Simplifying Reasoning about Objects with Tako

Gregory Kulczycki and Jyotindra Vasudeo
Virginia Tech, Falls Church, VA 22043
{gregwk, vasudeo}@vt.edu

ABSTRACT

A fundamental complexity in understanding and reasoning about object-oriented languages is the need for programmers to view variables as references to objects rather than directly as objects. The need arises because a simplified view of variables as (mutable) objects is not sound in the presence of aliasing. Tako is an object-oriented language that is syntactically similar to Java but incorporates alias-avoidance techniques. This paper describes the features of the Tako language and shows how it allows programmers to view all variables directly as objects without compromising sound reasoning. It discusses the benefits of such a language, including its use as an instructional tool to help teach students how to reason formally about their code.

Categories and Subject Descriptors

D.3.3 [**Programming Languages**]: Language Contructs and Features – *abstract data types, polymorphism, control structures.*

General Terms

Design, Education, Languages, Verification.

Keywords

Alaising, Semantics, Java, Tako

1. INTRODUCTION

References are pervasive in popular object-oriented languages. They permit efficient data assignment 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 use in such languages has frustrated students, programmers and formalists alike. As a result, significant research has focused on alias control techniques and alias-avoidance techniques for object-oriented languages [15].

Alias control techniques typically involve extending common object-oriented languages with annotations to ensure that certain types of aliasing do not occur [3][9][14][26]. They strive to conform as much as possible with a traditional style of object-oriented programming. Therefore, potentially aliased objects are still the norm, while alias-controlled objects are the exception.

In contrast, alias avoidance techniques typically involve a fundamental change to traditional object-oriented languages by replacing reference assignment—the primary cause of aliasing—with alternatives that do not introduce aliasing, such as value copying [3], destructive read [25], or swapping [12]. These approaches are also referred to as ones that use *unique references*, because in the implementation of languages that use them, each object must have exactly one reference to it. Despite the names *alias control* and *alias avoidance*, nearly all approaches to object aliasing—including ours—permits aliasing to some degree. In alias avoidance techniques, however, potential aliasing is the exception rather than the rule.

A common theme in languages that use alias control and even most alias avoidance techniques is that sound reasoning forces their semantics to be referenced-based. Variables that denote objects are viewed as mere references into a global heap, and method calls modify the heap abstraction rather than the abstract values of the variables (because the abstract *values* of the variables, according to the semantics, are *references*).

The language described in this paper, Tako, is different in this respect. It is intended to facilitate a simple value-based semantics called clean semantics [19] that has the following properties: (1) the state space is comprised of variables whose abstract values are objects rather than references, and (2) the effect of a method call is restricted to the abstract values of the variables involved: the arguments to the call and any relevant globals.

The key benefit of this approach is that it greatly simplifies reasoning about objects. Representing the state space abstractly is straightforward whether programmers are tracing through their code or reasoning about it symbolically. The fact that Tako supports a simple and sound view of the program state makes it particularly useful as an educational tool for introducing students to formal reasoning. From the perspective of object-oriented programming, a drawback of Tako is that it does not conform to some of the paradigms of traditional object-oriented programming. Despite this, Tako, like Java, contains all of the features traditionally found in object-oriented languages, such as classes, inheritance, and polymorphism.

Tako is essentially a redesign of Java that incorporates the alias-avoidance techniques found in Resolve. The Resolve language [30][31] is an integrated programming and specification language intended to support full, heavyweight program verification. For years, various universities including Ohio State, Clemson, and Virginia Tech have offered courses in which variants of Resolve have been used to introduce both undergraduate and graduate students to formal reasoning. Resolve has many features that facilitate formal verification, but as designers of Tako, we are primarily interested in the alias-avoidance features of Resolve and whether they can be successfully and independently applied to a traditional object-oriented language such as Java.

Section 2 of this paper introduces the features of Tako, with emphasis on how it differs from Java. Section 3 describes how

Tako supports clean semantics and facilitates reasoning, using examples similar to those we have used in courses at Virginia Tech. Section 4 discusses related work and future directions.

2. OVERVIEW AND FEATURES

As illustrated in Figure 1, a Tako stack implementation is syntactically similar to a Java stack implementation. They declare the same variables, they have similar methods, and, with one exception, they use the same keywords.

2.1 Data Assignment

The most important difference between the classes in Figure 1 is that wherever Java uses reference assignment—the main source of aliasing—Tako uses alternative data assignment mechanisms. As in the Resolve language, Tako uses swapping as its primary means of data assignment. In the body of the push method, the Java code assigns the object in *contents*[*top*] to *x* by copying its reference. But the Tako code uses a swapping operator (:=:) to swap the values of *contents*[*top*] with *x*. A call to Java's push method creates an alias between the incoming object *x* and the top element of the current stack; a call to Tako's push method transfers *x*'s object to the top of the stack, and replaces it with some unspecified but valid object of its type.

