Reasoning about Procedure Calls with Repeated Arguments and the Reference-Value Distinction

Gregory W. Kulczycki, Murali Sitaraman,
William F. Ogden,
and Gary T. Leavens

TR #02-13a,
December 2002, revised December 2003

Keywords: aliasing, language design, parameter passing, proof rules, specification, verification.

2002 CR Categories: D.2.1 [*Software Engineering*] Requirements/Specifications — languages; D.2.4 [*Software Engineering*] Software/Program Verification — correctness proofs, formal methods, programming by contract, reliability; D.3.1 [*Programming Languages*] Formal Definitions and Theory — semantics; D.3.3 [*Programming Languages*] Language Constructs and Features — Procedures, functions, and subroutines; F.3.1 [*Logics and Meanings of Programs*] Specifying and Verifying and Reasoning about Programs — Assertions, logics of programs, pre- and postconditions, specification techniques; F.3.3 [*Logics and Meanings of Programs*] Studies of Program Constructs — Functional constructs, object-oriented constructs.

Submitted for publication.

Department of Computer Science,
226 Atanasoff Hall
Iowa State University,
Ames, IA 50011-1041 USA

Preserving Clean Semantics for Calls with Repeated Arguments

Gregory W. Kulczycki, Murali Sitaraman
Computer Science, Clemson University, Clemson, SC 29634-0974 USA
gregwk@cs.clemson.edu, murali@cs.clemson.edu

William F. Ogden
Computer and Information Science, The Ohio State University, Columbus, OH 43210-1277 USA
ogden@cis.ohio-state.edu

Gary T. Leavens
Department of Computer Science, Iowa State University, Ames, IA 50011-1041 USA
leavens@cs.iastate.edu

Abstract

In a language with clean semantics, the effect of a call to an operation is local; this effect-restrictive property makes it easy for software engineers to understand and reason about their code. However, in order to give clean semantics for procedure calls in the presence of aliasing, it is necessary to view variables that refer to complex objects as mere references into a global store. The reasoning difficulties this indirection introduces do not disappear even when a language designer or a disciplined software engineer avoids explicit assignment of references – the more common source of aliasing. This is because of an independent source of aliasing that arises when procedures are called with repeated arguments and references are copied for parameter passing. This repeated argument problem exists in all well-known imperative languages. We examine the software engineering issues in solving the repeated argument problem, discussing in the process the reasoning problems introduced by aliasing and the benefits of preserving clean semantics. A key design consideration is avoiding value copying, both because it is inefficient and because it cannot, in general, be automated.

Key words: aliasing, call-by-reference, language design, parameter passing, procedure calls, proof rules, specification, and verification.

1. Introduction

References, or pointers, and a simple way to copy them using an assignment statement, are part of most modern imperative programming languages. Examples include languages such as C++, and Java. These features are convenient and efficient: copying a reference can be done automatically (without the user writing any extra code) and in a constant time. In contrast, copying an object's

entire representation is more expensive and cannot be automated in a way that works appropriately for all objects [18].

Copying references, though efficient, leads to aliasing. Aliasing is a well-known source of reasoning complexity in principle and in practice [10][19][23][26][58]. As early as 1973, Hoare [23] warned about pointers that, "their introduction into high-level languages has been a step backward from which we may never recover" (p.37). In their paper on object aliasing, Hogg *et al.* [26] point out that aliasing can make it "annoyingly difficult to prove the simple Hoare formula {x = true} y := **false** {x = true}" (p.11), because x and y may be aliased to the same Boolean object. In general, aliasing among mutable objects causes dependencies between such syntactically unrelated expressions, and forces programmers to reason about the global store and about locations within it [58], instead of reasoning locally about variables and their values. Various techniques for avoiding and minimizing aliasing from assignments are discussed in a section on related work.

While the introduction of aliasing through any means can complicate reasoning about software behavior, we focus here on the aliasing that arises when procedures are called with repeated arguments and references are copied as a means of passing parameters. Repeated arguments may be *explicit* as in the call p(x,x) or *implicit* when indexed array variables or global variables are passed as arguments, as in p(a[i],a[k]), when i equals k, or p(g), where g is a global variable already in the scope of p. Inside the body of a procedure called with repeated arguments, two or more different formal parameters become aliased, because references are copied in parameter passing. We address this repeated argument problem in this paper, because in addition to helping focus the discussion on aliasing, it is a problem in its own right. It does not disappear even when a language designer or a disciplined software engineer avoids explicit assignment of references – the most common source of aliasing. It exists in every well-known imperative language from FORTRAN I – which includes no reference variables or reference assignments – to Java, because they all pass parameters in procedure calls by copying references and also allow arguments to be repeated. As early as 1978, while considering the scenario in which a global variable is passed by reference as a parameter to a procedure that uses it, Cook [10] concluded that procedure calls with repeated arguments render Hoare logic unsound.

It is possible to specify and verify software behavior soundly in the presence of aliasing by viewing objects as references to a global store and by using suitable frame axioms to capture the non-local impact of procedure calls, but such an approach leaves the reasoning complexity intact, as noted later in this paper and elsewhere [49]. Consequently, we will take a different approach. We observe that reasoning about software behavior is much simplified when the effects of procedure calls are restricted to be local and objects can be viewed as something more natural than references. We capture this idea in a software engineering notion of languages having clean semantics in Section 2. We note that once a language allows aliasing to be introduced through repeated arguments (or by other means), it is no longer possible to give clean semantics for subsequent calls – neither for routine calls where there is no potential for argument repetition nor for single argument calls that use global variables. We discuss various aspects of the repeated argument problem and associated aliasing, and explain the objective of preserving clean semantics for procedure calls while allowing repeated arguments. Section 3 contains a solution to the problem that is based on object initialization. Section 4.1 explains how clean semantics can be preserved when preventing repeated arguments syntactically or aborting execution on encountering repeated arguments at run time. Section 4.2 contains an alternative that gives programmers control over

what is done when arguments are repeated. None of the techniques considered resorts to copying of general values. In the interest of streamlining the presentation, some discussion of related work is deferred to Section 5. Section 6 has our conclusions.

2. The Problem in Detail

Attempts to mitigate the complexities in reasoning from aliasing include applicative and functional programming languages, which do not support imperative features such as assignment and mutable objects, as well as imperative language features such as exchange statements and swapping [20]. Linear type systems [4][59] that prohibit aliases by using alternative data transfer techniques and language features to control aliasing [25][40][44] including many type systems (e.g., [6][9][47][55][60]) are among other research efforts with similar goals. Unfortunately, even if aliasing from assignments is eliminated completely by disallowing explicit reference assignments and using alternative data movement operators such as swapping or destructive reads, aliasing problem repeated arguments remains. Implicit repeated arguments resulting from array accesses cannot, in general, be detected with complete precision at compile time. Implicit repeated arguments resulting from passing global variables can only be detected by using a whole-program analysis or by annotating procedures with information about what global they access (as in Euclid [19]).

While the reasoning problem from repeated arguments is similar in spirit to the general aliasing problem, eliminating references and aliases by any method proposed to date still leaves the repeated argument problem as a separate issue. Focusing on this single source of aliasing, in addition to making it possible to discuss in depth, helps highlight the general reasoning difficulties from any source of aliasing. This is useful to motivate the related work on alias avoidance and control, and make the notion of clean semantics a more pervasive general language design notion.

2.1. Clean Semantics and Software Engineering

To capture the essence of what makes it easier to reason about some calls over others in software development, we introduce an intuitive, yet syntactic notion of clean semantics. A semantics *S* for a programming language *L* is clean if it has the following two properties: It is *variable-based*: *S* presents the state of a program as just the aggregate abstract values of all the variables defined at any particular point in that program. It is *effect restricted*: For each invocation of each operation *P*, the state change prescribed by *S* does not modify the values of any variables that are not syntactically germane to the invocation.

The variable-based property is simply the standard notion of state space in semantics. To define the notion of *syntactically germane*, we say that in an operation invocation, explicit parameters to the call as well as any global variables in scope are syntactically germane to the call. For example, suppose that `s` and `t` are two stacks of graphs (that happen to have the same values) as shown in Figure 1. The call `s.push(x)` does not have clean semantics if `s` and `t` are aliases, because it modifies `t` which is not syntactically germane to the call[1]. Stacks `s` and `t` may be aliases before the

[1] Calls with array indexed variables (e.g., `P(a[i])`) have clean semantics because both the array and the index are germane to the call. Here, we can take as the value of an array its mapping from indices to the values it contains; that is,

call to push either because the call is initiated from a repeated argument context as detailed in section 2.2 or because of an explicit assignment before the call. Regardless of its source, if aliasing were present, then the clean view of stacks in Figure 1 would lead to unsound reasoning.

Intentionally, we have avoided references and aliasing in explaining the idea of clean semantics. This is because we acknowledge that there are situations where a software engineer might *choose* to conceptualize a variable as a location (an abstraction of reference). Such a situation arises at a minimum when one attempts to specify, for example, a component to capture reference behavior [32]. Sharing is another reason why such a conceptualization might be necessary. However, by allowing objects to be viewed as references, the idea of clean semantics makes it also possible to view all complex objects —such as stacks, queues, and graphs— as references. In particular, if stacks s and t are viewed as (some abstraction of) references into a global store and the global store is introduced as syntactically germane to procedure calls, then the call s.push(x) has clean semantics, because it only affects the global store and not the reference t. This is illustrated in Figure 2.

