CleanJava: Language Design for the Simplification of Object-Based Reasoning

Greg Kulczycki

Technical Report RSRG-03-02
Department of Computer Science
451 Edwards Hall
Clemson University
Clemson, SC 29634-0974 USA

March 2003

Copyright © 2003 by the authors. All rights reserved.

CleanJava: Language design for the simplification of object-based reasoning*

Gregory Kulczycki
Clemson University
gregwk@cs.clemson.edu

22 March 2003

Abstract

References are an integral part of popular object-oriented languages. They permit efficient data movement and parameter passing, and implement object identity. But their current use complicates reasoning by routinely introducing object aliasing. As a result, a significant body of research has focused on alias control techniques for object-oriented languages. This paper aims to simplify reasoning about objects by allowing programmers to ignore references in reasoning without losing the performance benefits they provide. It introduces a language design principle, *cleanliness*, that promotes the view of all variables as independent object values regardless of their implementations. To illustrate how this principle can be realized efficiently in practice, it describes the design of CleanJava. CleanJava is the result of re-engineering Java, focusing on four main areas of language design that involve references: object initialization, data movement, parameter passing, and the implementation of typically linked data structures.

Keywords: Aliasing, cleanliness, data movement, initialization, Java, parameter passing, references, software engineering, swapping

1 Introduction

References are pervasive in popular object-oriented languages. They permit efficient data movement and parameter passing of non-trivial objects, and are used to implement object identity. However, the need to reason about references and the aliasing that results from their routine use in such languages has frustrated students, professional programmers, and formalists alike. As a result, a significant body of research has focused on alias control techniques for object-oriented

*This research is funded in part by NSF grant CCR-0113181.

languages. This paper is concerned with minimizing the need to reason not only about aliasing, but about references in general, without losing the performance benefits that references provide.

> *The objective of this paper is to simplify reasoning about objects by allowing programmers to view all variables as independent values,*[1] *and to do this without sacrificing efficiency.*

The paper introduces the notion of *cleanliness*, which refers to an abstract view of objects in which references can be ignored. To illustrate how the principle of cleanliness can be applied to the design of a high-level language without sacrificing efficiency, we bring together numerous ideas from the software engineering and language literature. We apply these ideas to re-engineering the Java language to make it clean, focusing on following four areas of language design in which references play a major role.

- initialization of objects
- data movement
- parameter passing
- the implementation of typically linked data structures

The resulting object-oriented language, CleanJava, has many desirable features. It frees programmers from the need to reason about references, eliminates aliasing in all but the lowest level components, makes no conceptual distinction between built-in and user-defined types, mitigates the need for system-wide garbage collection, and allows variable names to uniquely represent object identities. It does all this without requiring a major paradigm shift in programming and without sacrificing efficiency.

The main contribution of this paper is its description of cleanliness and its illustration of how this principle can be realized efficiently in the design of a high-level, object-oriented language.

2 Motivation

Problems relating to references have been around ever since they were first introduced into high-level languages. Hoare called their introduction "a step backwards from which we may never recover" [12] and cited reasoning difficulties for programmers in all his top concerns. In [13] he spells out these concerns more fully and proposes recursive data structures as an alternative to one common use of references. Kieburtz made the case for "programming without pointer variables" in [17] and described how high-level languages could eliminate them.

[1]When we use the term *value* by itself in this paper, we always refer to the value of the object rather the value of the reference. When we mean the value of the reference we will explicitly say so.

Despite these early works, pointers and references persisted in languages. The popularity of object-oriented languages brought a renewed concern with the main obstacle to clean reasoning—aliasing. Hogg and others outlined the issues surrounding object aliasing in [15], and suggestions for encapsulating aliasing appear in [3, 14]. These papers represent early work in the area of alias control, which has seen a number of recent publications, including [5, 6, 23].

Problems due to references and aliasing affect many areas of software engineering. Educators have noted that students have trouble understanding and reasoning about pointers [18, 24]. Formalists have noted the complexity that references and aliasing contribute to formal specification and verification, regardless of whether specification languages are value based or reference based [1, 28, 29]. To minimize complexity in reasoning (for example in C++) practitioners recommend value semantics over reference semantics [7, 22]. Finally, references are at the center of an ongoing debate involving manual memory management and automatic garbage collection.

The remainder of this paper is divided into four main parts. Section 3 introduces the notion of cleanliness and explains what we mean when we refer to clean languages. Section 4 illustrates how the principle of cleanliness can be applied to an object-oriented language by re-engineering Java to make it clean. Section 5 discusses some of the implications of designing clean languages. Sections 6 and 7 conclude by summarizing related work and identifying future directions for research.

3 Cleanliness

Implicit references in languages such as Java and explicit pointers in languages such as C++ both serve the same purpose—they are abstractions for addresses in memory. A *clean reasoning environment* is one in which programmers *do not* require abstractions of memory addresses to reason soundly about their code.

In a clean reasoning environment all variables can be viewed as independent objects. Reasoning about built-in integers in Java is clean since *int* variables always represent independent integers. Reasoning about Java Strings is *not* clean because string variables may become *null*, which is a reference rather than an object. If string variables could not be null then reasoning about them would always be clean, because Java strings are immutable and their (non-null) variables can be viewed as independent objects. Reasoning about mutable types in Java would not be clean even if null values were eliminated because aliasing among mutable objects sets up a dependency among them [23].

A *clean* mechanism or method is one that preserves a clean reasoning environment, and an *unclean* mechanism or method is one that corrupts a clean reasoning environment. If s and t are stacks and x is mutable, each of the following Java statements will corrupt an otherwise clean environment.

```
t = null;
t = s;
s.push(x);
```

The null assignment forces the programmer to reason about t as a reference rather than an object; the variable assignment introduces aliasing and and forces a programmer to reason about s and t as dependent objects; and the push method forces a programmer to reason about s and x as dependent objects, because x is aliased to the top element of s after the call.

