of the collection's clients to add to the collection only objects that they trust. If an object (like perverse above) is added which does not in fact meet its specification, then the collection will break, in this case, answering that it contains any other object passed to it, whether or not they have been added.

Grant Matcher

The Grant Matcher (Miller 1998) is an exemplar problem developed in the object-capability community to explore the requirement for a reference equality test in a secure distributed object-oriented system. The problem is that two people, Alice and Dana, agree to match each others' donation to a charity. Alice and Dana don't trust each other, but are willing to trust a third party grant matcher. Alice and Dana each supply the grant matcher with their donation (a money object) and an object denoting their idea of the agreed charity.

The puzzle is to write a grant matcher that avoids fraud, notably fraud by one person passing in a fake charity. A charity is a simple object that can accept a donation:

```
type Charity = interface {
  accept( donation : Money ) → Done
}
```

Writing a grant matcher that uses a primitive object identity is easy. Here's a match method that does the match, relying on a system.referenceEquals primitive like C#'s:

```
method match (aliceDonation : Money,
              aliceCharity : Charity,
              danaDonation : Money,
              danaCharity : Charity) → Boolean {
  if ((aliceDonation.amount
         .leftEquals(danaDonation.amount)) &&
      system.referenceEquals(aliceCharity, danaCharity))
    then {aliceCharity.accept(aliceDonation)
          danaCharity.accept(danaDonation)
          return true}
    else {return false}
}
```

This method checks that the donated amounts are the same, that the charities are the same object, and if so makes the donation. (Miller presents a Java implementation (Miller 1998) that deals correctly with concurrency, aliasing, transactions, and escrow; we ignore those concerns here to focus on equality).

Can we achieve the same thing using an abstract object-oriented left-handed equality comparison, like Java's equals or Ruby's ==? Most of the code would be the same, but the test changes to:

```
if ((aliceDonation.amount
       .leftEquals(danaDonation.amount)) &&
    aliceCharity.leftEquals(danaCharity))
```

The catch is that this design now permits a "man in the middle" attack (Miller 1998). A fake charity can masquerade as a real charity by delegating its implementation of leftEquals to a real charity, and then steal the money when given a donation:

```
class fakeCharity → Charity {
  def underlyingCharity : Charity = ...
  def backPocket : Account = ...
  method leftEquals(other : Object)
          { underlyingCharity.equals(other) }
  method hash { underlyingCharity.hash }
  method accept(donation : Money)
          { backPocket.accept(donation) }
}
```

If Alice passes a fake charity into a grant matcher implemented with leftEquals, she can steal Dana's money, because rather than delegating the accept method to the underlying charity it goes straight into Alice's backPocket. (Alice can steal money even without an underlying real charity by writing a perverse charity that always returns "true" from equals — provided no-one calls hash)

The problem arises because the leftEquals call is left-handed: all it can do is determine if aliceCharity is willing to believe (or to pretend) that it is "the same as" danaCharity. This relation is neither symmetric nor transitive. If a request such as aliceCharity.equals(danaCharity) returns true,

then Alice's charity (and by extension, Alice) is willing to go ahead. (Alice indicated she was willing to go ahead when she passed her charity into the grant matcher.)

Formally, we hypothesise that if a charity's leftEquals method returns true, then **self** and the method argument other are the same object (reference equality) and as a result, the other object will obey its specification. This is all subject to the constraint that the left-hand purse is itself trustworthy: aliceCharity **obeys** $\mathcal{C} \longrightarrow$ (aliceCharity $\equiv$ danaCharity) $\wedge$ (danaCharity **obeys** $\mathcal{C}$).

We can (almost) resolve this by restoring symmetry: we must also ask danaCharity if it considers aliceCharity acceptable: if Dana's charity is willing to go ahead, then presumably so is Dana. The change to the condition is almost trivial:

```
if ((aliceDonation.amount
        .leftEquals(danaDonation.amount)) &&
    aliceCharity.leftEquals(danaCharity) &&
    danaCharity.leftEquals(aliceCharity))
```

and now the transaction will only proceed if both charities agree. This ensures that no real charity can be spoofed by a fake charity. Where this solution differs from a system-wide third party identity primitive like system.referenceEquals is that a referenceEquals test can also detect when both sides are trying to cheat. Imagine both Alice and Dana give fake charities to the grant matcher. With a reference equality comparison, the grant matcher can detect that the two charities are different and abort the transaction. With a left-handed equality comparison, each fake charity will accept the other, and so the transaction will complete, with each charity stealing its own donation.