Languages that allow aliasing among mutable objects from some source (even those where the only source is calls with repeated arguments) *require* a view of sophisticated objects as references into a global store for clean semantics and sound reasoning. While we do not want to preclude such a view, it is important to supplement the notion of clean semantics with the additional goal of allowing objects to be viewed as taking values from a richer semantic space (as illustrated in Figure 1 for stacks of graphs). Formally, the benefits of this idea can be seen by comparing the specifications of stacks in Figure 8 and in Appendix A, and by comparing the proofs of correctness in Appendices A and B. Using data abstraction specifications, we have shown elsewhere that it is possible to capture the behavior of objects such as lists [54] and graphs [52][53], using a rich space of mathematical models such as strings and sets of edges, respectively, instead of using references. They can be implemented using references (with or without cycles) correctly by establishing appropriate correspondences between implementations and specifications, thus limiting the need to use references to a few implementations [32][33].

---

such a value tells us everything we need to know about the array and completely determines the results of indexing expressions. In this sense, an array's value is completely encapsulated and independent of other objects.

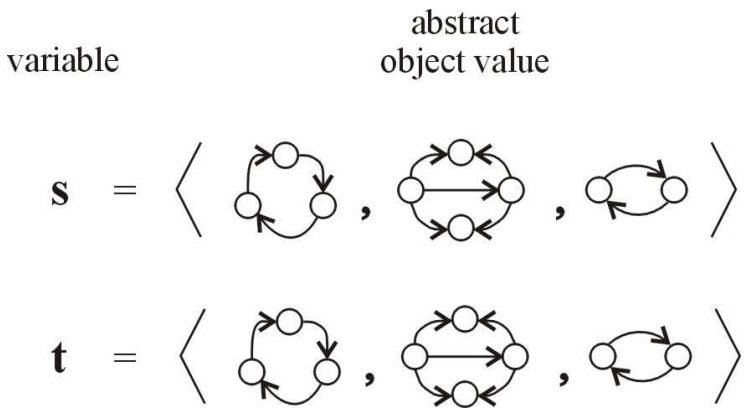

Figure 1. A state space for Stack objects

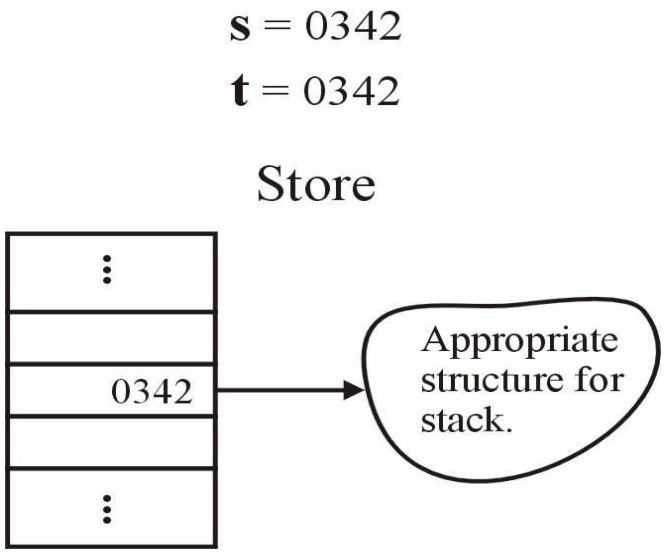

Figure 2. A less desirable state space for Stack objects

2.2. Understanding the Repeated Argument Problem

There are several aspects to the repeated argument problem that are explained in this section. Figure 3 contains an informal specification of an external operation, named transferTop that manipulates Stack objects. We use Java syntax for the programming aspects and a variation of the RESOLVE notation for formal specification [51][54], though others can be used [63]. The requires clause gives the precondition and the ensures clause gives its postcondition. In the postcondition #s denotes the old value of s (i.e., the value of s in the pre-state). A correct implementation should satisfy the postcondition if the precondition is satisfied.

```
public void transferTop(Stack s, Stack t);
    requires  (* s has at least one entry *);
    ensures (* s is unchanged except that it has lost its top entry, and
        t is unchanged except that it has acquired the top entry of #s as its new top entry *);
```

Figure 3. An informal specification of transferTop operation

Figure 4 contains an apparently correct and straightforward implementation of the specification. While this code could have been written as t.push(s.pop()) without using a local variable, we have found it easier to illustrate the distinction between verification of operations that return results and those that do not, using this expanded version.

```
public void transferTop(Stack s, Stack t) {
    Object x;
    x = s.pop();
    t.push(x);
 }
```

Figure 4. An implementation of the transferTop operation

A difficulty arises in interpreting the specification on a call with repeated arguments, such as in transferTop(u, u). On call transferTop(u, u), by substituting, the actual arguments for the formals, the postcondition in Figure 3 becomes contradictory (because u cannot have both lost an entry and acquired an entry) and the implementation in Figure 4 (that leaves u unchanged) is not correct. The more general problem is that the specification and implementations become inconsistent when arguments are repeated.

The Clean semantics problem from aliasing

The first problem is one of clean semantics. In particular, when there is aliasing, reasoning using the simple views of stacks in Figure 1 is unsound. To see why, consider a call such as s.pop() or t.push(x) that occurs in the body of the call transferTop(u, u). These calls violate the effect-restricted property of clean semantics, because s.pop() affects t that is not syntactically germane to the call. The clean semantics problem for the calls to s.pop() or t.push(x) remain even if transferTop is never called with repeated arguments in a calling program, because in modular reasoning, it is essential to show that the code in Figure 4 satisfies the specification in Figure 3 under all conditions. This reasoning needs to account for the fact that the formal parameters might be aliased. What is worse, modular reasoning about any procedure that uses a global variable — even one that has a single parameter— has the same problem, because that procedure might be called with that particular global variable as the argument.

The semantic space problem

While there is a direct problem for clean semantics within the body of a call with repeated arguments, there is no such problem in the calling code itself. This is because only u is germane in the call transferTop(u, u). However, capturing the different behavior of the code on calls with repeated arguments requires a semantic view of all objects as locations in the specification, as explained in this section. In the specification in Figure 5, unlike in typical Hoare logic, variable names stand for locations which themselves are abstractions of references.

```
public void transferTop(Stack s, Stack t);
    requires (* Store(s) must have at least one entry *);
    ensures   (* Store changes at most at locations s and t and
               if s = t then Store(s) is unchanged  and
               if s ≠ t then
                   Store(s) is unchanged except that it has lost its top entry, and
                   Store(t) is unchanged except that it has acquired the top entry of
                   Store(#s) as its new top entry *);
```

Figure 5. A specification of transferTop accounting for aliasing from repeated arguments

The notion of variables as locations and the introduction of a global store complicate reasoning about the code and the calls to transferTop. What is worse, this complexity cannot be limited to procedures such as transferTop with potential for repeated arguments, because the notion of stacks (and other variables) as locations must be made in any (mathematical) modeling of stack objects. Specifications and reasoning about calls to push and pop are rendered equally complex. Corresponding formal specifications of Loc_Based_Stack_Template and transferTop operation, and proofs of correctness of transferTop code and to the call transferTop(u, u) are given in Appendix A. The example shows that once a language allows repeated arguments and passes parameters by reference, in reasoning about *any* procedure call in that language it is necessary to view variables as mere locations – even if the language precludes all explicit aliasing.

Is it possible to prohibit repeated arguments through specification? For example, consider adding the conjunction s ≠ t to the requires clause of transferTop. Unfortunately, this solution does not solve the reasoning complexity problem, because the view of variables as locations is still necessary in reasoning. Alternatively, consider the specification in Figure 6, where stacks are abstract objects as in Figure 1 and there is no mention of a global store. This specification tries to prevent repeated arguments because it requires that arguments have distinct values. However, this precondition is too strong and has an unintended effect. When the procedure is called with two *distinct* stacks u and v that happen to have the same entries, the specification prohibits the call.

```
public void transferTop(Stack s, Stack t);
    requires (* s ≠ t and s has at least one entry *);
    ensures (* s is unchanged except that it has lost its top entry, and
            t is unchanged except that it has acquired the top entry of #s as its new top entry *);
```

Figure 6. An improper specification to prevent repeated arguments to transferTop

Additional issues and examples

To keep the discussion focused, we illustrate the issues in this paper using example calls such as transferTop(u, u) and transferTop(a[i], a[j]). However, similar issues arise when a global variable is passed as an argument as in the call p(g) when g is already in the scope of p, and when an array (or a record) and an element (or a field) are passed as arguments as in the call p(a, a[i]). A solution to the repeated argument problem must handle these situations as well. The issues raised in this paper are independent of the types of objects, though we use Stack examples throughout this paper. Similar problems arise, for example, with repeated argument calls to binary or tertiary operations

on other objects, such as a graph operation maxSubset operation that takes two 0-1 graphs with the same number of vertices and returns the intersecting graph in one of the two parameters. For another example, a tupleSum operation is discussed in subsection 3.5.