3.1 Cleanliness, value semantics, and models of storage

Cleanliness refers to the state or quality of being clean. It is a stricter notion than value semantics. A variable has value semantics if the programmer can think of it as a value. This is sometimes used to mean a variable can be thought of as a value *rather than* a reference, but not always. For example, it is possible to view explicit pointer variables in C++ as having value semantics because programmers can think of them as *values* of memory addresses. Likewise, parameter passing in Java can be viewed as having value semantics because the *value* of the variable (which is a reference) is being copied. But reasoning about explicit pointers in C++ and viewing variables as references in Java are both *unclean* because the programmer must reason abstractly about memory addresses.

Cleanliness is different from the notion of a direct model of storage as well. Variables operating under a direct model of storage refer directly to their representations, which are often allocated on the stack, whereas variables operating under an indirect model of storage refer indirectly (through references) to their representations, which are often allocated on the heap. The notion of cleanliness concerns how a programmer must reason about variables, and it is independent of how variables are implemented by the compiler. For efficient data movement, variables may be implemented with references, but as long as programmers can reason soundly about them *as if* they directly refer to objects, clean reasoning can still be achieved.

This last point is especially important in the context of object-oriented languages, where variables are regularly implemented with references. For example, even though dynamic binding may be implemented using reference variables, there is nothing inherently unclean about it. A shape variable s may represent a circle in one state and a square in another. As long as s can be viewed as an independent object in either state, reasoning is clean.

3.2 Clean languages

A clean language is one in which routine reasoning about objects is clean. Pure functional languages are clean, but most imperative languages are not. We see two broad uses for references in imperative languages that often corrupt a clean reasoning environment.

1. References are used to enable efficient data movement and parameter passing for non-trivial objects.

2. References are used to enable the efficient implementation of typically linked data structures such as lists and trees.

The first use is routine, so unclean mechanisms such as reference copying cannot be used in clean languages. In this paper we consider several clean alternatives to reference copying for data movement and parameter passing. The second use is not routine, but clean language designers must be able to contain the unclean reasoning that occurs in such implementations, and stop it from corrupting the reasoning of the data structure's client. For example, a list implementation may require unclean reasoning, but there must be a mechanism that permits programmers to reason about list objects cleanly.

4 From Java to Clean-Java

This section illustrates how to apply the principles of cleanliness to language design by re-engineering the Java language to make it clean. We call the resulting language CleanJava.[2] Like many other object-oriented languages, Java presents a particular challenge for clean technology because it uses reference variables for all objects, and variables that must be viewed as references are inconsistent with the notion of cleanliness. The first three areas of language design we concentrate on—initialization, data movement, and parameter passing—are areas in which Java's specified behavior corrupts a clean view of objects, either by introducing null variables or introducing aliasing. In such cases alternative approaches that preserve cleanliness are proposed. The last area—linked data structures—represents one of the few areas in which a clean view of objects may be at odds with efficient computing. For this purpose we introduce a referencing component tailored to the implementation of linked data structures and discuss how data abstraction and information hiding can be used to preserve a clean view for higher level components.

In each of the four areas of language design, we describe several options, discuss their appropriateness in terms of cleanliness and efficiency, and identify our choice for CleanJava. The final choice is rarely the only option that is both clean and efficient. Possibilities for clean languages based on other options are discussed in the final section of this paper.

4.1 Initialization

Issues relating to cleanliness arise as soon as a variable is declared. Consider the following variable declaration.

```
Circle c;
```

There are numerous ways to view the variable c in this state.

[2] The Java (and CleanJava) examples presented in this paper make use of generics, which are include in the 1.5 release of Java.

1. c is a null reference, and it should be assigned a circle object.
2. c is undefined, and it should be assigned a circle object.
3. c is a valid, but unspecified circle object.
4. c has the value of a default circle object.

The first view is the standard view of newly declared variables in Java, but it is obviously unclean since it requires programmers to think of c as a reference rather than a value. The second view is technically clean, since it does not involve references, but it introduces a notion of definedness that programmers must consider for each variable. This view is closely related to the first in terms of how programmers must reason about variables. In either case, a call to a method *c.draw()* just after the declaration of c would be illegal. In Java, a compiler error warns that the variable c may not have been initialized. A similar compiler error could be reported in a language with the notion of definedness warning that the variable c may not be defined.

The third view looks similar to the second, but has some important differences. Under this view the variable c would be defined, so that method calls such as *c.draw()* would be technically permissible. While a *draw()* method might seem of dubious usefulness for an object whose value is unknown, programmers may find use for other methods. For example, methods such as *clear()* and *update(Point center, Float radius)* could potentially replace constructors, acting as literals do for built-in types. This would further simplify the language, since such literals would not require the special treatment that constructors get in languages like Java. Decisions would still have to be made on the appropriateness of compiler warnings or error messages to help programmers avoid calling methods such as *draw()* on an object whose value is unknown.

Both the second and third views are consistent with a clean view of objects and may merit further examination, but we have chosen the fourth view for use in CleanJava. When a variable is declared, it gets a default value of its type. This is consistent with built-in types in Java such as *ints* (where the default value is 0) and *booleans* (where the default value is *false*). This approach raises two questions: Is it always possible to obtain a default value for a given type? and, Can this approach be implemented efficiently?

As for default values, these may be automatically generated for arbitrary types by assigning default values to all instance variables. This approach will work provided that all low-level types have default values, and that no recursive type structures are permitted. Java *does* permit recursive type structures, which are used primarily for implementing typically linked data structures. We provide a referencing component (described below) for this purpose, and prohibit recursive type structures in CleanJava. Though automatic initial values may be generated, programmers are encouraged to provide their own initialization routines for all types, just as Java programmers are encouraged to provide default constructors for all types. This is especially true for data structures implemented with arrays, since the initial value for an array of type T is an

array of length one that contains a default object of type T—rarely what the programmer would desire.

CleanJava replaces constructors with static *create* methods. If a *create()* method (without parameters) exists, it will be invoked for a newly declared variable. Any instance variable not explicitly given values in a creation method will contain default values. Consider the following creation methods.

```
// Circle creators
public static create() { this.r := 1; }
public static create(Float x, Float y, Float r) {
    this.x_value := x;
    this.y_value := y;
    this.radius := r;
}
```

Based on these creators, the following three circles are initialized to the same value.

```
Circle a;
Circle b := Circle.create();
Circle c := Circle.create(0, 0, 1);
```

Regarding the question of efficiency, it is often the case that a variable a is declared, and then a value from another variable b is assigned to a before the object a is used for anything. If memory for a default object is allocated to a as soon as it is declared, the time and space used for that allocation may have been wasted. This can be prevented by a simple compiler optimization related to lazy evaluation, called lazy initialization, where memory for an object is not actually allocated until the object is used. From a reasoning perspective the programmer can still think of the object as having a default value until it is modified.