Reflection

The other advantage of built in reference equality operators is that they are guaranteed to work on any object. A programmer defined abstract equality method could be buggy, and in some languages (like Self and JavaScript) some objects can have no methods whatsoever. An inspector or debugger may need to handle and manipulate these kind of objects: to put them into collections, checked for equality or/and identity, etc. This again is another advantage of the abstraction-breaking complicit in system-wide equality operators.

Gilad Bracha and Dave Ungar's model of reflection addresses all these issues (Bracha and Ungar 2004). A separable component of a language can offer privileged access to interact with any kind of object, however buggy or minimal, by supplying mirror objects that act as proxies for their reflectees. Because mirror objects are created by a trusted reflexion subsystem (say by calling reflection.mirror(o) to reflect on o) programmers can assume that the mirrors will implement their API correctly, without having to know anything about the implementation of that API. The mirror API can include methods (say reflecteeEquals) that performs reference comparison on their reflectees.

As Ungar wrote in a code comment in Self's core definition of its reference equality operator "==":

> == and !== should usually be avoided; if you really care about object identities then you should probably be using mirrors, since object identity is a reflective concept.

Reflection can offer yet another solution to the grant matcher problem. We create two mirrors, one for each charity, and then use the reflecteeEquals method on the mirrors to ask if the those mirrors are reflecting on the same object, i.e., that the charities are the same.

```
method match (aliceDonation : Money,
              aliceCharity : Charity,
              danaDonation : Money,
              danaCharity : Charity) → Boolean {
  def aliceCharityMirror = reflection.mirror(aliceCharity)
  def danaCharityMirror = reflection.mirror(danaCharity)
  if ((aliceDonation.amount
          .leftEquals(danaDonation.amount)) &&
      (aliceCharityMirror
          .reflecteeEquals(danaCharityMirror)))
    then {aliceCharity.accept(aliceDonation)
          danaCharity.accept(danaDonation)
          return true}
    else {return false}
}
```

The properties of this design are essentially the same as using reference equality directly, with an intrinsic reflection subsystem serving as a trusted third party able to verify objects' reference equality.

Reflective Equality

Do two mirrors reflect on the same object?

Grace

> *"The only reasonable numbers are zero, one, and infinity"*
>
> Bruce MacLennan (MacLennan 1995)

To demonstrate how these ideas can come together into a coherent programming language, we will use them to redesign equality support in Grace, a new educational object-oriented programming language (Black et al. 2012). Grace aims to remove "inessential difficulties" from programming, so this design needs to be simple, yet sufficient to cover all the cases we have discussed in this essay.

One Equals Operator

The first question is how many equality operators (or rather, equality operator families) should a language make generally available? For an autognostic object-oriented language,

that means these operators can be expected to be part of the public interface of most if not every object. As we've seen, most object-oriented languages provide at least two equality operators, typically reference equality and abstract equality, and many provide many more.

Having more operators should arguably make it easier for programmers to express the equality semantics that they need, but comes at the cost of programmers having to work out which operator to choose to compare objects. Programs would also have to support all the equality operators in every object. Given Grace's goals, we hope that just one family of equality operators will be enough, assuming that the core operator is well chosen.

Programmers can of course implement an infinity of families of equals operators if that makes sense in their domain — testing on keys, on values, implementation, abstraction — we've already met the cornucopia of equals functions supplied by Lisp. Some programs, especially those modelling complex domains or with complex optimised implementations, may well need many equality operators making many fine distinctions. Nothing in this design prevents programmers defining their own equality operators — the question we are considering is: how many operators should a language make generally available, that is, how many operators must every class implement? For an educational language, parsimony beats munificence: understanding the semantics of that single operator, how to use it, and how to implement it will be more than enough in a first or second programming course.

Autognostic, Abstract, Left-Handed Equals

For reasons already discussed at length in this essay, the equality operator should be autognostic — with direct access to only one object's representation (**self**) from the inside, and all other objects only via their public interfaces.

This means that equality will just be a method request leftEquals(other) that compares its receiver and argument objects, dynamically dispatched like any other request. Left-handed equals as a normal method implies that it can be overridden by the receiver, giving us abstract equality semantics. Where necessary, programmers can and should provide their own definitions of leftEquals. This solves the problems of value equality operators being unable to compare different implementations of the same interface: programmers can write appropriate comparisons for their abstractions. This choice also means that this operator will not form an equivalence relation — although particular implementations may provide an equivalence relation over some subset of all objects. But because leftEquals is not symmetric, programmers have to care about which object is on the left, and which on the right.