We conclude this section noting that the behavior of code when arguments are aliased can have far-reaching consequences. For example, data encapsulation may be compromised because internal representations invariants are violated. Unlike the implementation in Figure 4, the implementation of transferTop shown in Figure 7 is part of the Stack class and it has direct access to the internal "linked node" representation of stacks. The code is supposed to preserve an invariant that the linked nodes are acyclic, and it respects this invariant when arguments are not repeated. However, the call u.transferTop(u) leads to a Stack object u that is circular. Once a data abstraction is compromised in this way, the consequences become unpredictable.

```
public void transferTop(Stack t) {
   Node temp = this.node; this.node = this.node.next;  temp.next = t.node;  t.node = temp;
}
```

Figure 7. An implementation of the transferTop procedure with Representation Access

3. A Solution to the Problem

In this section we examine a solution to the repeated argument problem that uses initial variable semantics for second and subsequent arguments that are repeated. The *initialization semantics* of a function or procedure call such as p(u, u, u) is the same as that of T temp1, temp2; p(u, temp1, temp2), where T is the type of u and T temp1, temp2 denotes declaration and initialization of variables of type T. If p is a function, it returns the result of f(u, temp1, temp2). Under the initialization semantics, the call transferTop(u, u) has the effect "u is unchanged except that it has lost its top entry" as specified in the ensures clause in Figure 3, if "u has at least one entry" when the operation is called as specified in the requires clause. Design and implementation of a variation of Java based on the initialization semantics for parameter passing is discussed in [33]. The rest of the section formalizes the initialization semantics. It explains an efficient implementation that requires a modification to parameter passing by copying references. We begin with a formal specification of Stack_Template.

3.1. Specification of Stack_Template and Operations

The specification in Figure 8 formalizes the view of stacks in Figure 1 and presents stacks as mathematical strings of objects. Here, Λ denotes the empty string and o denotes string concatenation.

```
/* Stack_Template */
class Stack;
    uses String_Theory;
    type Stack is modeled by Str(Object);

public Stack();
    ensures this = Λ;

public void push(Object x);
    updates this, x;
    ensures this = ⟨#x⟩ ∘ #this;

public void pop(Object x);
    updates this, x;
    requires this ≠ Λ;
    ensures #this = ⟨x⟩ ∘ this;
```

Figure 8. Specification of a Stack component

A formal specification of transferTop is given in Figure 9. On the call transferTop(u, u), the ensures clause is interpreted as though the call were T temp; transferTop(u, temp).

```
uses Stack_Template;
public void transferTop(Stack s, Stack t);
    requires |s| > 0;
    ensures s^R ∘ t = #s^R ∘ #t and |s| = |#s| - 1;
```

Figure 9. A specification of the transferTop operation

3.2. Formal Reasoning

One way to present the initialization semantics is to distinguish calls with repeated arguments from others. Presented this way, the semantics of the call P(a, a) is given as T temp; P(a, temp) followed by another rule where the parameters are known to be distinct as in P(a, b). We will also need different rules for handling calls with array indexed variables and global variables. While different sets of rules are unavoidable for the other solutions considered in this paper, it is possible to give initialization semantics using a single rule. To see how this can be done, suppose that v <– w denotes an "initializing transfer operator" and that it gives the value of w to v, but makes v's value an initial value for the type of v. Using this operator, the semantics of a call of the general form P(a, b) can be given as T t1, t2; t1 <– a; t2 <– b; P(t1, t2); b <– t2; a <– t1. When a and b are distinct, it is easy to see that this sequence has the desired effect. On call P(a, a), the semantics is equivalent to T t1, t2; t1 <– a; t2 <– a; P(t1, t2); a <– t2; a <– t1 and has the net effect of the call T t2; P(a, t2) because after the first transfer a gets the initial value and it is transferred to t2. After the call, the value of the first parameter becomes the value of a because of the reverse order of transfer on return: a <– t2; a <– t1. Given this idea, we discuss a single proof rule for procedure call that gives initialization semantics.

Figure 10 contains a proof rule to describe the semantics of a procedure call. Without loss of generality, we consider a call with only two arguments. The *Context* for this rule must include the specification of the called procedure, *p*, as shown in the first hypothesis. The notation *assertive_code* means the statements and assertions (including assumptions) that precede the call to *p*. The rule shows what needs to be proved before a call to *p* for an assertion *Q* to be confirmed after the call. The rule introduces two local verification variables, specially named $\%a_x$ and $\%b_y$ to which the values of the corresponding actual arguments are transferred. The names *a* and *b* have been subscripted with names of the formal parameters *x* and *y* to avoid naming conflicts in the repeated argument case *p*(*a*,*a*) when an automated verification system is used.

The rule requires two conjuncts to be proved. The first of these is the precondition of *p*, which needs to be proved when the formals, *x* and *y*, are replaced by the actuals, *a* and *b*. Also, assuming the postcondition of *p* holds, the assertion *Q* needs to be confirmed (with proper substitutions). Since the specification of *p* may be relational, multiple post-state values may result for the same input values. So the second conjunct states that as long as the post-state values of the arguments satisfy the post condition, *Q* must be confirmed. Verification variables are prefixed with *?* to denote possible post-state values. The formal output names $?a_x$ and $?b_y$ replace actual arguments *a* and *b*, before *Q* is confirmed so that the names used in the two sides of the implication are consistent; that the value of the first formal parameter is used when arguments are repeated is reflected indirectly by the ordering in the substitutions in *Q*. When arguments are repeated and *Q* only involves *a*, for example, notice that only the output value of the first parameter $?a_x$ will affect *Q*, not $?b_y$. In the absence of repetition, the order of result return has no impact on the resulting state.

(void *p*(*T1 x*, *T2 y*); requires *pre*; ensures *post*;) ∈ *Context*,
Context \ *assertive_code*; *T1* $\%a_x$; *T2* $\%b_y$; $\%a_x$ <– *a*; $\%b_y$ <– *b*;
confirm *pre* [*x* ← $\%a_x$, *y* ← $\%b_y$] **and**
(∀$?a_x$: *T1*, ∀$?b_y$: *T2*, *post*[#*x* ← $\%a_x$, *x* ←$?a_x$, #*y* ← $\%b_y$, *y* ←$?b_y$] ⇒ *Q'* [*a* ←$?a_x$][*b* ←$?b_y$]);

Context \ *assertive_code*; *p*(*a*, *b*); **confirm** *Q*;
where *Q'* = *Q* with substitutions for $\%a_x$, $\%b_y$, $?a_x$, and $?b_y$ to avoid name conflicts.

Figure 10. A general proof rule for verification of procedure calls

The semantics of the initializing transfer operator is given in Figure 11. Here *T*. **is_initial**(*x*) is a predicate that tells if its argument *x* (which should be of type *T*) has an initial value. Since we assume it is possible to initialize variables of any type using the declaration T temp, every type must have a (public) constructor with no arguments – a practice that is widely followed in component design. The predicate *T*. **is_initial** for a particular type *T* is the postcondition of *T*'s constructor operation. For example, the predicate Stack.**is_initial**(s) is to true if s is a stack variable with value Λ (see the ensures clause of the constructor operation in Figure 8). We use a predicate instead of asserting s = Stack.**is_initial** because a constructor operation may provide one of multiple possible initial values. If the ensures clause of the constructor is omitted for a type *T*, the predicate *T*.**is_initial**(*x*) becomes trivially true, because an initial object of type *T* is allowed to have any abstract value. While this may not be desirable in most cases, it does not cause any difficulties for our formal system.

Context \ *assertive_code*;
confirm (∀initial_T: *T*, *T*.**is_initial**(initial_T) ⇒ *Q'* [b ← initial_T][*a* ← *b*]);

Context \ *assertive_code*; *a* <– *b*; **confirm** *Q*;
where *Q'*= *Q* with substitution for initial_T to avoid name conflicts.

Figure 11. Semantics of the initializing transfer operator

To more fully illustrate the use of the rules, we have included a proof of correctness for the implementation of transferTop from Figure 4 in Appendix B. In that appendix we also verify a client program that calls transferTop with repeated arguments. The use of the procedure call rule for the common calling situation in which no arguments are repeated is illustrated in verification of calls to the push operation and pop function in the transferTop code. The use of the same call rule when arguments are repeated is seen in the proof of the client program. It is instructive to compare the complexity of specification and reasoning in Appendix A over that in Appendix B based on clean semantics.

We conclude this section with an example. Consider the code, **for** (i = j; i <= (j+k)/2; i++) { p(a[i], a[k – i]) }, where j and k are within array bounds. For a particular instance, assume that a is an array of stacks of graphs and p is the transferTop operation. The code leads to a repeated argument situation if (k-j) is even. Under the initialization semantics for handling repeated arguments, a programmer can infer correctly the contents of a after the call – in particular, that when (k-j) is even a[(k-j)/2] is "unchanged except that it has lost its top entry" based on the specification in Figure 3. If this is the overall behavior the client desired then there is nothing more to do. If a different value is required at that array index, then a client can write specialized code to modify that value. All different ways of handling repeated arguments analyzed in this paper require such specialized reasoning. They are better than current imperative language practice of allowing repeated arguments and passing parameters by reference, because they support clean semantics.

3.3. Efficient Implementation

We discuss one efficient way for a compiler to realize initialization semantics using a modification of parameter passing by reference. In several situations (assuming parameter aliasing is the only source of aliasing), a compiler can detect (i.e., statically) that a call does not involve repeated arguments and it can generate code for passing parameters by copying references as usual. Examples include calls such as p(u) where u is not a global variable in the scope of p and calls of the form p(u, v). When repeated arguments are detected statically as in calls p(u, u) or p(a, a[i]) it generates T temp; p(u, temp) or T temp; p(a, temp), respectively. Only when repeated arguments cannot be detected statically, it uses the following implementation. After copying the reference of an actual argument to the parameter stack, the compiler generates code to replace the actual reference with a special, invalid address. For example, if a[i] is the argument, the compiler generates code to copy the reference at a[i] to the parameter stack and replace a[i] with a special, invalid address. Then it generates code to check for the special address in transferring subsequent arguments. If the checking succeeds – this presents a repeated argument situation, and the generated code creates an initial value of the appropriate type as the parameter. On return, parameters are transferred back (in reverse order) from the parameter stack. In this

implementation, initialization overhead is involved only for repeated arguments. A compiler may further defer initialization until an object is actually accessed, avoiding the initialization cost if a repeated argument is not used. This optimization is valid for well-engineered software where it is possible to declare variables of every type without side-effects. It is not possible to the extent a particular software design or a language design violates this principle.