4.2 Data movement

There are many potential data movement alternatives to assignment, though only a few—copying, destructive read, and swapping—are consistently mentioned in the literature. A rare analysis of the many approaches to data movement appears in [30]. Consider the following approaches for circles c and d.

1. *assignment* (c = d) — The reference of d is copied to c, and both variables point to the same circle.

2. *destructive read* (c = d) — The variable c points to the circle that d used to point to, and d becomes null.

3. *relational transfer* (c ← d) — c gets the value d had, and d becomes an unspecified but valid circle.

4. *clearing transfer* (c $\leftarrow$ d) — c gets the value d had, and d gets the default value of a circle.

5. *swap* (c :=: d) — c gets the value d had, and d gets the value that c had.

6. *copy* (c := d) — The value of d is copied to c.

Java uses assignment for objects. It copies a reference from one variable to another and introduces aliasing as a result. Aliasing and its associated reasoning complexity is precisely what clean language designers hope to minimize, so its routine introduction during data movement is unacceptable in CleanJava. The destructive read operation forces programmers to reason about references by making one variable a null reference. This can be avoided—as described in the previous section—by introducing a notion of definedness into the language. The case for undefined variables, however, is even weaker here than it was for newly declared variables. A newly declared variable that remains undefined is easily caught be the compiler, but variables whose definedness status can continually change add significant complexity to the task of tracking and detecting undefined variables. Instead of having to worry about run-time errors due to null pointer exceptions, programmers would have to worry about run-time errors due to undefined variable exceptions. We refer to this class of errors as *missing-object* errors, since they arise when programmers attempt to do something with an object that is not really there. Tools such as ESC/Java [8] can help programmers catch these missing-object errors statically, but the errors themselves seem misplaced in a clean language. If missing-object errors arise ultimately because of references, then clean language designers, who hope to minimize the negative impact of references, should also hope to eliminate the possibility of such errors.

The relational transfer approach can avoid missing-object errors because a defined object always exists for any variable, but calling *c.draw()* when the value of c is unknown is probably not something the programmer intended to do. A run-time error will not occur during execution, but unexpected results might. Producing a static warning message to the effect that the programmer is trying to do something with an unspecified object would be helpful, but would involve the same computational and reasoning complexity that statically detecting missing-object errors involves.

In comparison to the data movement operations mentioned so far, the clearing transfer operation is straightforward. A value is transfered from one variable to another and the value of the transferring variable is cleared (e.i., given its default value). The drawback of this approach is its efficiency. To perform this operation a new default object must be created and memory from an old object must be reclaimed. Assume that c and d are circle objects implemented internally by the compiler as pointers to circles. Then $c \leftarrow d$ would deallocate the circle object that c points to, move c's pointer to d's circle, create a new circle with a default value, and move d's pointer to the new circle. The deallocation is performed because we are assuming that the operation occurs in a clean language, so that c's reference is unique. The clearing transfer operation will

generally be much more efficient than the copy operation since the amount of storage a default value displaces will always be less than or equal to the amount of storage a non-default value displaces.[3] Lazy evaluation could be used here to improve general efficiency.

CleanJava uses swapping as its primary means of data movement. Swapping is efficient because the compiler can represent all non-trivial objects using references, and implement it by swapping references. It is also clean because a programmer can reason about it as if values are swapped. In many cases, swapping can be made just as efficient as assignment. A compiler may implement small objects—such as integers and booleans—as values and exchange those values during a swap statement. This would enable an Integer variable in a language like CleanJava to have the performance profile of a Java *int* rather than a Java *Integer* (the wrapper class for ints). The classical way to swap objects is to create temporary storage and perform three copies. However, several machine languages already implement an *exchange* instruction that executes in one clock cycle, so that swapping need not incur any more overhead than assignment.

One area in which swapping is not be suitable in an object-oriented language is for data movement from a type to its supertype. For shape s and circle c, where circle is a subtype of shape, the assignment $s = c$ is acceptable because both s and c end up pointing to the circle object. However, the swap statement $s :=: c$ would not work, because c is not allowed to represent a non-circle shape object. The most direct way to avoid this problem is to prohibit swapping between types and their supertypes and to provide one of the clean transfer methods listed above in the language. CleanJava provides the clearing transfer operator for this purpose.

The final approach listed above, copying, is an obvious part of any clean language. When it can be done efficiently, copying provides an excellent way to cleanly move data from one place to another. Were it not for the inefficiency of copying for non-trivial objects, references might never have worked their way into high-level languages, and all such languages might have remained perfectly clean. The particular way in which CleanJava handles copying is described in detail in the next subsection.

4.3 Function assignment

Java methods return references rather than values, potentially violating the principle of variable independence necessary for a clean reasoning environment. For example, assume that r is a record, s is a stack, and *top()* is a method that returns a reference to a stack's top element. Then the following statement sets up a dependency between r and s.

```
r = s.top();   // r points to the top element in s
```

[3]This will be true of automatically generated default values and by convention should be true for user-defined default values as well.

Changes to r will now affect the value of s, and reasoning about references becomes mandatory. In addressing this problem for CleanJava, two opposing approaches to functions had to be considered.

- Functions may modify their arguments.
- Functions may *not* modify their arguments.