Swapping is described as simultaneous assignment in [17] and is proposed as an alternative to both reference and value copying in [12]. Swapping is a constant-time operation because a compiler can implement it by swapping object references; swapping preserves unique references, so programmers can reason about it as if object values are swapped. Swapping never creates aliases. For efficiency, a compiler can implement small objects such as integers and booleans directly as objects while implementing non-trivial objects using references.

Swapping is symmetric—it requires both objects to be of the same type. For assigning a type to a supertype, Tako provides an initializing transfer operator (⟵). The initializing transfer operation $s \longleftarrow c$ assigns circle c to shape s and initializes c to a new circle object. The approach does not introduce aliasing, but it does require the creation of a new object, and—because Tako's underlying implementation guarantees unique references—it ultimately requires memory from an old object to be reclaimed. Therefore, initializing transfer is less efficient than swapping.

2.2 Function Assignment

Both the swap statement and the transfer statement require a variable on either side of the operator. For assigning the result of an expression evaluation to a variable, Tako provides a function assignment operator (:=), as in the *new* expression that initializes the *contents* array in the stack's constructor in Figure 1. The same operator is used to copy values from one variable to another, as in the statement *MAX* := *n*.

Java methods return references rather than values, potentially introducing aliasing. For example, the Java *pop* method is a side-effecting function that returns a reference to *contents*[*top*], so that the assignment *x* = *s*.*pop*() causes two references to point to the same object. Tako avoids this problem with the introduction of a distinguished *result* variable that always holds the return value of a non-void method. The result variable is automatically initialized at the beginning of the method, and whatever object value it holds at the end of the method is returned. Since the result variable goes out of scope at the end of the method, we know that any returned reference (to the result variable's object) will continue to be unique. The syntax ***return*** ⟨*expression*⟩ is not permitted in Tako, though the *return* keyword may be used alone to denote that the program should return from the method with the current *result* value.

Consider the following Tako *pop* method written as a side-effecting function.

```
public Object pop() {
    top--;
    result :=: contents[top];
}
```

The result variable gets a new initial Object as soon as the

```
public class BddStack {
    private final int MAX;
    private Object[] contents;
    private int top;
    public Stack(int n) {
        MAX := n;
        contents := new Object[MAX];
    }
    public void push(Object x) {
        assert depth() < MAX;
        contents[top] :=: x;
        top++;
    }
    public void pop(Object x) {
        assert depth() > 0;
        top--;
        x :=: contents[top];
    }
    public int depth() {
        result := top;
    }
}
```

```
public class BddStack {
    private final int MAX;
    private Object[] contents;
    private int top;
    public Stack(int n) {
        MAX = n;
        contents = new Object[MAX];
    }
    public void push(Object x) {
        assert depth() < MAX;
        contents[top] = x;
        top++;
    }
    public Object pop() {
        assert depth() > 0;
        top--;
        return contents[top];
    }
    public int depth() {
        return top;
    }
}
```

Figure 1. A Tako bounded stack implementation (left) compared to a Java bounded stack implementation (right)

method is invoked, as if the first statement were *Object* ***result*** := ***new*** *Object*(). After the swap operation, *result* holds the object originally held by *contents*[*top*] and *contents*[*top*] holds the newly created object originally held by *result*. Thus, the method returns the object originally at the top of the current stack. Internally, it places a new initialized object in the cell of the *contents* array that previously held the stack's top element.

The function assignment operator can also be used to copy objects. The compiler expects the function assignment operator to have a variable on the left-hand side and an expression on the right-hand side. If it does not find a variable on the left-hand side, it will report an error. However, if it finds a variable rather than an expression on the right-hand side, it will check to see if a *replica* method has been implemented for the variable's type. If so, it will call the replica method for the variable's object; if not, it will report that no replica method could be found. Thus, the compiler considers the statement *s* := *t* to be special syntax for *s* := *t.replica*(). The replica method is intended to be used for small objects where copying is the preferred form of data assignment, such as integers and booleans. In principle, though, any Tako class can be extended with a replica method.

2.3 Parameter Passing

Parameter passing in Java is accomplished by copying the references of the arguments to the formal parameters without copying them out again. This approach is problematic for a language that intends to facilitate value semantics, because the semantics that describe this form of parameter passing are difficult to formulate without introducing the notion of reference. Java parameter passing cannot be viewed as in-out because assignments to formals inside the body are not reflected in the actuals. It cannot be viewed as copying object values in only, because in Java you *can* update an argument's conceptual object value as long as you do not change its reference value.

Tako avoids this difficulty by fully supporting in-out (also called value-result) parameter passing. Conceptually, programmers can reason about in-out parameter passing as if object values are copied into the method, the method is executed, and object values are copied out again. However, a compiler can implement in-out parameter passing efficiently by copying references to the formal parameters, executing the method, and copying references back out, which is what C# does with *ref* parameters.