3.4. Warnings and Debugging Provisions

Given the implementation outlined in the previous section, we expect that a compiler would generate two kinds of warnings: when a call involves repeated arguments for sure and when there is repeated argument potential. In addition to the static warnings, we expect that a compiler will have a development/debugging code generation option under which it generates code to produce run-time warnings when repeated argument calls are encountered. This is easy to instrument in the above implementation because the warning is generated when special address checking discussed above succeeds. Of course, code for deployment would not include code to generate warnings.

3.5. The Optional Role of Value Copying

The repeated argument problem cannot be solved by copying only because it cannot be implemented efficiently for non-trivial objects and cannot be mechanized correctly, in general [18]. However, copying causes no problem for clean semantics. Therefore, a language design should allow a programmer to direct invocation of replication operation optionally. This option is easy to provide for primitive types such as Integers, where copying is both cheap and available automatically. For other types, this option should be made available, if the programmer has supplied an implementation of a specially-named function, say, "replica". But the language semantics for a call such as p(a, a) should be defined when p is specified so that the compiler can interpret it properly as p(a, a) – a true repeated argument situation, or as p(a.replica(), a.replica()). We propose an expression *evaluation* mode for this purpose. For example, suppose that an integer addition function that takes two integers and returns the sum is defined as int addInt(eval i, eval j). Then the call addInt(i, i) is seen as a shorthand by the compiler and it substitutes the expression i.replica() in the call with the result addInt(i.replica(), i.replica()). For another example, consider the specification of a tupleSum operation given in Java-like syntax in Figure 12.

```
public Tuple tupleSum (eval Tuple x, eval Tuple y);
    ensures tupleSum = x ⊕ y;
```

Figure 12. A specification of Tuple summation operation

If the programmer has defined replica for Tuple, then the call tupleSum(t, t) is interpreted as tupleSum(t.replica(), t.replica()). Otherwise, the call has a type error and is rejected by the compiler. However, if tupleSum is specified without the evaluates mode, regardless of whether a replica operation has been defined, the call tupleSum(t, t) is given initialization semantics. When some of the parameters of the procedure *p* are declared to be in expression evaluation mode, the initialization semantics is amended to evaluate the expressions before processing other arguments. For example, regardless of whether tupleSum operation is declared to be tupleSum(**eval** Tuple x, Tuple y) or tupleSum(Tuple x, **eval** Tuple y), the result of tupleSum(t, t) is t ⊕ t. This is because for

both calls tupleSum(t.replica(), t) and tupleSum(t, t.replica()) the replica function is evaluated before t is passed.

4. Alternative Design Considerations

4.1. Precluding and Aborting on Calls with Repeated Arguments

A language designer may use a *disciplined approach* by prohibiting repeated arguments. In this approach, parameters can be passed by copying references, yet clean semantics will be preserved for calls because no observable aliasing can result. We distinguish and describe two versions of the disciplined approach: a rigid version and a relaxed version.

In the *rigid* version of the disciplined approach, the language *syntactically* precludes calls with repeated arguments from arising. In other words, if a call has explicit or implicit repeated arguments, a compiler error will result. This is the approach proposed in Harms and Weide [20], where, citing Cook's observations on repeated arguments, they *syntactically* disallow calls such as p(u,u), calls involving indexed array expressions such as p(a[i],a[k]), and calls involving record field access or global variables. This proposal can be improved by noticing calls such as p(x.first, y.first) can be allowed. Similarly, if the global variables that a procedure can access are declared statically (as in Euclid [41]), then it is possible to allow them to be passed as arguments and have a compiler check violations.

The main difficulty with the rigid disciplined approach is that it must be conservative. It must prohibit calls such as p(a[exp1], a[exp2]), because the equivalence of the expressions cannot be checked statically. The code segment considered in the previous section, **for** (i = j; i <= (j+k)/2; i++) { p(a[i], a[k – i]) }, needs to be rewritten explicitly using explicit temporary variables, as **for** (i = j; i <= (j+k)/2; i++) { T ai = a[i]; T ak = a[k-i]; p(ai, ak); a[k-i] := ak; a[i] := ai }. This is undesirable.

Unlike the rigid version, the *relaxed* version of the disciplined approach precludes repeated arguments *dynamically*. For example, in Euclid [41], an attempt to pass repeated arguments by reference is caught at runtime and results in the program aborting execution. Linear type systems that read parameters destructively can be viewed as following the relaxed disciplined approach as well, because the same parameter cannot be read twice [59][60]. The advantage of the relaxed version over the rigid version is that more procedure calls are allowed in the relaxed version – all calls that do not lead to the repeated argument problem at run time. For example, the call p(a[i],a[k]) would be legal syntactically and allowed dynamically as long as i does not equal k. Reasoning about total correctness of such code thus involves reasoning about the values of i and k.

Clean semantics of the relaxed disciplined approach [41] is case-based and it can be given using different patterns of repetition. While a naïve compiler would emit code that always checks for repeated arguments on every call, a more efficient implementation could follow the implementation strategy discussed in 3.3. On encountering a special address, the compiler will generate code to abort execution. One advantage for the initialization semantics approach that continues execution is when a programmer wants to repeat arguments and take advantage of their effects (as suggested in the array of stack example in the last section). Alternatively, suppose that a programmer repeats arguments by mistake. Also, suppose that a compiler instrumented to give warnings and equipped

with a debugging provision as discussed in 3.4 is used for the initialization approach. There is little difference between the relaxed disciplined and initialization approaches in terms of error revealing potential. If argument repetition is detectable statically, then the compiler will warn about the possibility as well as help reveal further syntactic errors. If argument repetition happens at runtime and happens during software development and debugging, under the initialization approach code generated with the debugging flag would produce a runtime warning *and* reveal any subsequent errors that may not be related to the call with repeated arguments. The code for the relaxed disciplined approach would have to abort, though it could be instrumented to give a warning before aborting.

A more significant difference arises when arguments are repeated dynamically during deployment. In this case, a solution that allows the program to continue might also allow it to recover, whereas the abortion semantics requires the program to crash. For example, suppose that following a call with repeated arguments, a program decides to take one of many actions depending on the value of one of the variables affected by the call. If the variable has an unexpected value (because an earlier call with was an error), then the program might just follow one of the action paths that might contain an error recovery solution. This is important for systems where aborting has catastrophic consequences. For these reasons, it is more desirable to solve the clean semantics problem without necessarily aborting on calls with repeated arguments.

4.2. The Multi-pattern Semantics

The multi-pattern semantics [3][36] allows one to write multiple specifications and implementations for the same operation so that each handles a distinct pattern of repetition among the procedure's actual parameters. It can be thought of as a variant of the relaxed disciplined approach [41]; as in that approach, the global variables that a procedure can access are declared as part of its signature.

In the multi-pattern approach, a specification has multiple "specification cases", separated by an **also** keyword, each of which corresponds to a particular pattern of repeated arguments. By default, the first specification case must be satisfied by the procedure when it is called without repeated arguments. Specification cases that start with a **repeated** declaration, such as the second specification case in Figure 13, must be satisfied by the procedure when called with the pattern of repeated arguments specified. In a specification case with repeated arguments, the specifier is only allowed to use the first repeated argument (in each group).

```
uses Stack_Template;
public void transferTop(Stack s, Stack t);
    requires |s| > 0;
    ensures s^R o t = #s^R o #t and |s| = |#s| - 1;
  also repeated(s,t)
    requires |s| > 0;
    ensures s = #s;
```

Figure 13. An extended form of specification, suitable for the multi-pattern approach

An implementation of the above specification is given in Figure 14. An implementation of a procedure has multiple bodies, each of which handles a distinct pattern of repeated arguments. The first body again handles the case where no arguments are repeated. And again, when implementing a specification for a repeated argument case, only the first repeated argument (in each group) may be used. A call to the procedure is dispatched to one of the multiple bodies in the procedure depending on which arguments are repeated in the call.

```
public void transferTop(Stack s, Stack t) {
    // multibody that is run when there are no repeated arguments
    Object x;  x = s.pop(); t.push(x);
} repeated(s,t) {
    // multibody that is run when t is a repetition of s; t cannot be used in this body
    skip;
}
```

Figure 14. Handling of repeated arguments using multiple bodies

Reasoning about calls in the multi-pattern semantics is case-based, in general. One splits the proof that a call achieves some effect into one case for each specified pattern. The caller must find, for each specification case, a predicate that implies the specified pattern of repeated arguments, and show that their disjunction is satisfied in the call's pre-state. Then these predicates are used as the assumptions in each case. Of course, if one can statically prove that only one pattern is possible at a call site, this reduces the proof to considering the corresponding specification case. But in general, all must be considered, especially for calls involving indexed array variables. For example, in a call such as transferTop(a[i], a[k]), the caller would use $i \neq k$ and $i = k$ as the predicates that would imply that the arguments are not repeated or are repeated, and the proof would have one case using each of these as a hypothesis. If it were known at the call site that one of these was false, then only the other case would need to be considered.