The benefit of the first approach in terms of CleanJava is that it is consistent with the way that Java handles methods. It would allow, for example, the statement r = *s.pop()*, in which s looses its top value to r. Though there is nothing inherent in this approach that corrupts a clean reasoning environment, side-effect free functions can help simplify reasoning in object-oriented languages (see Hogg's comments in [14]). Therefore, CleanJava makes a distinction between functions and procedures. Functions return a value and may not modify their arguments, while procedures do not return a value but may modify their arguments. Instead of a *pop function*, CleanJava requires a *pop procedure* that is called with the following statement.

s.pop(r);   *// s looses its top value to r*

The distinction between functions and procedures gives functions the role of object creators. The function assignment operator and the copy operator are the same in CleanJava (":="), indicating a relation between the two. In fact, copying in CleanJava is simply a shorthand for a function assignment involving a distinguished method named *replica*. The compiler treats the following statements the same.

a := b;
a := b.replica();

In either case, if no *replica()* method exists for b's type, an error is reported. In Java, the assignment operator behaves differently for built-in types than it does for user-defined types: it copies values for built-in types, and copies references for user-defined types. In CleanJava, all variables are values, and all operators have consistent value semantics for all types. However, copying is potentially expensive for large objects, so users are given the option of implementing it for some types and not for others. A type that permits copying is considered *replicable*. All that a user has to do to make a type replicable is provide a *replica()* method. For routine data movement on all types, CleanJava provides efficient swapping as described above.

4.4 Parameter passing

Parameter passing in Java is accomplished by copying the references of the arguments to the formal parameters. Copying references introduces a cleanliness problem when arguments are repeated as parameters to a procedure call. We consider the following parameter passing approaches in terms of their cleanliness.

1. *Reference copying* — This is Java's approach to parameter passing. References of actuals are copied to formals at the beginning of the call.[4]

2. *Disciplined approach* — Repeated arguments are prohibited by the compiler or avoided by the programmer, and reference copying is used otherwise.

3. *Object copying* — Object values of actuals are copied in to formals at the beginning of the call, and copied out at the end of the call.

4. *Swapping* — Values of actuals are swapped in to formals at the beginning of the call, and swapped out at the end of the call.

Consider the following *append* method for queues.

```
/** Append the specified queue to this queue,
    emptying the specified queue. */
public void append(Queue<Item> q) {
    Item x;
    while (!q.isEmpty()) {
        q.dequeue(x);
        this.enqueue(x);
    }
}
```

The following statement calls the *append* method on queues *q1* and *q2*.

```
q1.append(q2);
```

If the queues are distinct objects, the code will append *q2* to *q1*. However, if *q1* and *q2* are aliases, the **while** loop will not terminate. The influence of the repeated argument problem can be detected in Java's *List* interface. The list operation *addAll* is unspecified if the current list and the collection being added are the same object.[5] In a clean language, *q1* and *q2* are always distinct, but the problem needs to be addressed when the same variable is repeated for both arguments as show below.

```
q1.append(q1);
```

Allowing such code forces programmers to reason about variables *q* and *this* in the implementation as if they were potential aliases, which is unacceptable in a clean language.

Since repeating arguments as parameters is considered bad programming practice anyway, a disciplined approach to parameter passing may be used, whereby the programmer tries to avoid repeating arguments and the compiler reports an error if it detects a repeat. This approach can be effective for calls such as *q1.append(q1)*, but consider the same call where the variable *a* is an array of queues.

[4] Known as *call-by-value* to Java programmers, since *values* of references are being copied.

[5] See `http://java.sun.com/j2se/1.4.1/docs/api/java/util/List.html`.

```
a[i].append(a[j]);
```

The arguments are only repeated when i equals j, a condition that cannot, in general, be determined until run-time. Prohibiting array variables as parameters seems too restrictive, so a run-time error would occur if the method is invoked when i and j are equal.

The object copying approach is clean even when arguments are aliased, so it is also clean when arguments are repeated. Assume that *q1* is repeated in the call to *append*. The queue object that *q1* denotes is copied to both formals, *this* and *q*. Thus, at the beginning of the procedure, *this* and *q* denote objects with the same value, but they are independent. At the end of the procedure, *this* has doubled in length, and *q* is empty. The copying out is the interesting part. Both *q* and *this* must be copied back to *q1*, but *q1* is only one variable, so it will get the value of the *last* formal copied back, because copying always overwrites the value of the target object. Thus, the object copying approach requires the language to specify the order in which the formals are copied back to the actuals.

The object copying approach sounds grossly inefficient, but in fact, when arguments are not potentially repeated, it behaves just like the reference copying approach and can therefore be optimized using reference copying. It becomes expensive only when arguments are repeated as parameters. Potential repeated arguments would only be permitted for replicable objects, so this approach would have to be used in conjunction with a disciplined approach for non-replicable objects.

The approach to parameter passing that CleanJava uses is the swapping approach. As with the object copying approach, when objects are not potentially repeated it can be implemented with reference copying. Conceptually, the swapping approach involves the following steps.

1. The compiler creates formals which have initial values.
2. The actuals are swapped with the formals in left to right order.
3. The procedure body is executed.
4. The formals are swapped with the actuals in the reverse order that they were swapped in (right to left).

Thus, in the call to *q1.append(q1)*, compiler variables *this* and *q* are created and have empty queues as initial values. Next, *q1* is swapped with *this*, so that *q1* has an empty queue object. Then *q1* is swapped with *q*. Both of them were empty queues before the swap, so they are both empty queues after the swap. The procedure body is executed, yielding no change to *this* or *q* since *q* is empty. Finally, *q* is swapped out to *q1*, which remains empty, and then *this* is swapped out with *q1*, giving *q1* its original value back. The swapping approach is equivalent to an approach using clearing transfer.

4.5 Parameter modes

CleanJava has five parameter modes: **in**, **out**, **inout**, **restore**, and **eval**. The first three (*in*, *out*, and *inout*) can only be used in procedures, the fourth (*restore*) can only be used in functions, and the last (*eval*) is typically used for replicable objects and may be used either in procedures or functions. The *inout* mode is the default mode for all procedure parameters and the *restore* mode is the default mode for all function parameters. Therefore, in practice, only three of the parameter modes will be used explicitly: *in*, *out*, and *eval*. The modes are described briefly below. The reader should be careful not to confuse the modes for procedures (*in*, *out*, and *inout*) with Ada's modes of the same name. While there are some similarities, there are very real differences, and we feel that the mode definitions below are both intuitive and flexible in the context of CleanJava's approach to parameter passing.

in The *in* parameter mode indicates that the value going *in* to the procedure is important for reasoning about the procedure body, but the value coming *out* of the procedure is not important for reasoning about the client code. Thus, the value of the variable after the call is unspecified from the client's perspective. One common use for this mode is for objects inserted into a container, as in the following method interface.