The effects of in-out parameter passing with references and in-out parameter passing with objects are semantically equivalent whenever the arguments are not aliased [11]. Since Tako avoids aliasing, in-out parameter passing is an appropriate choice. Note that Tako uses in-out parameter passing by default for all parameters, even the current *this* parameter, which cannot be a *ref* parameter in C#.

Using in-out parameter passing gives programmers the option of writing certain methods as procedures (i.e., void methods) rather than side-effecting functions. For example, the Tako *pop* method is a procedure whereas the Java *pop* method must be written as a side-effecting function. A function is free of side-effects if its execution does not change the program state. Keeping functions free from side-effects simplifies reasoning about programs that include conditions, as in if and while statements.

One aliasing problem that in-out parameter passing does not solve is the repeated argument problem [19]. As long as parameter passing is implemented by copying references—whether in-out or in-only—aliasing can be introduced when arguments are repeated in a call. For example, the call *q.append*(*q*) introduces aliasing between the implicit formal parameter *this* and the explicit formal parameter in the body of the append method. The call *a*[*i*].*append*(*a*[*j*]) does the same when *i* and *j* are equal.

We are currently exploring alternative designs for handling repeated arguments. One option is to throw a runtime error when repeats occur, another is to initialize the second and subsequent repeats, as described in [19]. In either case, the compiler will warn programmers when arguments are potentially repeated.

Tako includes an *eval* parameter mode that indicates that a function is expected for evaluation. The *eval* mode is often used for small types such as integers and booleans. 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. Since the result of an expression evaluation is always a new object, repeated arguments do not pose a problem for eval parameters.

A potential problem with in-out parameter passing and Java-like inheritance concerns the passing of subtypes. If *c* has type *Cat* and *d* has type *Dog*, what should the effect be of "*s.push*(*c*); *s.pop*(*d*);"? If we blindly permit this, it results in the dog variable d holding a cat object, causing a type violation. Some object-oriented languages that support the conceptual equivalent of in-out parameters (such as C#) do not allow programmers to pass subtype objects to them. In Tako, this is not an option—particularly since Tako does not yet support generics. The stack class in Figure 1 would be of little value if we were restricted to populating the stack with objects of type *Object*.

One option is that when a parameter is transferred back, an implicit cast is done. If the cast cannot be made, a runtime error occurs. Due to the poor performance of runtime casts in Java, this solution, though adequate, is not the most efficient, and we are currently exploring alternatives.

2.4 Initialization

As in the case of the initializing transfer operator, Tako sometimes requires the compiler to automatically create new, initialized objects. This is done for newly declared variables as well. In Java, the declaration "*Circle c*;" does not initialize *c*, and if *c* is not assigned to before it is used, a compile-time error occurs. In Tako, the declaration *Circle c* is interpreted by the compiler as *Circle c* := ***new*** *Circle*().

Types that do not have a default constructor get initialized to null. Tako tries to facilitate a view of all variables as objects, so including null values in the language may seem odd. Technically, a value semantics can accommodate null values by introducing them as distinguished "object" values. Specifications are complicated when null values are permitted [7]. However, types derived from interfaces cannot have constructors, and some classes, such as the bounded stack class in Figure 1, effectively require parameters in their constructors. Therefore, in the current version of Tako, types may be nullable or non-nullable, based on whether a default constructor is provided for the class. Non-nullable classes are encouraged because they simplify reasoning. Null pointer exceptions do not occur with non-nullable classes.

2.5 Pointer Component

One of the primary motivations for Tako is the simplification of object-based reasoning through alias avoidance. By providing efficient alternatives to data assignment and parameter passing that avoid aliasing, Tako supports the construction of typical classes as pure value types, allowing programmers to reason about variables directly as objects. There are circumstances, however, in which aliasing cannot be avoided without sacrificing efficiency. For example, references and aliasing are needed to implement all the methods in the list component in constant time. For the efficient implementation of lists and other typically linked data structures, Tako provides a pointer component that models a system of linked locations. Each location holds data (objects) and has a fixed number of links to other locations. Position variables reside at the various locations. In the following Position interface, k is the number of outgoing links at each location.

```
  interface Position {
    static final int k;
    public void takeNew();
    public void moveTo(Position p);
    public void redirectLink(int k, Position p);
    public void followLink(int k);
    public void swapContents(Object x);
    public boolean isWith(Position p);
    public boolean isAtVoid();
  }
```

Presenting pointers in the form of a class has the advantage that programmers can reason about pointers the same way they reason about any other object. No special proof rules are needed for pointers, and no universal heap structure needs to be included in the semantics. Programmers can maintain a sound view of position variables as abstract object values just as they can with any other variable in Tako.

```
allocate p;      p.takeNew();
  p -> q;          p.moveTo(q);
  p -> q^PREV;     p.moveTo(q); p.follow(PREV);
  p^NEXT -> q;     p.redirectLink(NEXT, q);
  p <-> q;         p :=: q;
  p *:=: s;        p.swapContents(s);
```