To implement the semantics, when a procedure is called, if a repeated argument situation is not detectable statically, as in general is the case for a call with array indexed arguments, a compiler needs to generate code to check if an actual pattern of repeated arguments matches one of the specified patterns. This can be made efficient for the case where there are no repeated arguments by using the idea described in Section 3.3. When no arguments are repeated, the default (first) multibody is called. Otherwise, a decision tree is used to find the corresponding multibody which may take time O(n log n) in the worst case [3]. A call with a pattern of repeated arguments that

does not correspond to a specified pattern is illegal, and leads to an abort at runtime if it cannot be detected statically.

The multi-pattern semantics requires that specifiers and programmers think about the behavior for each potential pattern of repeated arguments, and a multi-dispatching compiler. However, it is clean and it provides flexible control. Unlike the implicit semantics of the initialization approach which is fixed for a call, the multi-pattern approach allows it to be tailored. This flexibility also introduces opportunities for optimization. For example, if an application needs no action as the semantics for the call transferTop(u, u), then this is achieved by the multi-pattern example given in Figure 13; if the application wishes to delete the top element of u in that case, that can also be done; or a global variable could be consulted to determine which of these actions to take. Also, the multi-pattern semantics does not require or rely on the ability to initialize objects, thus it can work when variables of a type cannot be created. Though it needs a search to find the corresponding body in the general case, Assaad and Leavens note that this is not an efficiency problem in practice [3], in part because most procedures take only a few arguments, and in part because a compiler can use the extra information about aliasing to generate more highly optimized code.

5. Related Work

Previous treatments of parameter passing generally fall into one of two categories: those that prohibit repeated arguments, such as the procedure call rules introduced by Hoare [22], Cook [10], and Ernst [14]; and those that introduce references to handle repeated arguments, such as the procedure call rules given by Cartwright [8] and Gries [17]. Crank and Felleisen compare formal semantics for alternative parameter passing techniques, including parameter passing by reference and value [12]. Their conclusion is that "using call-by-value [...] seems the most attractive choice" (p.10) from a reasoning perspective; this is consistent with the clean semantics presented in this paper. Gries gives a rule for value-result parameters in [17] that handles repeated arguments by specifying the order of variable substitution. However the solutions we presented do not depend on value copying.

Though the focus of this paper is on achieving clean semantics avoiding aliasing from calls with repeated arguments, it is clear that a language designer should address all sources of aliasing. Hogg *et al.* surveyed the problem of object aliasing in [26]. It summarizes various alternatives to reference assignment (and deep copying) that do not introduce aliasing, including copying [2][30], destructive read [4][7][44][59], and swapping [20][30]. Destructive read has been used in conjunction with linear type systems [59]. Though it avoids aliasing, it requires programmers to distinguish between typical object values and distinguished null values. They correspond to the issues with the semantics we have introduced to describe the no op approach in this paper. Tools that can statically detect and trace null-pointer exceptions, such as ESC/Java [39], could potentially be leveraged to make languages with destructive read operators practical. Swapping avoids aliasing by exchanging conceptual object values, but can be implemented as a constant time operation by exchanging pointers [20]. Swapping is a symmetric operation, so it cannot be used in traditional object oriented systems to assign an object of one type to a variable associated with its supertype. A language may define a clearing transfer operator for that purpose [33][62].

Alias control can serve as an intermediate step towards achieving the eventual goal of clean semantics, though it cannot guarantee clean semantics in all situations. Hogg's *islands* [25] and Almeida's *balloon types* [2] are examples of approaches that aim to encapsulate aliasing by prohibiting external references into an alias-protected object. In their paper on flexible alias protection, Noble, *et al.*, suggested that there was a better way to provide improved alias encapsulation without limiting the programmer [47]. Other alias control techniques include ownership types by Clarke, which extend the ideas of flexible alias protection [9], and confined types by Bokowski, which are motivated by security concerns [6]. Numerous techniques for pointer analysis have been proposed. Though the computational difficulty of alias detection is high [29][35], some techniques have been shown to be fast and scalable [21]. These techniques can aid reasoning by helping to avoid extra case analysis when doing verification [34][50].

We have acknowledged that clean semantics does not preclude use of references which are unavoidable in some situations. Earlier work on specification and reasoning about object-oriented programs [1][13][37][38][45] is important. Some, like JML [37] and Object-Z [13], use reference semantics to capture object identity, but reasoning in the face of potential aliasing remains a problem, and Smith [56] and Müller [45] both suggest that some form of alias control is necessary to make reasoning tractable. Abadi and Leino introduce a logic to handle object-oriented programs with reference variables in [1], but admit that the rules are necessarily more complex than Hoare's "in part, because of aliasing." While reference variables need to be viewed as locations in a global store, the impact of procedure calls can be confined by frame properties [5][45], which limit the locations that can be potentially updated. O'Hearn [49] discusses a form of local reasoning by partitioning the object store.

6. Conclusions

We have explained and resolved the problem of providing clean semantics for procedure calls, including those with repeated arguments. We have shown that this goal can be accomplished efficiently without depending on copying. Though this can be achieved by precluding repeated arguments statically or by aborting execution on encountering repeated arguments at run time, we have shown these extreme measures are not necessary. The multi-pattern semantics requires that specifiers and programmers consider the behavior for each potential pattern of repeated arguments to provide maximum flexibility. The initialization semantics is implicit and simple.

While we have focused on the aliasing that stems from calls with repeated arguments in this paper, all sources of aliasing among mutable objects must be addressed eventually. Our vision for the future is a reformed programming paradigm in which languages have clean and rich semantics, making it easier to specify, develop, and maintain software. By providing clean semantics for calls with repeated arguments we have taken an important first step toward that vision.

Acknowledgments

This research is funded in part by the National Science Foundation under grant CCR-0113181. We would like to thank all members of our research groups at Clemson University, Iowa State

University, and Ohio State University for the discussions concerning the topics of this paper. Our specials thanks are due to Bruce Weide for his comments throughout the development of this paper.

References

[1] M. Abadi and K.R.M. Leino, "A Logic of Object-Oriented Programs," M. Bidoit and M. Dauchet (eds.), TAPSOFT '97: Theory and Practice of Software Development, 7th International Joint Conference, 1997, pp. 682-696.

[2] P.S. Almeida, "Balloon Types: Controlling Sharing of State in Data Types," Proceedings ECOOP '97, number 1241 in Lecture Notes in Computer Science, 1997, pp. 32-59.

[3] M. Assaad and G.T. Leavens, Alias-free parameters in C for Better Reasoning and Optimization, tech. report TR #01-11, Dept. of Computer Science, Iowa State Univ., Ames, IA, Nov. 2001. ftp://ftp.cs.iastate.edu/pub/techreports/TR01-11/TR.pdf

[4] H.G. Baker, "Lively Linear Lisp --- 'Look {Ma}, No Garbage!'," ACM SIGPLAN Notices, vol. 27, no. 8, Aug., 1991, pp. 89-98.

[5] A. Borgida, J. Mylopoulos, and R. Reiter, "On the Frame Problem in Procedure Specifications", IEEE Transactions on Software Engineering, vol. 21, no. 10, Oct. 1995, pp. 785-798.

[6] B. Bokowski and J. Vitek, "Confined Types," Proceedings 14th Annual ACM SIGPLAN Conference (OOPSLA '99), Nov., 1999, pp.82-96.

[7] J. Boyland, "Alias burying: Unique variables without destructive reads," Software—Practice and Experience, vol. 31, no. 6, May 2001, pp. 533-553.

[8] R. Cartwright and D. Oppen, "Unrestricted Procedure Calls in Hoare's Logic," Proceedings of the 5th ACM SIGACT-SIGPLAN Symposium on Principles of Programming Languages, 1978, pp. 131-140.

[9] D.G. Clarke, J.M. Potter, and J. Noble, "Ownership Types for Flexible Alias Protection," OOPSLA '98 Conf. Proc., ACM Press, 1998, pp. 48-64.

[10] S.A. Cook, "Soundness and Completeness of an Axiom System for Program Verification," SIAM Journal of Computing, vol. 7, no. 1, 1978, pp. 70-90.

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

[12] E. Crank and M. Felleisen, "Parameter passing and the lambda calculus," *Proc. 18th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL)*, ACM Press, Jan. 1991.

[13] R. Duke, G. Rose, and G. Smith, Object-Z: A Specification Language Advocated for the Description of Standards, Tech. report 94-45, SVRC University of Queensland, 1994.

[14] G.W. Ernst, "Rules of Inference for Procedure Calls," Acta Informatica, vol. 8, 1997, pp. 145-152.

[15] G.W. Ernst, R.J. Hookway, and W.F. Ogden, "Modular Verification of Data Abstractions with Shared Realizations," IEEE Transactions on Software Engineering, vol. 20, no. 4, 1994, pp. 288-207.

[16] D.P. Friedman, M. Wand, and C.T. Haynes, Essentials of Programming Languages, McGraw-Hill, New York, 1992.

[17] D. Gries and G. Levin, "Assignment and Procedure Call Proof Rules," ACM Transactions on Programming Languages and Systems, vol 2, no. 4, 1980, pp. 564-579.

[18] P. Grogono and M. Sakkinen, "Copying and Comparing: Problems and Solutions," E. Bertino (ed.), ECOOP 2000, LNCS 1850, 2000, pp. 226-250.

[19] J.V. Guttag, J.J. Horning, and R.L. London, "A Proof Rule for Euclid Procedures," E.J. Neuhold (ed.), IFIP TC-2 Working Conference on Formal Description of Programming Concepts, 1977.

[20] D.E. Harms and B.W. Weide, "Copying and Swapping: Influences on the Design of Reusable Software Components," IEEE Transactions on Software Engineering, vol. 17, no. 5, May 1991, pp. 424-435.