> **public void** push(**in** Item x);

The compiler will report an error if an object with this mode is passed to another procedure as an *out* or *inout* parameter.

out The *out* parameter mode is the opposite of the *in* mode. It indicates that the value going *in* to the procedure is irrelevant to the implementer of the procedure body, but the value coming *out* of the procedure is important to the client. Thus, the value of the variable passed in to the procedure body is unspecified from the implementor's perspective. One common use for this mode is for objects removed from a container, as in the following method interface.

> **public void** pop(**out** Item x);

The compiler will report an error if an object with this mode is passed to another procedure as an *in* or *inout* parameter.

inout The *inout* parameter mode indicates that the object may be modified, and that both the incoming and outgoing values are important. It is the default mode for all parameters in procedures, and therefore it will rarely be indicated explicitly.

restore The *restore* parameter mode indicates that the object being passed in should have the same abstract value after the function call as it did before the call. This condition provides wide flexibility to the implementer of the function but is not generally enforcible by a compiler. For example, an

operation to copy a container data structure might remove the elements from the original container, copy them, put the copies into a new container, and then put the original elements back into the original container. In this case, the incoming value and the outgoing value of the original container are the same. The compiler can issue a warning if the object is passed to a procedure as an *in*, *out*, or *inout* parameter. In general, however, there is no way to statically guarantee that the parameter will be returned to its original value if it is modified by the function body—it is up to the programmer to adhere to this convention. The *restore* parameter mode is the default mode for function parameters and therefore it will rarely be indicated explicitly.

eval The *eval* parameter mode indicates that a function is expected for *evaluation*. As with function assignment, if a variable a is given where a function is expected, the compiler will translate it as *a.replica()*. If no *replica* function is found, the compiler will report an error. The *eval* mode is often used for small types such as integers and booleans.

4.6 Typically linked data structures

There are several approaches that a clean language designer may want to consider for implementing linked data structures.

1. *Rich library approach* — Include a collection of typically linked data structures with the language and require programmers to use these to build other such data structures.

2. *Recursive data structure approach* — Include a mechanism in the language that supports recursive data structures.

3. *Pointer component approach* — Include a pointer component in the language that is tailored toward the implementation of linked data structures, and ensure that the language has a mechanism that supports data abstraction and information hiding.

The *rich library approach* is the least flexible but ensures that even the lowest level classes remain clean. It is doubtful that a general purpose language can provide a rich enough library to satisfy both the functional and performance needs of all programmers, but this may be a viable approach for, say, a scripting language or a domain-specific language.

The *recursive data structure approach* provides special syntax and semantics that allow programmers to implement acyclic data structures such as lists and trees efficiently and cleanly. Variations of this approach can be found in many functional and logic languages. Though it cannot be used to implement cyclic structures, it may be a reasonable approach for clean language designers if combined with complementary library components.

CleanJava combines the rich library approach with the *pointer component approach*. The pointer component approach is not fully clean, but is the only

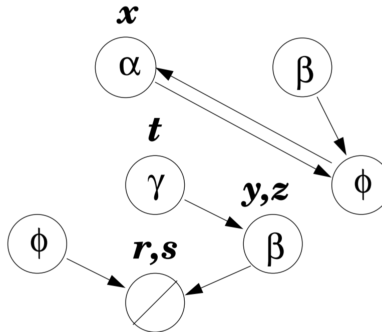

Figure 1: A system of linked locations.

approach that gives programmers maximal flexibility for data structure implementations. Its use in a clean language is justified because implementing linked data structures does not constitute a *routine* use of the language, especially when many such data structures can be provided by a sufficiently rich component library. For this approach to be effective in a clean language, unclean reasoning in data structure implementations must not corrupt the reasoning of programmers who use the data structure. In object-oriented languages, the class mechanism serves this purpose.

4.7 The position class

The position class in CleanJava captures the behavior and performance benefits of traditional pointers and is designed to make implementing linked data structures easy. It is based on a formally specified pointer component detailed in [19] and its generic interface is given below.

```
public interface Position<Item> {
  public void new();
  public void free();
  public void moveTo(Position<Item> q);
  public void swapObject(Item x);
  public void redirect(Integer i, Position<Item> q);
  public void follow(Integer i);
  public Boolean isWith(Position<Item> q);
  public Boolean isAtVoid();
}
```

Positions are created by a factory class called *LocationSystem* whose name derives from the underlying abstraction that positions take part in—a system of linked locations. A system of linked locations is made up of a finite number of locations and one-way connections between them called links. Each location contains a specific object and a fixed number of links to other locations. A distinguished location known as *void* serves as the default location. Figure 1 shows a system of locations containing symbol objects and one link per location.

The list below shows example method calls and gives brief explanations of how they modify the system of linked locations.

- *p.new()* — Memory is allocated for a default object of type *Item*, the object is placed in a newly created location, links are established and directed to void, and p is relocated to the new location.
- *p.free()* — Memory taken by the object at p's location is freed, the location and its links are abandoned, and p is relocated to void.
- *p.moveTo(q)* — p is relocated to q's location.
- *p.swapObject(x)* — The object at p's location is swapped with x.
- *p.redirect(i,q)* — The i-th link at p's location is redirected to q's location.
- *p.follow(i)* — p is relocated to the location targeted by its i-th link.
- *p.isWith(q)* — True if p is located with q.
- *p.isAtVoid()* — True if p is at the void location.

The underlying notion of a location is similar to that of a pointer with two important differences. First, locations are linked to other locations, and position variables may follow a link, or have links at their locations redirected toward other locations. Second, locations are not dereferenced. Before a programmer can manipulate an object it must be swapped out of its location. An example use of the position class to implement a traversable list is given in the appendix.

The fact that positions are part of a factory pattern is important because it allows different location systems (which serve as factories) to provide different implementations for the position interface. Thus, safe pointer and checked pointer implementations can be used to help ensure proper memory management [2, 24]. The language itself must provide a default, no-frills implementation that yields maximum performance, but it can also provide at least one garbage collecting implementation, *GCLocationSystem*, that includes a method *freeInaccessible* to clean up local memory leaks.

5 Implications

This section looks at some of the implications of clean languages for software engineering. It begins with a discussion of the benefits that programmers can gain by using CleanJava over Java, and then addresses the paradigm shift that must accompany that change. Finally, it looks at the broader impact that clean languages could have on software engineering.