Just a left-handed reference equality operator isn't quite enough. Programmers need to be able to implement it: the recursion needs a base case. To maintain autognosticism, such an implementation can only consider the object itself. A self reference equality operator can fulfil this role: it must be confidential (to ensure it can only be called from within its defining object) and primitive (implemented by the underlying platform) — as discussed above.

Time and Eternity

The next design choice is what the implied consequences of two objects being equal should be. Implied consequences, because an abstract equals method can be overridden, so meeting this guarantee depends on the way particular abstractions choose to interpret equality.

The two main choices are always equals, or equals now. If we had two equality operators, we could have both. But we would rather have only one operator; this keeps the design simpler, and means that novices don't have to worry about which equals they should use.

For immutable objects, there is no observable difference between equals now and always equal: this is Baker's EGAL argument.

Similarly, mutable, distinguishable objects that model phenomena in the domain of the program will need object identity for their modelling: this is also an always equals relationship and (self) reference equality will do the trick.

The other main use-case for equality is the collections library: mutable collections in particular. All collections rely heavily on equality. Implementors of hashed or indexed collections need always equal semantics because otherwise objects would have to be reindexed whenever their contents change. On the other hand, equals now semantics seem essential to clients of mutable collections: with equals always semantics, two different mutable collections can never be equal to each other even when they contain exactly the same elements. Either every mutable collection would have to be converted into an immutable collection before comparison, or more likely, equals now semantics would be reintroduced as a second equality operator in the collections API (but, perhaps, not for all objects, unless we conclude two operators really are necessary overall).

Since mutable collections seem integral to Smalltalk-style object-oriented programming, our design explicitly opts for equals now semantics. This has the unfortunate side effect of potentially breaking collections that really need stable equality, hashes, or even comparisons between objects. We are willing to accept that risk for three reasons. First, because once we have opted for an abstract, programmable equality operator there is no guarantee it will be implemented correctly anyway. Second, because always equal semantics is a permissible implementation of equals now, so programmers who need the tighter semantics can adopt it anyway. Third, because an empirical study showed that in Java at least, this problem rarely arises in practice (where "rarely" means that the study was unable to find this problem (Nelson et al. 2010)).

Reflective Equality

Again, for the reasons we have described above, Grace's existing reflective mirrors should be extended to support an equals operator that gives reflective equality (i.e. reference equality on the objects reflected in the mirror). We see how this spoils the claim to have only one family of equals operations: we are effectively squeezing a non-autognostic primitive reference equality operator in through the back door. For example, collections that need a stable always equal relationship can use reflection to get it.

The key advantage, to us, of this back door solution is precisely that it is a back door: the language itself and its core libraries still have only one general equal operator. Reflection is the underlying breach of autognosticism, because it opens up reflected objects' implementations anyway: once we've gone that far it seems churlish not to support reflective equality.

Value Equality

What support, if any, should an object-oriented language provide for value equality? If objects are truly autognostic, then one answer is none: objects are only accessible behind a public interface. Value equality compares the representations of two objects, effectively reducing objects to their representations and nothing more. But what if this is all you need? What about the simple cases, the Wirthian sum-of-product concrete data types introduced in Pascal and COBOL and adopted by most functional programming languages ever since — *algebraic* data types (abbreviated "ADT") rather than *abstract* data types (also abbreviated "ADT") (Cook 2009). Value equality seems a good fit for these kind of objects.

Most object-oriented languages do not support value equality, although Scala's case classes are a notable exception. We can extend our design with a publicFieldsEqual primitive that acts as if the programmer had written a method that compares all the public fields of another object with its own public fields. This primitive remains autognostic because it needs only consult the public interface of an object to access its public fields.

Public Field Equality

Are two objects' public fields equal?

This is the design decision we are still on the fence about.