Figure 2. Special syntax for position objects

The Position interface provides a way for programmers to view pointers in a value-based reasoning environment. However, the compiler cannot implement a Position class as it can other classes because position variables must not only provide the functional benefits of pointers but the performance benefits as well. For example, the call *p.moveTo*(*q*) moves *p* to *q*'s location, effectively resulting in *p* and *q* being aliases. Although the programmer can reason about the statement as a method call, the Tako compiler will implement it by copying a single reference.

With the help of the special syntax shown in Figure 2, the implementation of linked data structures using Tako pointers has a relatively straightforward translation into Java, as illustrated in Figure 3. The Tako pointer component shares many similarities with Resolve's *Location_Linking_Template*, whose specification and reasoning in terms of clean semantics is detailed in [20].

3. VALUE-BASED REASONING

This section describes the properties of the value-based reasoning system that Tako facilitates and presents an example of specification and verification using Tako.

3.1 Clean Semantics

The value-based semantics for Tako should have the following two properties. First, the state space should be based on the object values of variables. Specifically, at any point in the program, the state consists of the abstract object values of the currently defined programming or conceptual variables. Conceptual variables are similar to model variables or data groups [8][22], and they are used to help model the program state. An example of their use is given in the mathematical model of the reference-based stack component below. The second property of our semantics is a frame property [6]. It states that the portion of the state space that can be modified by a method call is restricted to certain syntactically discernible variables—the arguments to the call and any global (static) variables listed in the *affects* clause of the method's declaration.

Together, the *variable-based* property and the *effects-restricted* property define the behavior of a clean semantics as given in [19]. This notion was proposed as a syntactic yet formalizable way to capture the notion of localized reasoning about operation

```
public class LinkedList {
    class Node is Object^(NEXT);
    private Node head, pre, last;
    private int left_length, right_length;

    public LinkedList() {
        allocate head;
        pre -> head;
        last -> head;
    }

    public void insert(Object x) {
        Node post, new_pos;
        post -> pre^NEXT;
        allocate new_pos;
        new_pos *:=: x;
        pre^NEXT -> new_pos;

        ...
```

```
public class LinkedList {
    class Node {
        Node next = null;
        Object contents = new Object();
    }
    private Node head, pre, last;
    private int left_length, right_length;

    public LinkedList() {
        head = new Node();
        pre = head;
        last = head;
    }

    public void insert(Object x) {
        Node post, new_pos;
        post = pre.next;
        new_pos = new Node();
        new_pos.contents = x;
        pre.next = new_pos;

        ...
```

Figure 3. A portion of a linked list implementation using Tako (left) compared to one using Java (right)

invocations. The potential for aliasing in a programming language complicates reasoning and makes clean semantics harder to achieve. However, a system may permit aliasing and still conform to clean semantics, as with Tako's pointer component.

3.2 Simple Stack Specification

This section describes the specification and reasoning for a Tako stack component. The stack component is a typical Tako component because it specifies a single mathematical model for its type and does not require any conceptual state variables to describe its behavior. We have used this example and others like it to introduce graduate students to formal reasoning in courses not normally associated with formal methods, such as software engineering, and theory of algorithms. The software engineering course covers general topics, but approximately the last 25% of the course focuses on component-based software engineering and formal methods. Tako code is used to illustrate key principles. The algorithms course uses the Cormen et al. text [10], whose latest edition puts greater emphasis on demonstrating the correctness of algorithms and includes discussions on loop invariants. Their "proofs" of correctness are typical mathematical proofs given in natural language. We occasionally augment these proofs using formally specified Tako components and demonstrate formal correctness through a symbolic reasoning table, like the one described below.

```
import spec.MathString;

public interface Stack {

   model MathString;

   initialization ensures
        this = EMPTY_STRING;

   public void push(Object x);
     ensures this = <#x> o #this;

   public void pop(Object x);
     requires |this| > 0;
     ensures #this = <x> o this;

   public int depth();
     ensures this = #this and result = |this|;

}
```

Figure 4. A specification for a Tako stack

Figure 4 gives a specification for an unbounded Tako stack. The *model* clause indicates that an object of type Stack is modeled as (has a conceptual value of) a mathematical string of objects. The clause does not specify a variable name as the current stack *this* is implied. A string is similar to a sequence except that it is not indexed. The *initialization ensures* clause gives the behavior of the default (no-argument) constructor, which in this case guarantees that an initial stack will be empty.