5.1 CleanJava vs Java

The main benefit of CleanJava over Java is that it frees programmers from the need to reason about references routinely. All objects in CleanJava are independent of each other, and programmers can view them as values just as they view ints and booleans in Java. In fact, the distinction between built-in types and user-defined types goes away,[6] and all objects include a swap operator for data movement. There is a distinction between replicable and non-replicable objects, but any object can be made replicable by adding a replica method. The programmer has a responsibility to consider the implications on performance before making a class replicable. Since all objects are independent, there is no need to worry about unintended aliasing. A programmer can trace an object through the code secure in the knowledge that changes to other variables will not affect it. Null pointer exceptions will no longer occur, and the distinction between mutable and immutable types loses its importance.

System wide garbage collection is no longer a necessity in CleanJava. Memory leaks in CleanJava can be caught by local garbage collecting implementations of the position class. Typically linked data structures provided by the library such as lists and trees can be verified against memory leaks. By using a predictable garbage collecting implementation or non-garbage collecting implementation, CleanJava becomes a viable language for real-time and embedded systems where predictability is paramount.

5.2 Paradigm shift

Consider a programmer who is comfortable with Java but is forced to switch to another programming language. Several languages are given below along with a partial list of areas in which the Java programmer would have to adjust to a new paradigm.

- *Smalltalk* — Some paradigm shift. Variables of all types (even built-in types) are considered references, and the distinction between mutable and immutable types is more critical than in Java. Also, it uses dynamic rather than static typing.

- *C++* — Some paradigm shift. Classes may be designed with value semantics or reference semantics. Memory management is typically manual.

- *C++ with STL* — Medium paradigm shift. Template use is more common than with C++ alone. Generic algorithms are typical. Value semantics are emphasized.

- *Ada* — Medium paradigm shift. The language is object-based with object-oriented features added in. It does not use object notation, and is built around modules rather than classes. It relies heavily on generics.

[6]Built-in types will still have special syntax for them, such as $i + j$ for Integers and $b1 \mid\mid b2$ for Booleans.

- *C* — Large paradigm shift. No concept of objects. Weak type checking.
- *ML* — Huge paradigm shift. The underlying programming model is functional rather than imperative.

If CleanJava were added to the list above, we would be tempted to place it after C++ but before C++ with STL. The main paradigm shift a programmer makes when switching from Java to CleanJava involves the notion of cleanliness and its corollary—that all variables directly represent independent objects. Fortunately, this notion is easily grasped since small built-in types such as integers and booleans already behave this way. Another important change in thinking that must be made is that swapping is the standard way to move non-trivial objects. This change will be slightly more difficult, but once programmers see that many assignments in traditional code can be replaced by swaps with no adverse affect on behavior, they should quickly adjust. The other changes—side-effect free functions, different parameter passing mechanisms, and the targeted pointer component—are changes that have similar counterparts in other languages. It should also be noted that paradigm shifts are easier to digest if they make it easier to understand and reason about the resulting software, as CleanJava does.

A conceptual difference between Java and CleanJava that bears special mention concerns the notion of object identity. Programming languages generally fall into two categories with respect to identity. They either have built-in identity or name identity. Object-oriented languages such as Java have built-in identity, and use an identity test operator "==" to check if two variables refer to the same object. A problem in languages with name identity occurs when "a single object may be accessed in different ways and bound to different variables without having a way to find out if they are the same object" [16]. Clean languages do not have this problem because different variables always denote independent objects.[7] Clean languages could be said to have a special type of name identity called *unique name identity* that avoids at least some of the problems of common name identity.

5.3 Impact on software engineering

The potential benefits that clean languages offer for software engineering are significant. Some of these are discussed below.

- *Education.* Koenig and Moo suggest that C++ students should be introduced to "value-like" classes before pointers are ever mentioned [18], and a common complaint from Java instructors is that they must teach students about references early on. If clean languages were used, students could be taught software construction using clean classes and treatment of pointers and references could be deferred until the implementation of low-level components was discussed.

[7] Thus, CleanJava does not require the identity test operator "==". Built-in types such as integers and booleans can use "=" as syntactic sugar for a value equality function since the Java assignment operator is no longer used.

- *Industry.* Clean languages allow professional programmers to benefit from code that is easier to understand and maintain without sacrificing performance. The less time programmers have to spend worrying about the complexity inherent in aliasing and indirection, the more time they can spend focusing on the problem they are trying to solve.

- *Formal Methods.* Even with formal specification languages such as JML (Java Modeling Language) [21] that are tailored to object-oriented languages, aliasing introduces significant complexity into formal reasoning and modular verification. If significant portions of programs could be written in a clean reasoning environment, it may become feasible to specify and verify non-trivial software projects.

- *Software Components.* If software components do not have accurate and understandable behavioral specifications, their potential for black-box reuse is limited; if components are not efficient, programmers will continue to build and use custom components. The value based components of clean languages lend themselves to easy specification without compromising performance concerns.

6 Related work

The notion of cleanliness is closely related in spirit to the work of Hoare and Kieburtz, who expressed concerns about references in general rather than worrying about aliasing in isolation [12, 17]. However, they put considerable effort into finding clean ways to implement linked data structures, while we focus on alternatives for routine uses of references such as data movement and parameter passing, and rely on abstraction and encapsulation to hide unclean implementations of linked data structures. The significant body of research on alias control for object-oriented languages motivated our decision to apply clean technology to an object-oriented language such as Java rather than a procedural language such as Ada.

Alias control techniques focus on alias encapsulation rather than alias prevention [3, 6, 14, 23]. This is typically done by preventing reference variables outside an object from being aliased to the object's representation (the object's instance variables). Techniques vary in their flexibility and generally aim to allow programmers to view objects as they would typically view them. These techniques are not intended to eliminate the need for programmers to reason about references and therefore cannot be considered clean.

Alternatives to reference copying that have appeared in the literature include destructive read, copying, and swapping [3, 4, 10]. A survey of techniques that includes all of those mentioned here is in [30]. A discussion of cleanliness as it relates to parameter passing can be found in [20]. The repeated argument problem is often discussed in the context of formal proof rules for procedure calls [9, 11]. Recursive data structures were proposed in [13], and a fuller explanation of pointer components appears in [19].