[21] N. Heintze and O. Tardieu, "Ultra-fast Aliasing Analysis using CLA: A Million Lines of C Code in a Second," PLDI01, 2001.

[22] C.A.R. Hoare, "Procedures and Parameters: An Axiomatic Approach," E. Engeler (ed.), Proceedings Symposium on Semantics of Algorithmic Languages, 1971, pp. 102-116.

[23] C.A.R. Hoare, Hints on Programming Language Design, tech. report CS-73-403, Computer Science Dept., Stanford University, 1973. Reprinted in Programming Languages: A Grand Tour, E. Horowitz, ed., Computer Science Press, Rockville, MD, 1983, pp. 31-40.

[24] C.A.R. Hoare, "Recursive Data Structures," International Journal of Computer and Information Science, vol. 4, no. 2, 1975, pp. 105-132.

[25] J. Hogg, Islands: "Aliasing Protection in Object-Oriented Languages," Proceedings OOPSLA '91, volume 26 of ACM SIGPLAN Notices, 1991, pp. 271-285.

[26] J. Hogg, D. Lea, A. Wills, D. deChampeaux, and R. Holt, "The Geneva Convention On The Treatment of Object Aliasing," OOPS Messenger vol. 3, no. 2, Apr. 1992, pp. 11-16. http://gee.cs.oswego.edu/dl/aliasing/aliasing.html

[27] J.E. Hollingsworth, L. Blankenship, and B.W. Weide, "Experience Report: Using RESOLVE/C++ for Commercial Software," Proc. ACM SIGSOFT 8th International Symp. on the Foundations of Software Engineering (FSE), ACM Press, 2000, pp. 11-19.

[28] J. Horning, "A Case Study in Language Design," F.L. Bauer and M. Broy (eds.), Program Construction, LNCS 69, Springer-Verlag, New York, 1979.

[29] S. Horowitz, "Precise Flow-Insensitive May-Alias Analysis is NP-hard," ACM Transactions on Programming Languages and Systems, vol. 19, no. 1, Jan. 1997, pp. 1-6.

[30] R.B. Kieburtz, "Programming Without Pointer Variables," In Proc SIGPLAN '76 Conf. on Data: Abstraction, Definition and Structure, 1976. ACM Press.

[31] A. Koenig and B. E. Moo, "Rethinking How to Teach C++," Journal of Object-Oriented Programming, Feb 2001, pp. 25-27.

[32] G. Kulczycki, M. Sitaraman, W. F. Ogden, and J. E. Hollingsworth, Component Technology for Pointers: Why and How, Technical Report RSRG-03-03, Department of Computer Science, Clemson University, Clemson, SC 29634-0974, April 2003, 19 pages.

[33] G. Kulczycki, Direct Reasoning and Efficiency through Language and Software Design, Ph. D. Dissertation, Department of Computer Science, Clemson University, Clemson, SC 29634-0974, December 2003.

[34] V. Kuncak, P. Lam, and M. Rinard, "Role Analysis," POPL02, 2002.

[35] W. Landi, "Undecidability of Static Analysis," ACM Lett. Program. Lang. Syst., vol. 1, no. 4, Dec. 1992, pp. 323-337.

[36] G.T. Leavens and O. Antropova,. ACL -- Eliminating Parameter Aliasing with Dynamic Dispatch, tech. report TR #98-08a, Dept. of Computer Science, Iowa State Univ., Ames, IA, 1998.

[37] G.T. Leavens, A.L. Baker, and C. Ruby, "JML: A Notation for Detailed Design," Behavioral Specifications of Businesses and Systems, H. Kilov and B. Rumpe and I. Simmonds, eds., Kluwer Academic Publishers, Boston, 1999.

[38] K.R.M. Leino and G. Nelson, Data Abstraction and Information Hiding, tech. Report 160, Compaq Systems Research Center, Palo Alto, CA, 2000.

[39] K.R.M. Leino, G. Nelson, and J.B. Saxe, "ESC/Java User's Manual," Technical Note 2000-002, Compaq Systems Research Center, October 2000.

[40] K.R.M Leino, A. Poetzsch-Heffter, and Y. Zhou, "Using Data Groups to Specify and Check Side Effects," PLDI02, 2002.

[41] R.L. London, J.V. Guttag, J.J. Horning, B.W. Lampson, J.G. Mitchell, and G.J. Popek, "Proof Rules for the Programming Language Euclid," Acta Informatica, vol. 10, no. 1, 1978, pp. 1-26.

[42] D.C. Luckham and N. Suzuki, "Verification of Array, Record, and Pointer Operations in Pascal," ACM Transactions on Programming Languages and Systems, vol. 1, no. 2, Oct. 1979, pp. 226-244.

[43] B. Meyer, Object-oriented Software Construction, 2nd ed., Prentice-Hall, New York, 1997.

[44] N.H. Minsky, "Towards Alias-Free Pointers," ECOOP '96 -- Object-Oriented Programming: 10th European Conf., Linz Austria, Springer-Verlag, New York, 1996, LNCS vol. 1098, pp. 189-209.

[45] P. Müller, Modular Specification and Verification of Object-Oriented Programs, Springer-Verlag, 2002, vol 2262 Lecture Notes in Computer Science.

[46] D.R. Musser, G.J. Derge, and A Saini, STL Tutorial and Reference Guide, 2nd ed., Addison-Wesley, 2001.

[47] J. Noble, J. Vitek and J. Potter, Flexible Alias Protection. ECOOP '98 -- Object-Oriented Programming, 12th European Conference, Brussels, Belgium, E. Jul, ed., volume 1445 of Lecture Notes in Computer Science, 1998, pp. 158-185.

[48] W.F. Ogden, The Proper Conceptualization of Data Structures, The Ohio State University, 2000.

[49] P. O'Hearn, J. Reynolds, and H. Yang, "Local Reasoning about Programs that Alter Data Structures," CSL01, 2001.

[50] N. Rinetzky and M. Sagiv, "Interprocedural Shape Analysis for Recursive Programs," CC01, 2001.

[51] M. Sitaraman, M. and B. Weide, eds., Special Feature: Component-Based Software Using RESOLVE, ACM SIGSOFT Software Engineering Notes, vol. 19, no. 4, Oct. 1994, pp. 21-67.

[52] M. Sitaraman, B. W. Weide, and W. F. Ogden, "Using Abstraction Relations to Verify Abstract Data Type Representations," IEEE Transactions on Software Engineering, March 1997, 157-170.

[53] M. Sitaraman, B. W. Weide, T. J. Long, and W. F. Ogden, "A Data Abstraction Alternative to Data Structure/Algorithm Modularization," *LNCS Volume on Generic Programming*, Eds. D. Musser and M. Jazayeri, Springer-Verlag, 2000, 102-113.

[54] M. Sitaraman, S. Atkinson, G. Kulczycki, B. W. Weide, T. J. Long, P. Bucci, W. Heym, S. Pike, and J. Hollingsworth, "Reasoning about Software-Component Behavior," Proceedings ICSR-6, Springer Verlag, 2000, LNCS vol. 1844, pp. 266-283.

[55] F. Smith, D. Walker, and G. Morrisett, "Alias Types," ESOP00, Berlin, March 2000.

[56] G. Smith, "Reasoning about Object-Z Specifications," Proceedings of Asia-Pacific Software Engineering Conference, 1995, pp. 489-497.

[57] R. D. Tennent, "Language Design Methods Based on Semantic Principles," Acta Informatica, vol 8, 1977, pp. 97-112.

[58] M. Utting, "Reasoning about Aliasing," *Formal Aspects of Computing*, vol 3, 1997, pp. 1-15.

[59] P. Wadler, "Linear Types Can Change the World!," IFIP TC 2 Working Conference on Programming Concepts and Methods, 1990.

[60] D. Walker, and G. Morrisett, "Alias Types for Recursive Data Structures," Workshop on Types in Compilation, 2000.

[61] B.W. Weide and W.D. Heym, "Specification and Verification with References," Proc. OOPSLA 2001 SAVCBS Workshop, tech. rep. #01-09a, Dept. of Comp. Sci., Iowa State Univ., 2001, pp. 50-59. http://www.cs.iastate.edu/~leavens/SAVCBS/papers-2001/weide-heym.pdf

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

[63] J.M. Wing, "A Specifier's Introduction to Formal Methods," IEEE Computer, vol. 23, no. 9, Sept. 1990, pp. 8-24.