A Java stack specified in JML (the Java Modeling Language [21]) would likely be modeled using a JMLObjectSequence. JML provides three different sequences depending on weather the sequence holds object types (references to objects), equals types (non-clonable objects), or value types (object values). Variables in Tako always denote object values, so all mathematical structures hold value types, eliminating the need for such a distinction.

A hash mark (#) is used only in an ensures clause—it denotes the incoming value of a variable. The expression $\langle e\rangle$ is a unary string containing the element *e*, and the symbol o denotes string concatenation. So the ensures clause for *push* states that the current stack is equal to the string containing the original value of *x* concatenated with the original stack value. Notice that the outgoing value of *x* is left unspecified. The method is specified this way partly because of the new paradigm for component construction in Tako. If we specified that the outgoing value of *x* was the same as the incoming value of *x* (*x* = #*x*) we would effectively force the implementer to make a deep copy of *x*. When we don't specify how *x* changes, the reasoning system guarantees only that *x* contains a valid value of its type.

Only actual parameters and global variables listed in an *affects* clause may be modified by a method, so Tako does not provide a *modifies* or *assignable* clause as JML does. None of the stack methods modifies any global state variables, so none of them have an affects clause.

In the *depth* method, the keyword ***result*** denotes the return value. Also, since the current stack (***this***) is considered the first parameter to the call, our frame property states that its value *may* be modified. But we do not want the function to have side-effects, so we must explicitly state in the ensures clause that it is *not* modified.

Once students are introduced to formal specification, they practice reading the specifications by tracing through code. The following tracing table (Table 1) gives an example. Students are given variable values for state 0 and asked to fill in the other states. The assertion *x* = ?? indicates that *x* is a valid but unspecified value of its type.

Table 1. Tracing table for simple stack code

St	Facts
0	s = ⟨ 4, 5 ⟩ **and** t = ⟨ 7, 8, 9 ⟩ **and** x = 3
t :=: s;	
1	s = ⟨ 7, 8, 9 ⟩ **and** t = ⟨ 4, 5 ⟩ **and** x = 3
t.push(x);	
2	s = ⟨ 7, 8, 9 ⟩ **and** t = ⟨ 3, 4, 5 ⟩ **and** x = ??

3.3 More Complex Stack Specification

The fact that the potential for aliasing does not exist greatly simplifies our ability to represent and understand the state space. Consider the tracing table in Table 2 for similar code with the swap statement replaced by an assignment. If this were Java code, we would know that the object value of *s* in state 2 is ⟨ 3, 4, 5 ⟩ since *s* and *t* point to the same object. But if we assume that variables denote strict object values we will conclude that *s* = ⟨ 4, 5 ⟩ is unchanged, making the simple value-based specification unsound in the presence of aliasing.

Table 2. Problematic tracing table for code with aliasing

St	Facts
0	s = ⟨ 4, 5 ⟩ **and** t = ⟨ 7, 8, 9 ⟩ **and** x = 3
t = s;	
1	s = ⟨ 4, 5 ⟩ **and** t = ⟨ 4, 5 ⟩ **and** x = 3
t.push(x);	
2	s = /* *what goes here?* */ **and** t = ⟨ 3, 4, 5 ⟩ **and** x = ??

We can remedy this by giving the stack a specification that accounts for aliasing, such as the one in Figure 5. Here, stack variables are modeled as locations, and the conceptual variable *obj* maps locations to mathematical strings (conceptual stack objects). Like to model variables in JML, conceptual variables do not correspond to programming objects, but they are necessary for reasoning about the component. The conceptual variable *obj* is a global variable, so it must be listed in the affects clause of any method (such as push) that potentially modifies its value.

With this specification, we can reason soundly about the stack component in the presence of aliasing, as shown in Table 3. However, even this specification is an oversimplification of object-oriented logics as it does not account for the effects of (future) inheritance.

```
import spec.MathString;
import spec.Location;

public interface Stack {

   var obj: Location → MathString;
   model Location;

   public void push(Object x);
      affects obj;
      ensures this = #this and
         obj(this) = <#x> o #obj(this) and
         ∀r: Stack, if r ≠ this then
            obj(r) = #obj(r);
```

Figure 5. Portion of a reference-based stack specification

We can simplify the appearance of the specification in Figure 5 to something resembling Figure 4 by indicating—as JML does—that all variables have reference semantics. This approach, however, will not simplify the states in our tracing table. There is no rule that can tell us how to transition from the state "s = ⟨ 4, 5 ⟩ **and** t = ⟨ 4, 5 ⟩ **and** x = 3" through *s.push*(*x*), to the next state, without telling us whether *s* and *t* are aliased.