The Euclid language [25] attempts to minimize the reasoning complexity associated with aliasing by permitting pointers in a restricted way. Several choices made for clean mechanisms were influenced by ongoing research associated with the RESOLVE language [26, 27].

7 Status and conclusions

We hope to pursue the development of CleanJava while investigating possibilities for other clean languages. Therefore, one of our immediate goals is an implementation of CleanJava. This project consists of two stages. The first stage—currently underway—involves implementing CleanJava with Java. This first version of the language will be suitable for classroom use, help us get input from practitioners, and allow us to investigate issues related to understanding and reasoning in clean languages. These issues include understanding how programmers work in clean languages, what specific paradigm shifts are involved, and how unique name identity affects software design. In the second stage, CleanJava will be implemented in a low-level language that allows us to focus our efforts on optimizations and efficiency. This implementation will allow us to evaluate the language in terms of performance and predictability.

Given the variety of clean mechanisms described in this paper, other clean languages may be designed using a mix-and-match approach with the options presented here. For example, a clean language designer might use a definedness approach to instantiation, a relational transfer approach to general data movement, and an object copying approach to parameter passing. A clean language that permits side-effecting functions might be more palatable to traditional object-oriented programmers. Also, the implications of a clean language based strongly on relational transfer merit further study, since research on this mechanism is almost non-existent.

Clean languages free programmers from the need to reason routinely about references and aliasing, and allow them to view variables as independent values. This clean view of objects simplifies understanding and reasoning and eliminates the arbitrary distinction between built-in and user-defined types. It can be realized without sacrificing efficiency and without a requiring programmers to make a major paradigm shift in how programs are written. The potential impact of clean languages on education, programming, formal methods and software engineering promises to be significant. Clean and efficient languages are possible, and cleanliness should be a fundamental consideration in the design of the next generation of languages.

References

[1] M. Abadi and K. R. M. Leino. A logic of object-oriented programs. In M. Bidoit and M. Dauchet, editors, *TAPSOFT '97: Theory and Practice*

of Software Development, 7th International Joint Conference, pages 682–696. Springer-Verlag, New York, 1997.

[2] A. Alexandrescu. *Modern C++ Design: Generic Programming and Design Patterns Applied*. Addison-Wesley, 2001.

[3] P. S. Almeida. Balloon types: Controlling sharing of state in data types. In *Proceedings ECOOP '97*, number 1241 in Lecture Notes in Computer Science, pages 32–59, New York, 1997. Springer-Verlag.

[4] H. G. Baker. 'Use-once' variables and linear objects—storage management, reflection and multi-threading. *ACM SIGPLAN Notices*, 30(1):45–52, January 1995.

[5] B. Bokowski and J. Vitek. Confined types. In *Proceedings 14th Annual ACM SIGPLAN Conference (OOPSLA '99)*, pages 82–96. ACM Press, November 1999.

[6] D. G. Clarke, J. M. Potter, and J. Noble. Ownership tpyes for flexible alias protection. In *OOPSLA '98 Conference Proceedings*, pages 48–64. ACM Press, 1998.

[7] J. O. Coplien. *Advanced C++ Programming Styles and Idioms*. Addison-Wesley, 1991.

[8] C. Flanagan, K. R. M. Leino, M. Lillibridge, G. Nelson, J. B. Saxe, and R. Stata. Extended static checking for java. In *Proceeding of the ACM SIGPLAN 2002 Conference on Programming language design and implementation*, pages 234–245. ACM Press, 2002.

[9] D. Gries and G. Levin. Assignment and procedure call proof rules. *ACM Transactions on Programming Languages and Systems*, 2(4):564–579, October 1980.

[10] D. E. Harms and B. W. Weide. Copying and swapping: Influences on the design of reusable software components. *IEEE Transactions on Software Engineering*, 17(5):424–435, May 1991.

[11] C. A. R. Hoare. Procedures and parameters: An axiomatic approach. In E. Engeler, editor, *Proceedings Symposium on Semantics of Algorithmic Languages*, pages 102–116. Springer-Verlag, 1971.

[12] C. A. R. Hoare. Hints on programming language design. In C. A. R. Hoare and C. B. Jones, editors, *Essays in Computing Science*. Prentice Hall, New York, 1989.

[13] C. A. R. Hoare. Recursive data structures. In C. A. R. Hoare and C. B. Jones, editors, *Essays in Computing Science*. Prentice Hall, New York, 1989.

[14] J. Hogg. Islands: Aliasing protection in object-oriented languages. In *Proceedings OOPSLA '91*, volume 26 of *ACM SIGPLAN Notices*, pages 271–285. ACM, 1991.

[15] J. Hogg, D. Lea, A. Wills, D. deChampeaux, and R. Holt. The Geneva Convention on the treatment of object aliasing. *OOPS Messenger*, 3(2):11–16, 1992.

[16] S. N. Khoshaflan and G. P. Copeland. Object identity. In *OOPSLA '86 Conference Proceedings*, pages 406–416. ACM Press, September 1986.

[17] R. B. Kieburtz. Programming without pointer variables. In *Proceedings of the SIGPLAN '76 Conference on Data: Abstraction, Definition, and Structure*. ACM Press, 1976.

[18] A. Koenig and B. Moo. Teaching standard C++. *JOOP*, 11(7):11–17, 1998.

[19] G. W. Kulczycki, M. Sitaraman, W. F. Ogden, and J. E. Hollingsworth. Component technology for pointers: Why and how. Technical Report RSRG-2, Department of Computer Science, Clemson University, Clemson, SC, 29634, Mar. 2003.

[20] G. W. Kulczycki, M. Sitaraman, W. F. Ogden, B. W. Weide, and G. T. Leavens. Reasoning about procedure calls with repeated arguments and the reference-value distinction. Technical Report 02-13, Department of Computer Science, Iowa State University, Ames, Iowa, 50011, Dec. 2002.

[21] G. T. Leavens, A. A. Baker, and C. Ruby. JML: A notation for detailed design. In H. Kilov, B. Rumpe, and I. Simmonds, editors, *Behavioral Specifications of Businesses and Systems*, chapter 12. Kluwer, 1999.