Summary

> *The story is not all mine, nor told by me alone. Indeed I am not sure whose story it is.*
>
> Ursula Le Guin, The Left Hand of Darkness, (LeGuin 1969)

So this is where we finish up — not in the sense of stopping, not yet, but in the sense that this is where we've got to so far:

- a x.leftEquals(y) request, understood by most if not all objects;
- a request that is the negation of leftEquals(y)
- a refEqualsSelf(y) primitive method, inherited or otherwise available to all objects;
- default definitions of leftEquals(_) (and its negation) in terms of refEqualsSelf.
- a reflecteeEquals(otherMirror) request as part of a reflection system, (if there is one).
- a publicFieldsEquals(t) primitive method, inherited or otherwise available to all objects;

Sometimes a little code is worth a screenful of bullet points:

```
type Object = interface {
  == (other : Object) → Boolean
  != (other : Object) → Boolean
  hash → Boolean
  ...
}

trait graceObject {
  method refEqualsSelf(other : Object)
    is primitive, confidential { }
  method == (other : Object) {refEqualsSelf(other)}
  method != (other : Object) {! (self == other)}
  ...
}

trait graceValue {
  use graceObject
  method publicFieldsEquals(other : Object)
    is primitive, confidential { }
  method == (other : Object) {publicFieldsEquals(other)}
  ...
}


type Reflection = interface {
  reflect(reflectee : Object) → ObjectMirror
  ...
}

type ObjectMirror = Object & interface {
  ...
}
```

Note that we have renamed a few things in the code: leftEquals has become ==, and to show the complete family we include its inverse != and hash. (We did consider not including the inverse !=; but apart from being willfully perverse to disregard every language design precedent, it is particularly perverse when the interface has to include hash anyway.)

We also moved reflecteeEquals to be == on the mirrors. If you've survived this long, you should know not to ask how that is implemented: it really doesn't matter if the reflection subsystem ensures there is just one unique mirror object created for each reflectee (so == is reference equality), or if there can be many mirrors for each reflectee (and == compares reflectees in some other abstract way). A program using the reflection API has no way to tell the difference.

This seems to be is the most we can do in a world of individually encapsulated autognostic objects, and the least we can get away with.

Conclusion

> *Light is the left hand of darkness,*
> *and darkness the right hand of light.*
>
> Ursula Le Guin, The Left Hand of Darkness, (LeGuin 1969)

Ursula K. LeGuin's subtle and celebrated feminist novel "The Left Hand of Darkness" (LeGuin 1969) tells of Genly Ai, an emissary to the planet Gethen. Written mostly from Genly's perspective, the novel includes quotations from Gethen's myths and legends, and excerpts from Ai's reports home. Ai must navigate the Gethenians' androgynous mutable sexuality (as an immutable biological male, Ai is seen as perverse by the Gethenians) and their resulting culture based on status and equality.

Object identity, reference equality, and value equality have been in object-oriented programming languages since SIMULA. Encapsulation, autognosis, coming from Smalltalk, is perhaps the most important principle of all. Self demonstrated that abstract equality comparisons (and many control structures) could be implemented solely as method requests, without any special cases, with a negligible runtime cost. Ruby shows that this kind of design can be practical in a scripting language today, with much less implementation effort. Let's stop contorting languages to run on a thirty years old Lisa, and take advantage of today's and tomorrow's machines to make languages that can be more straightforward, more simple, more trustworthy, and more graceful.

Acknowledgements

We thank Sophia Drossopoulou for many discussions on these topics, and William Cook, Shriram Krishnamurthi, Joe Gibbs Politz, and the anonymous reviewers for their comments. This work was supported in part by a James Cook Fellowship and by the Royal Society of New Zealand Marsden Fund.

References

J. Aldrich. The power of interoperability: why objects are inevitable. In *Onward!*, pages 101–116, 2013.

B. Alexander. ABC's for object-gifted children. *ParcPlace Newsletter*, 5(1), Spring 1992.

H. G. Baker. Equal rights for functional objects or, the more things change, the more they are the same. *OOPS Messenger*, 4(4), Oct. 1993.

A. P. Black. Object identity. In *Proc. of the Third Int'l Workshop on Object Orientation in Operating Systems (IWOOOS '93)*. IEEE Computer Society Press, 1993.

A. P. Black, K. B. Bruce, M. Homer, and J. Noble. Grace: the absence of (inessential) difficulty. In *Onward!*, pages 85–98, 2012.

G. Booch. *Object Oriented Analysis and Design with Applications*. Benjamin Cummings, second edition, 1994.

G. Bracha and D. Ungar. Mirrors: design principles for meta-level facilities of object-oriented programming languages. In *OOPSLA*, pages 331–344, 2004.