Table 3. Sound tracing table for stack code with aliasing

St	Facts
0	s = @47 **and** t = @53 **and** x = 3 **and** contents = { @47 |→ ⟨ 4, 5 ⟩, @53 |→ ⟨ 7, 8, 9 ⟩ }
t = s;	
1	s = @47 **and** t = @47 **and** x = 3 **and** contents = { @47 |→ ⟨ 4, 5 ⟩, @53 |→ ⟨ 7, 8, 9 ⟩ }
s.push(x);	
2	s = @47 **and** t = @53 **and** x = 3 **and** contents = { @47 |→ ⟨ 3, 4, 5 ⟩, @53 |→ ⟨ 7, 8, 9 ⟩ }

3.4 Reasoning about Stack Reverse

Consider the specification and implementation for the stack reverse method in Figure 6. The method has no precondition, and the postcondition states that the mathematical string that models the stack will be reversed. The implementation pops elements one at a time from the current stack and pushes them onto a temporary stack. Before the method returns, the current stack is swapped with the temporary stack. We want to reason about the correctness of this implementation with respect to its specification, so we include an invariant for the loop. The decreasing clause allows us to prove that the loop terminates.

A tracing table for the reverse method is given below. It demonstrates that when the current stack has a value of ⟨ 3, 4 ⟩, the reverse method will change its value to ⟨ 4, 3 ⟩, satisfying the postcondition of the reverse method.

Once students are comfortable with tracing tables, we introduce them to symbolic reasoning [13][30]. A symbolic reasoning table for the reverse procedure is given in Table 5. For each state, the table shows a path condition, facts, and obligations. The path condition must hold for the program to enter the specified state, the facts tell us what we know about the values of the variables in that state, and the obligations tell us what needs to be true before we can move to the next state.

```
public void reverse()
        ensures this = REV(#this);
    {
        Stack temp;
        Object x;
        while (this.depth() != 0)
           decreasing |this|;
           maintaining REV(temp) o this = #this;
        {
           this.pop(x);
           temp.push(x);
        }
        this :=: temp;
    }
```

Figure 6. Specification and implementation of stack reverse

In general, obligations come from preconditions of called operations and facts come from their postconditions. For example, the requires clause of the pop method indicates that the stack must be non-empty. All variables are indexed with the current state, so in state 1 we have an obligation to show that $|\boldsymbol{this}_1| > 0$. The pop method ensures that the old value of the stack is equivalent to the new value of the *x* parameter concatenated with the new value of the stack. So in state 2 we have the fact that $\boldsymbol{this}_1 = \langle x_2 \rangle$ o $\boldsymbol{this}_2$. We also know that $temp_2 = temp_1$ in accordance with the frame property.

Table 4. Tracing table for stack reverse method

St	Facts
0	**this** = ⟨ 3, 4 ⟩ **and** temp = ⟨ ⟩ **and** x = 0
while (**this**.length() != 0) {	
1	**this** = ⟨ 3, 4 ⟩ **and** temp = ⟨ ⟩ **and** x = 0
this.pop(x);	
2	**this** = ⟨ 4 ⟩ **and** temp = ⟨ ⟩ **and** x = 3
temp.push(x);	
3	**this** = ⟨ 4 ⟩ **and** temp = ⟨ 3 ⟩ **and** x = ??
// **this**.length() != 0	
1′	**this** = ⟨ 4 ⟩ **and** temp = ⟨ 3 ⟩ **and** x = ??
this.pop(x);	
2′	**this** = ⟨ ⟩ **and** temp = ⟨ 3 ⟩ **and** x = 4
temp.push(x);	
3′	**this** = ⟨ ⟩ **and** temp = ⟨ 4, 3 ⟩ **and** x = ??
} // **this**.length() = 0	
4	**this** = ⟨ ⟩ **and** temp = ⟨ 4, 3 ⟩ **and** x = ??
this :=: temp;	
5	**this** = ⟨ 4, 3 ⟩ **and** temp = ⟨ ⟩ **and** x = ??

The facts in state 0 come from the precondition (if any) of the method you are trying to prove correct, and the obligations in the last state come from the method's postcondition.

A reasoning table for code that involves a loop is slightly more complex than one that does not. It effectively breaks up the table into three separate sub-tables dedicated to proving the following three properties: *initialization* – the invariant is true when the loop first begins; *maintenance* – if the invariant holds at the beginning of the *n*-th iteration, it also holds at the end of the (*n*+1)-th iteration; and *termination* – the invariant and the negation of the while condition allow you to prove what you want to prove (in our case, the postcondition of the reverse method).