Appendix A

This section presents a formal, reference-based stack specification and as well as a formal specification for the transferTop operation. It gives proof rules for reference copying procedure call and function assignment. Finally, it gives example proofs using these specifications and rules. The formal specification in Figure 1 is given in the RESOLVE notation [51][54] for a Stack type. Here, the mathematical type Location (informally an address) denotes an arbitrary mathematical set, which we assume is defined in an external memory manager facility, Memory_Manager_Fac. The only thing of interest here about Location is its cardinality, which is presumably a function of available memory capacity. Given this background, a Stack variable can be modeled mathematically as a Location. Every new stack variable occupies a new location. The Store variable models the portion of the global store that maps locations to conceptual stack objects, which are modeled as mathematical strings of entries.

```
/* Loc_Based_Stack_Template */
class Stack;
      uses String_Theory, Memory_Manager_Fac;
      var Store: Location → Str(Object);
      type Stack is modeled by Location;

   public Stack();
      updates Store;
      ensures Store(this) = Λ and  (∀r: Stack if Stack.num(#r) ≠ Stack.num(#this) then
                  r ≠ this and Store(r) = #Store(r));

   public void push(Object x);[2]
      updates Store;
      ensures this = #this and Store(this) = ⟨#x⟩ ∘ #Store(this) and
                  (∀r: Stack, if r ≠ this then Store(r) = #Store(r));

   public Object pop();
      updates Store;
      requires Store(this) ≠ Λ;
      ensures this = #this and #Store(this) = ⟨pop()⟩ ∘ Store(this) and
                  (∀r: Stack, if r ≠ this then Store(r) = #Store(r));
```

Figure 1. A specification of a Stack component that accounts for references

Based on this modeling, the rest of the specification describes Stack's public operations. The ensures clause for the constructor contains a type function, **num**, which returns the serial number for the specified stack. Every stack object gets a unique serial number when it is constructed. This mechanism provides a way to indicate object identity without assuming how it is implemented. Thus, in the condition Stack.**num**(#r) ≠ Stack.**num**(#this), r is any stack except the one being constructed. The ensures clause for the constructor guarantees that Store(this) is the empty string, denoted by Λ, that the location of the new stack is unique, and that the contents of all other stack variables remain unaffected. The operations push and pop change Store at location this. In the

[2] For simplicity, we assume x cannot be aliased to a Stack object.

specifications "o" denotes string concatenation. The specification asserts explicitly that when the contents of "this" is changed, then the contents of all other stack variables remain unaffected. The syntactically germane variables for the push operation include the actuals that correspond the formal parameters this and x, and the global Store variable (which is explicitly listed in the **updates** clause). All other defined variables remain unchanged. The formal, reference-based specification for the transferTop operation is given in Figure 2.

public void transferTop(Stack s, Stack t);
updates Store;
requires |Store(s)| > 0;
ensures s = #s **and** t = #t **and**
(**if** s ≠ t **then** (Store(s)^R o Store(t) = #Store(s)^R o #Store(t) **and**
|Store(s)| = |#Store(s)| - 1)) **and**
(**if** s = t **then** Store(s) = #Store(s)) **and**
(∀r: Stack **if** r ≠ s **and** r ≠ t **then** Store(r) = #Store(r));

Figure 2. A reference-based specification of transferTop

The procedure call proof rules for reference copying are straightforward and do not involve variable declaration or initialization. The complexity of proofs involving reference copying comes from the need to distinguish references from object values in the specification. We will need two rules for the proofs given later in this appendix. A procedure call rule and a function call rule. Both rules account for an updated mathematical global variable G:

Context \ assertive_code;
confirm pre [x ← %a_x, y ← %b_y] **and**
∀?a_x: T1, ∀?b_y: T2, ∀?G: T3, post[#x ← a_x, x ←?a_x, #y ← b_y, y ←?b_y, #G ← G, G ← ?G]
⇒ Q'[a ←?a_x][b ←?b_y][G ←?G];
Context U { **void** p(T1 x, T2 y) **updates** G } \ assertive_code; p(a, b); **confirm** Q;

where Q'= Q with substitutions for ?a_x, ?b_y, and ?G to avoid name conflicts.

Figure 3. A reference copying proof rule that includes a mathematical global

Context \ assertive_code;
confirm pre [x ← %a_x] **and**
∀?a_x: T1, ∀?b_f: T2, ∀?G: T3, post[#x ← a_x, x ←?a_x, f() ←?b_f, #G ←G, G ←?G]
⇒ Q'[b ←?b_f][a ←?a_x][G ←?G];
Context U { T2 f(T1 x) **updates** G } \ assertive_code; b = f(a); **confirm** Q;

where Q'= Q with substitutions for ?a_x, ?b_f, and ?G to avoid name conflicts, and where f() in the post condition of function f denotes its returned value.

Figure 4. A reference copying proof rule for function calls

The remaining part of this appendix uses the reference copying approach to parameter passing to demonstrate the following proofs:

- **Proof #1** A proof of client code with repeated arguments using the specification of transferTop given in Figure 2.
- **Proof #2** A proof of correctness for transferTop using the specification in Figure 2, and the referenced based specification for Stack given in Figure 1.

Proof #1: A proof of client code with repeated arguments using the specification of transferTop given in Figure 2. We want to prove the following assertive program:

{ **void** transferTop(Stack s, Stack t) **updates** Store } \
assume Store(u) = ⟨a⟩;
transferTop(u, u);
confirm Store(u) = ⟨a⟩;

By applying the parameter passing proof rule in Figure 3, this reduces to:

assume Store(u) = ⟨a⟩;
confirm |Store(u)| > 0 **and**
((∀?u_s: Stack, ∀?u_t: Stack, ∀?Store: Location → Str(Object),
u = ?u_s **and** u = ?u_t **and**
(**if** ?u_s ≠ ?u_t **then** (?Store(?u_s)^R ∘ ?Store(?u_t) = Store(?u_s)^R ∘ Store(?u_t) **and**
|?Store(?u_s)| = |Store(?u_s)| - 1) **and**
(**if** ?u_s = ?u_t **then** ?Store(?u_s) = Store(?u_s) **and**
(∀r: Stack, **if** r ≠ ?u_s **and** r ≠ ?u_t **then** ?Store(r) = Store(r))) **implies**
?Store(?u_s) = ⟨a⟩);

Reducing the assume-confirm sequence to an implication yields:

(Store(u) = ⟨a⟩) ⇒ (|Store(u)| > 0 **and**
((∀?u_s: Stack, ∀?u_t: Stack, ∀?Store: Location → Str(Object),
u = ?u_s **and** u = ?u_t **and**
(**if** ?u_s ≠ ?u_t **then** ?Store(?u_s)^R ∘ ?Store(?u_t) = Store(?u_s)^R ∘ Store(?u_t) **and**
|?Store(?u_s)| = |Store(?u_s)| - 1) **and**
(**if** ?u_s = ?u_t **then** ?Store(?u_s) = Store(?u_s)) **and**
(∀r: Stack, **if** r ≠ ?u_s **and** r ≠ ?u_t **then** ?Store(r) = Store(r))) **implies**
?Store(?u_s) = ⟨a⟩));

Given that ∀?u_s: Stack, u = ?u_s and ∀?u_t: Stack, u = ?u_t, this reduces to:

(Store(u) = ⟨a⟩) ⇒ (|Store(u)| > 0 **and**
∀?Store: Location → Str(Object),
(**if** u ≠ u **then** ?Store(u)^R ∘ ?Store(u) = Store(u)^R ∘ Store(u) **and**
|?Store(u)| = |Store(u)| - 1) **and**
(**if** u = u **then** ?Store(u) = Store(u)) **and**
(∀r: Stack, **if** r ≠ u **and** r ≠ u **then** ?Store(r) = Store(r))) **implies**
?Store(u) = ⟨a⟩);

Since u = u, we can eliminate the conjunct that describes what happens when u ≠ u.

(Store(u) = ⟨a⟩) ⇒ (|Store(u)| > 0 **and**
∀?Store: Location → Str(Object),
?Store(u) = Store(u) **and**
∀r: Stack, **if** r ≠ u **and** r ≠ u **then** ?Store(r) = Store(r) **implies**
?Store(u) = ⟨a⟩);

We now see that ∀?Store: Location → Str(Object), ?Store(u) = Store(u), so we can further reduce this to:

(Store(u) = ⟨a⟩) ⇒ (|Store(u)| > 0 **and**
∀r: Stack, **if** r ≠ u **and** r ≠ u **then** ?Store(r) = Store(r) **implies**
?Store(u) = ⟨a⟩);

When Store(u) = ⟨a⟩ this becomes:

|⟨a⟩| > 0 **and** ∀r: Stack, **if** r ≠ u **and** r ≠ u **then** Store(r) = Store(r) **implies** ⟨a⟩ = ⟨a⟩;

Which it true because |⟨a⟩| = 1 and because ⟨a⟩ = ⟨a⟩ is always true.

Proof #2: A proof of correctness for transferTop using the specification in Figure 2, and the referenced based specification for Stack given in Figure 1. We need to prove:

```
{ Object pop() updates Store } U { void push(Object x) updates Store } \
void transferTop(Stack s, Stack t);
    updates Store;
    requires |Store(s)| > 0;
    ensures s = #s and t = #t and
        (if s ≠ t then (Store(s)^R o Store(t) = #Store(s)^R o #Store(t) and
                |Store(s)| = |#Store(s)| - 1)) and
        (if s = t then Store(s) = #Store(s)) and
        (∀r: Stack if r ≠ s and r ≠ t then Store(r) = #Store(r));
{
    Object x;
    x = s.pop();
    t.push(x);
}
```

By a proof rule for procedure body declaration this reduces to:

```
{ Object pop() updates Store } U { void push(Object x) updates Store } \
assume |Store(s)| > 0;
Object x;
x = s.pop();
t.push(x);
confirm s = #s and t = #t and
    (if s ≠ t then (Store(s)^R ∘ Store(t) = #Store(s)^R ∘ #Store(t) and
        |Store(s)| = |#Store(s)| - 1)) and
    (if s = t then Store(s) = #Store(s)) and
    ∀r: Stack, if r ≠ s and r ≠ t then Store(r) = # Store(r);
```

By the proof rule for procedure call (Figure 3) for push, this reduces to:

```
{ Object pop() updates Store } \
assume |Store(s)| > 0;
Object x;
x = s.pop();
confirm ∀?t_s: Stack, ∀?x_x: Object, ∀?Store: Location → Str(Object),
    (?t_s = t and ?Store(?t_s) = ⟨x⟩ ∘ Store(?t_s) and
    ∀r: Stack, if r ≠ ?t_s then ?Store(r) = Store(r)) implies
    (s = #s and ?t_s = #t and
    (if s ≠ ?t_s then (?Store(s)^R ∘ ?Store(?t_s) = #Store(s)^R ∘ #Store(?t_s) and
        |?Store(s)| = |#Store(s)| - 1)) and
    (if s = ?t_s then ?Store(s) = #Store(s)) and
    (∀r: Stack, if r ≠ s and r ≠ ?t_s then ?Store(r) = #Store(r)));
```