C. Chambers, D. Ungar, and E. Lee. An efficient implementation of Self, a dynamically-typed object-oriented language based on prototypes. *OOPSLA*, 1989.

W. R. Cook. On understanding data abstraction, revisited. In *OOPSLA*, pages 557–572, 2009.

D. Crockford. *JavaScript: the Good Parts*. O'Reilly, 2008.

T. V. Cutsem and M. S. Miller. Proxies: design principles for robust object-oriented intercession APIs. In *Proceedings of the 6th Symposium on Dynamic Languages, (DLS)*, pages 59–72, 2010.

T. V. Cutsem and M. S. Miller. Trustworthy proxies - virtualizing objects with invariants. In *ECOOP*, pages 154–178, 2013.

O.-J. Dahl, B. Myhrhaug, and K. Nygaard. *SIMULA: Common Base Language*. Norwegian Computing Center, Oct. 1970.

S. Drossopoulou, J. Noble, and M. S. Miller. Swapsies on the Internet. In *PLAS*, 2015.

E. Gamma, R. Helm, R. E. Johnson, and J. Vlissides. *Design Patterns*. AW, 1994.

A. Goldberg and D. Robson. *Smalltalk-80: The Language and its Implementation*. Addison-Wesley, 1983.

J. Gosling, B. Joy, G. Steele, and G. Bracha. *Java Language Specification*. Addison-Wesley Professional, 3rd edition, 2005.

R. Hickey. *Clojure Reference Manual*. clojure.org, 2016.

D. Hoyte. *Let Over Lambda*. Lulu.com, 2008.

B. Jacobs. Objects and classes, co-algebraically. In B. Freitag, C. B. Jones, C. Lengauer, and H.-J. Schek, editors, *Object Orientation with Parallelism and Persistence*, volume 370, chapter 5, pages 83–103. Kluwer, 1996.

R. Johnson. Erlang, the next Java, Aug. 2007. `http://www.-cincomsmalltalk.com/userblogs/ralph`.

R. Johnson. Object-oriented programming and design, 2008. `http://st-www.cs.uiuc.edu/users/johnson/598rej/`.

S. Krishnamurthi, B. S. Lerner, and J. G. Politz. *Programming and Programming Languages*. Shriram Krishnamurthi, 2016. `http://papl.cs.brown.edu/2016/`.

U. K. LeGuin. *The Left Hand of Darkness*. Macdonald & Co Limited, 1969.

O. Lehrmann Madsen, B. Møller-Pedersen, and K. Nygaard. *Object-Oriented Programming in the BETA Programming Language*. Addison-Wesley, 1993.

B. J. MacLennan. *Principles of Programming Languages: Design, Evaluation, and Implementation*. OUP, 1995.

MC 900 Ft. Jesus. Truth is out of style. Nettwerk Records, 1989.

P. L. McCullough. Transparent forwarding: First steps. In *OOPSLA*, 1987.

B. Meyer. *Object-oriented Software Construction*. Prentice Hall, second edition, 1997.

M. S. Miller. The grant matcher puzzle. http://www.erights.org/elib/equality/grant-matcher, Oct. 1998.

S. Nelson, D. Pearce, and J. Noble. Understanding the impact of collection contracts on design. In *TOOLS Europe*, 2010.

J. Noble. The myths of object-orientation. In *ECOOP*, pages 619–629, 2009.

M. Odersky, L. Spoon, and B. Venners. How to write an equality method in Java. artima developer, June 2009.

G. Orwell. *Animal Farm*. Secker and Warburg, 1945.

G. A. Pascoe. Encapsulators: A new software paradigm in Smalltalk-80. In *OOPSLA*, 1986.

H. Reichel. An approach to object semantics based on terminal co-algebras. *ArticleinMathematical Structures in Computer Science*, 5(2):129–152, June 1995.

G. Steele. *Common Lisp the Language*. Digital Press, 1990.

T. S. Strickland, S. Tobin-Hochstadt, R. B. Findler, and M. Flatt. Chaperones and impersonators: Runtime support for reasonable interposition. In *OOPSLA*, 2012.

B. Stroustrup. Sixteen ways to stack a cat. Technical Report CSTR-161, AT&T Bell Laboratories, Oct. 1991.

D. Ungar and R. B. Smith. SELF: the Power of Simplicity. *Lisp and Symbolic Computation*, 4(3), June 1991.

W. Whitman. *Leaves of Grass*. David McKay, 1891.