Table 5. Symbolic reasoning table for stack reverse method

St	P Cond	Facts	Obligations
0		Object.**is_init**(x_0) **and** Stack.**is_init**($temp_0$) **and** **this**$_0$ = #**this**	$\lvert$**this**$_0\rvert \neq 0 \Rightarrow$ REV($temp_0$) o **this**$_0$ = #**this**
while (**this**.length() != 0) {			
1	$\lvert$**this**$_1\rvert \neq 0$	REV($temp_1$) o **this**$_1$ = #**this and** x_1 = ??	$\lvert$**this**$_1\rvert > 0$
this.pop(x);			
2	$\lvert$**this**$_1\rvert \neq 0$	**this**$_1 = \langle x_2 \rangle$ o **this**$_2$ and $temp_2 = temp_1$	
temp.push(x);			
3	$\lvert$**this**$_1\rvert \neq 0$	**this**$_3 = \langle x_2 \rangle$ o **this**$_2$ **and** x_3 = ?? **and** $temp_3 = temp_2$	REV($temp_3$) o **this**$_3$ = #**this and** $\lvert$**this**$_3\rvert < \lvert$**this**$_1\rvert$
}			
4	$\lvert$**this**$_4\rvert = 0$	REV($temp_4$) o **this**$_4$ = #**this and** x_4 = ??	
this :=: temp;			
5	$\lvert$**this**$_4\rvert = 0$	**this**$_4 = temp_4$ **and** $temp_5$ = **this**$_4$ **and** $x_5 = x_4$	**this**$_5$ = REV(#**this**)

The obligation in state 0 must be discharged to prove the initialization property. Discharging the first part of the obligation in state 3 proves maintenance. And discharging the obligation in state 5 proves the termination property. Note that the second part of the obligation in state 3 comes from the decreasing clause. It must be discharged to prove that the while loop terminates.

The obligations may be discharged with a theorem prover, but they may also be simple enough for students and programmers to reason about themselves. Take, for example, the obligation in state 5. We want to prove that ***this***$_5$ = *REV*(#***this***). We know from the facts in state 5 that ***this***$_5$ = *temp*$_4$, so it suffices to show that *temp*$_4$ = *REV*(#***this***). We know from the facts in state 4 that *REV*(*temp*$_4$) o ***this***$_4$ = #***this***, and we know from the path condition that $\lvert$***this***$_4\rvert = 0$ which can only happen if *this*$_4$ is empty. So we know *REV*(*temp*$_4$) = #***this***. Hence, *temp*$_4$ = *REV*(#***this***).

4. DISCUSSION

The difficulty of reasoning in the presence of aliasing is well known [15], and numerous techniques to control aliasing in object-based languages have been proposed [2][3][9][23]. We have used the term alias avoidance to refer to techniques that promote alias-free alternatives to reference assignment, such as the approach based on destructive read in [25] and the approach based on swapping in [12]. The term uniqueness has generally come to refer to techniques such as [25] and variations that employ the destructive read operator. They preserve unique references to objects, but they also support a borrowing mechanism that allows programmers to violate the uniqueness condition when they deem it useful. Borrowing raises the potential for aliasing and is therefore not conducive to value semantics. However, there is nothing fundamental about the destructive read operator that prevents it from being used in a language that does support value-based reasoning.

Most proposals for controlling object aliasing attempt to minimize the impact of their approach on programmers who have become accustomed to an object-oriented style of programming. While some practitioners have reported positive experiences when using a swap operator in object-based applications [16], we understand that allowing programmers to view all classes as value types is a radical departure from the traditional object-oriented paradigm. Among other things, it removes the distinction between primitive types and user-defined types; it forces programmers who want to modify an object inside a container to remove, modify, and re-insert it; and it requires programmers to rethink common design patterns whose implementations traditionally use aliasing, such as the singleton and the observer patterns. The question of whether programmers accustomed to traditional object-oriented paradigms can make the transition to object-oriented languages that support direct reasoning can only be answered empirically. The desire to answer this question is one of our primary motivations for creating the Tako compiler.

The current focus for the Tako language is its use as an educational tool to introduce students to formal reasoning. But we would like to develop it into a practical programming language that could be used alone or with Java components to develop non-trivial applications. Practical concerns include compiler optimizations, especially in the areas of automatic initialization and automatic casting. The current implementation of the Tako compiler is available at SourceForge under the name *takocompiler*. We have developed a medium-sized application (about 40 classes) in Tako that interfaces with Java Swing components. We have no immediate plans to see how other object-oriented languages might benefit from alias-avoidance, but we would consider it a worthy long-term pursuit. In-out parameter passing is easier to implement on the .NET platform than the JVM, making C# appealing for our research, and Eiffel's value-based expanded types may serve as a basis for alias avoidance.

From a research perspective, we are still exploring the impact that alias avoidance and clean semantics will have on advanced language features such as inheritance and concurrency. To this end, we hope to leverage both Resolve research and the ongoing research on specification and verification of Java-like languages. We are exploring using Tako in the context of both lightweight and full verification. The general trade-offs on verification rigor are described in [33]. The ultimate goal might be a verifying compiler that—due to the relative simplicity of Tako's semantics—could be much more automated than a comparable tool for Java, and would generate verified Java byte code.

5. ACKNOWLEDGMENTS

Our thanks to Murali Sitaraman, Bill Ogden, Bruce Weide and other members of the Reusable Software Research Group for their valuable insights into the specification and verification of object-based languages.