[22] D. R. Musser, G. J. Derge, and A. Saini. *STL Tutorial and Reference Guide*. Addison-Wesley, Boston, 2nd edition, 2001.

[23] J. Noble, J. Vitek, and J. Potter. Flexible alias protection. *Lecture Notes in Computer Science*, 1445:158–185, 1998.

[24] S. M. Pike, B. W. Weide, and J. E. Hollingsworth. Checkmate: concerning C++ dynamic memory errors with checked pointers. In *Proceedings of the 31st SIGCSE Technical Symposium on Computer Science Education*. ACM Press, March 2000.

[25] G. J. Popek, J. J. Horning, B. W. Lampson, J. G. Mitchell, and R. L. London. Notes on the design of euclid. *ACM SIGPLAN Notices*, 12(3):11–18, March 1977.

[26] M. Sitaraman, S. Atkinson, G. Kulczycki, B. W. Weide, T. J. Long, P. Bucci, W. Heym, S. Pike, and J. E. Hollingsworth. Reasoning about software-component behavior. In *Procs. Sixth Int. Conf. on Software Reuse*, pages 266–283. Springer-Verlag, 2000.

[27] M. Sitaraman and B. W. Weide. Component-based software using RESOLVE. *ACM Software Engineering Notes*, 19(4):21–67, 1994.

[28] G. Smith. Reasoning about Object-Z specifications. In *Proceedings of Asia-Pacific Software Engineering Conference*, pages 489–497. IEEE Computer Society Press, December 1995.

[29] B. W. Weide and W. D. Heym. Specification and verification with references. In *Proceedings OOPSLA Workshop on Specification and Verification of Component-Based Systems*. ACM, October 2001.

[30] B. W. Weide, S. M. Pike, and W. D. Heym. Why swapping? In *Proceedings RESOLVE 2002 Workshop*, 2002. `http://people.cs.vt.edu/~edwards/RESOLVE2002/proceedings/Weide2.html`.

Appendix

This appendix contains two CleanJava classes. The first is a traversable list implemented using the Position class, and the second is a preemptable queue implemented using the List class. The classes include coding examples from each of the language areas discussed in this paper: instantiation, data movement, parameter passing, and linked data structures.

A Traversable list

A traversable list may be viewed abstractly as two strings—a left string and a right string, with a conceptual cursor position that is located between the two strings. The cursor is at the beginning of the list when the left string is empty, and the cursor is at the end of the list when the right string is empty. The list is "traversed" by advancing the cursor—taking one element from the beginning of the right string and placing it on the end of the left string. The following List class in CleanJava is implemented using the Position class and its associated factory, LocationSystem. Both are described in section 4.7. All methods take constant time.

```
public class List<Item> {

  // instantiate a location system with objects of
  // type Item and one link per location
  private LocationSystem<Item> sys :=
      LocationSystem<Item>.create(1);

  private Position<Item> head := sys.position();
  private Position<Item> pre := sys.position();
  private Position<Item> last := sys.position();
  private Integer leftlength, rightlength;

  // creation method
  public static List create() {
      head.new();
      pre.moveTo(head);
      last.moveTo(head);
  }

  public void insert(in x: Item);
      Position<Item> post := sys.position();
      Position<Item> newpos := sys.position();
      post.moveTo(pre);
      post.follow(1);
      newpos.new();
      newpos.swapObject(x);
      pre.redirect(1, q);
```

```
        newpos.redirect(1, post);
        if (rightlength = 0) { last.follow(1); }
        rightlength++;
}

public void remove(out x: Item) {
        Position<Item> oldpos := sys.position();
        Position<Item> newpost := sys.position();
        newpost.moveTo(pre);
        newpost.follow(1);
        oldpos.moveTo(newpost);
        newpost.follow(1);
        pre.redirect(1, newpost);
        oldpos.swapObject(x);
        oldpos.free();
        if (rightlength = 1) { last.moveTo(pre); }
        rightlength−−;
}

public void advance() {
        pre.follow(1);
        leftlength++;
        rightlength−−;
}

public void advanceToEnd() {
        pre.moveTo(last);
        leftlength := leftlength + rightlength;
        rightlength := 0;
}

public void reset() {
        pre.moveTo(head);
        rightlength := leftlength + rightlength;
        leftlength := 0;
}

public void swapRight(List s) {
        Position<Item> thispost := sys.position();
        Position<Item> spost := sys.position();
        thispost.moveTo(this.pre);
        thispost.follow(1);
        spost.moveTo(s.pre);
        spost.follow(1);
        pre.redirect(1, spost);
        s.pre.redirect(1, thispost);
```

```
        this.last :=: s.last;
        this.rightlength :=: s.rightlength;
        if (this.rightlength = 0) {
            this.last.moveTo(this.pre);
        }
        if (s.rightlength = 0) {
            s.last.moveTo(s.pre);
        }
    }

    public Integer leftLength() {
        return leftlength;
    }

    public Integer rightLength() {
        return rightlength;
    }
}
```

B Preemptable queue

A preemptable queue is one that allows an element to be injected at the beginning of the queue, and permits the examination of it's last element (here by allowing it to be swapped out and swapped back in again). An append method is included in this example. This queue is implemented with the List class shown above. Unlike the implementation above, which uses position variables and requires the programmer to reason abstractly about memory addresses, this implementation is entirely clean. All methods take constant time except for *append* and *swapLast*, which take linear time.

```
public class PreQueue<Item> {

    private List<Item> s;

    // creation method
    public static PreQueue create() { }

    public void enqueue(in x: Item);
        s.advanceToEnd();
        s.insert(x);
        s.reset();
    }

    public void dequeue(out x: Item) {
        s.remove(x);
    }
```

```
    public void inject(in x: Item) {
        s.insert(x);
    }

    public void append(PreQueue q) {
        Item x;
        while (q.length() != 0) {
            q.dequeue(x);
            this.enqueue(x);
        }
    }

    public void swapLast(Item x) {
        Item y;
        while (s.rightLength() > 1) {
            s.advance();
        }
        s.remove(y);
        x :=: y;
        s.insert(y);
        s.reset();
    }

    public Integer length() {
        return s.rightLength();
    }
}
```