6. REFERENCES

[1] Abadi, M. and Leino, K.R.M. A logic of object-oriented programs. Dauchet, M. ed. In *Proceedings TAPSOFT '97*, pages 682-696. Springer-Verlag, New York, 1997.

[2] Aldrich, J., Kostadinov, V. and Chambers, C., Alias annotations for program understanding. In *Proceedings OOPSLA '02*, pages 311-330. ACM Press, 2002.

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

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

[5] Bokowski, B. and Vitek, J., Confined types. In *Proceedings OOPSLA '99*, pages 82-96. ACM Press, 1999.

[6] Borgida, A., Mylopoulos, J. and Reiter, R., ... And nothing else changes?: The frame problem in procedure specifications. In *Proceedings of the 15th International Conference on Software Engineering*, pages 303-314. IEEE Computer Society Press, 1993.

[7] Chalin, P. and Rioux, F., Non-null references by default in the Java Modeling Language. In *Proceedings SAVCBS '05*, pages 70-76. 2005.

[8] Cheon, Y., Leavens, G.T., Sitaraman, M. and Edwards, S. Model variables: Cleanly supporting abstraction in design by contract. *Software: Practice and Experience*, *35* (6), pages 583-589. 2005.

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

[10] Cormen, T.H., Leiserson, C.E., Rivest, R.L., and Stein, C., *Introduction to Algorithms*, 2nd ed., McGraw-Hill, 2001.

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

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

[13] Heym, W. *Computer Program Verification: Improvements for Human Reasoning*, Ph.D. thesis, The Ohio State University (1995)

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

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

[16] Hollingsworth, J.E., Blankenship, L. and Weide, B.W., Experience report: Using Resolve/C++ for commercial software. In *Proceedings FSE '00*, pages 11-19. ACM Press, 2000.

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

[18] Krone, J. *The Role of Verification in Software Reusability*, Doctoral Thesis, The Ohio State University, 1988.

[19] Kulczycki, G., Sitaraman, M., Ogden, W.F. and Weide, B.W. *Clean Semantics for Calls with Repeated Arguments*, Technical Report RSRG-05-01, Clemson University, 2005.

[20] Kulczycki, G., Sitaraman, M., Weide, B. and Rountev, N., A specification-based approach to reasoning about pointers. In *Proceedings SAVCBS '05*, pages 55-62. 2005.

[21] Leavens, G.T., Baker, A.A. and Ruby, C. JML: A notation for detailed design. Simmonds, I. ed. *Behavioral Specifications of Businesses and Systems*, Kluwer, 1999.

[22] Leino, K.R.M., Data groups: Specifying the modification of extended state. In *Proceedings OOPSLA '98*, pages 144-153, ACM Press, 1998.

[23] Meyer, B., *Object-Oriented Software Construction*, 2nd ed. Prentice Hall, 1997.

[24] Müller, P. and Poetzsch-Heffter, A. Modular specification and verification techniques for object-oriented software components. Sitaraman, M. and Leavens, G. eds. *Foundations of Component-Based Systems*, Cambridge University Press, Cambridge, United Kingdom, 2000.

[25] Minsky, N.H., Towards alias-free pointers. In *Proceedings ECOOP '96*, pages 189-209. 1996.

[26] Noble, J., Vitek, J. and Potter, J. Flexible alias protection. *Lecture Notes in Computer Science*, *1445*. pages 158-185. 1998.

[27] Ogden, W.F. *The Proper Conceptualization of Data Structures.* The Ohio State University, Columbus, OH, 2000.

[28] O'Hearn, P., Reynolds, J. and Yang, H. Local reasoning about programs that alter data structures. *Lecture Notes in Computer Science*, *2142*, 2001.

[29] Popek, G.J., Horning, J.J., Lampson, B.W., Mitchell, J.G. and London, R.L. Notes on the design of Euclid. *ACM SIGPLAN Notices*, *12* (3), poages 11-18. 1977.

[30] Sitaraman, M., Atkinson, S., Kulczycki, G., Weide, B.W., Long, T.J., Bucci, P., Heym, W., Pike, S. and Hollingsworth, J.E., Reasoning about software-component behavior. In *Proceedings ICSR '00*, pages 266-283. Springer-Verlag, 2000.

[31] Sitaraman, M. and Weide, B.W. Component-based software using Resolve. *ACM Software Engineering Notes*, *19* (4), pages 21-67. 1994.

[32] Weide, B.W. and Heym, W.D., Specification and verification with references. In *Proceedings SAVCBS '01*. 2001.

[33] Wilson, T., Maharaj, S., and Clark, R.G., Omnibus verification policies: a flexible, configurable approach to assertion-based software verification. In *SEFM '05*. IEEE Press. 2005.