Using the fact that $\forall ?t_s$: Stack, $?t_s = t$, and the fact that $\forall ?x_x$: Object, $?x_x$ is never used, we can reduce this to:

```
{ Object pop() updates Store } \
assume |Store(s)| > 0;
Object x;
x = s.pop();
confirm ∀?Store: Location → Str(Object),
    (?Store(?t_s) = ⟨x⟩ ∘ Store(?t_s) and
    ∀r: Stack, if r ≠ t then ?Store(r) = Store(r)) implies
    (s = #s and ?t_s = #t and
    (if s ≠ t then (?Store(s)^R ∘ ?Store(t) = #Store(s)^R ∘ #Store(t) and
        |?Store(s)| = |#Store(s)| - 1)) and
    (if s = t then ?Store(s) = #Store(s)) and
    (∀r: Stack, if r ≠ s and r ≠ ?t_s then ?Store(r) = #Store(r)));
```

Next, we need to apply the function call rule for pop. Before doing this, we rename ?Store as ??Store to avoid naming conflicts.

```
{ Object pop() updates Store } \
assume |Store(s)| > 0;
Object x;
x = s.pop();
confirm ∀??Store: Location → Str(Object),
    (??Store(t) = ⟨x⟩ ∘ Store(t) and
    ∀r: Stack, if r ≠ t then ??Store(r) = Store(r)) implies
    (s = #s and t = #t and
```

(**if** s ≠ t **then** (??Store(s)^R ∘ ??Store(t) = #Store(s)^R ∘ #Store(t) **and**
|??Store(s)| = |#Store(s)| - 1)) **and**
(**if** s = t **then** ??Store(s) = #Store(s)) **and**
(∀r: Stack, **if** r ≠ s **and** r ≠ t **then** ??Store(r)= #Store(r)));

We now apply the function call rule (Figure 4) for pop, which reduces the assertive code to:

assume |Store(s)| > 0;
Object x;
confirm Store(s) ≠ Λ **and** (∀?s_s: Stack, ∀?x_{pop}: Object, ∀?Store: Location → Str(Object),
?s_s = s **and** #Store(?s_s) = ⟨?x_{pop}⟩ ∘ ?Store(?s_s) **and**
(∀r: Stack, **if** r ≠ ?s_s **then** ?Store(r) = #Store(r))) **implies**
(∀??Store: Location → Str(Object),
(??Store(t) = ⟨?x_{pop}⟩ ∘ ?Store(t) **and**
∀r: Stack, **if** r ≠ t **then** ??Store(r) = ?Store(r)) **implies**
(?s_s = #s **and** t = #t **and**
(**if** ?s_s ≠ t **then** (??Store(?s_s)^R ∘ ??Store(t) = #Store(?s_s)^R ∘ #Store(t) **and**
|??Store(?s_s)| = |#Store(?s_s)| - 1)) **and**
(**if** ?s_s = t **then** ??Store(?s_s) = #Store(?s_s)) **and**
(∀r: Stack, **if** r ≠ ?s_s **and** r ≠ t **then** ??Store(r) = #Store(r)));

Using the fact that ∀?s_s: Stack, ?s_s = s and eliminating the declaration of "Object x" (which has no effect on the confirm clause), we can rewrite the expression as:

assume |Store(s)| > 0;
confirm Store(s) ≠ Λ **and** (∀?x_{pop}: Object, ∀?Store: Location → Str(Object),
#Store(s) = ⟨?x_{pop}⟩ ∘ ?Store(s) **and**
(∀r: Stack, **if** r ≠ s **then** ?Store(r) = #Store(r))) **implies**
(∀??Store: Location → Str(Object),
(??Store(t) = ⟨?x_{pop}⟩ ∘ ?Store(t) **and**
∀r: Stack, **if** r ≠ t **then** ??Store(r) = ?Store(r)) **implies**
(s = #s **and** t = #t **and**
(**if** s ≠ t **then** (??Store(s)^R ∘ ??Store(t) = #Store(s)^R ∘ #Store(t) **and**
|??Store(s)| = |#Store(s)| - 1)) **and**
(**if** s = t **then** ??Store(s) = #Store(s)) **and**
(∀r: Stack, **if** r ≠ s **and** r ≠ t **then** ??Store(r) = #Store(r)));

Reducing the assume-confirm sequence to an implication and eliminating the # symbols yields:

|Store(s)| > 0 ⇒ Store(s) ≠ Λ **and** (∀?x_{pop}: Object, ∀?Store: Location → Str(Object),
Store(s) = ⟨?x_{pop}⟩ ∘ ?Store(s) **and**
(∀r: Stack, **if** r ≠ s **then** ?Store(r) = Store(r))) **implies**
(∀??Store: Location → Str(Object),
(??Store(t) = ⟨?x_{pop}⟩ ∘ ?Store(t) **and**
∀r: Stack, **if** r ≠ t **then** ??Store(r) = ?Store(r)) **implies**
(s = s **and** t = t **and**

(**if** s ≠ t **then** (??Store(s)^R ∘ ??Store(t) = Store(s)^R ∘ Store(t) **and**
    |??Store(s)| = |Store(s)| - 1)) **and**
(**if** s = t **then** ??Store(s) = Store(s)) **and**
    (∀r: Stack, **if** r ≠ s **and** r ≠ t **then** ??Store(r) = Store(r)));

This complex expression has the form:

∀s, t: Stack, ∀?x_{pop}: Object, ∀?Store, ??Store: Location → Str(Object),
|Store(s)| > 0 ⇒ Store(s) ≠ Λ **and**
(*first_antecedent* **implies** (*second_antecedent* **implies** *consequent*));

Therefore, it suffices to prove ∀s, t: Stack, ∀?x_{pop}: Object, and ∀?Store, ??Store: Location → Str(Object), that:

1. |Store(s)| > 0 ⇒ Store(s) ≠ Λ
2. |Store(s)| > 0 **and** *first_antecedent* **and** *second_antecedent* ⇒ *consequent*

The first assertion is clearly true. The consequent in the second assertion can be broken down into four separate assertions:

1. s = s **and** t = t
2. **if** s ≠ t **then** ??Store(s)^R ∘ ??Store(t) = Store(s)^R ∘ Store(t) **and** |??Store(s)| = |Store(s)| - 1
3. **if** s = t **then** ??Store(s) = Store(s)
4. ∀r: Stack, **if** r ≠ s **and** r ≠ t **then** ??Store(r) = Store(r)

Assertion (1) is obviously true. To deal with assertion (2), assume that s ≠ t. Using the string equalities from the *first_antecedent* and the *second_antecedent* we can simplify the string equality in assertion (2) as follows:

??Store(s)^R ∘ ??Store(t) = Store(s)^R ∘ Store(t)
    /* *apply string equality from second_antecedent* */
??Store(s)^R ∘ ⟨?x_{pop}⟩ ∘ ?Store(t) = Store(s)^R ∘ Store(t)
    /* *apply string equality from first_antecedent* */
??Store(s)^R ∘ ⟨?x_{pop}⟩ ∘ ?Store(t) = (⟨?x_{pop}⟩ ∘ ?Store(s))^R ∘ Store(t)
    /* *algebraic string manipulation yields the following* */
??Store(s)^R ∘ ⟨?x_{pop}⟩ ∘ ?Store(t) = ?Store(s)^R ∘ ⟨?x_{pop}⟩ ∘ Store(t)

The frame properties in the *second_antecedent* and the *first_antecedent* allow us to conclude that ??Store(s) = ?Store(s) and ?Store(t) = Store(t) respectively. Thus the string equality in assertion (2) is true. The numeric equality in assertion (2) also simplifies:

|??Store(s)| = |Store(s)| - 1
    /* *apply string equality from first_antecedent* */
|??Store(s)| = |⟨?x_{pop}⟩ ∘ ?Store(s)| - 1
    /* *algebraic string manipulation yields the following* */

|??Store(s)| = 1 + |?Store(s)| - 1

Using the frame property in the *first_antecedent* allows us to conclude that ??Store(s) = ?Store(s). Therefore assertion (2) is true. Assertion (3) works under the assumption that s = t. Substituting s for t in the string equality from the *second_antecedent* and combining it with the string equality from the *first_antecedent* will yield the string equality in assertion (3). Assertion (4) follows directly from the frame properties in the *first_antecedent* and the *second_antecedent*.

Appendix B

This appendix uses the initializing parameter passing approach to demonstrate the following proofs:

- **Proof #3** A proof of client code with repeated arguments using the specification of transferTop in Figure 9 of the main paper.
- **Proof #4** A proof of correctness for transferTop using the specification in Figure 9 of the main paper, and the straightforward specification for Stack given in Figure 8 of the main paper.

Proof #3: A proof of client code with repeated arguments using the specification of transferTop in Figure 9 of the main paper and the initializing approach to parameter passing. We want to prove the following assertive program:

{ **void** transferTop(Stack s, Stack t) } \
assume |u| = 1;
transferTop(u, u);
confirm u = Λ;

By the general proof rule for procedure call (Figure 10 in the main paper) this becomes:

assume |u| = 1;
Stack $\%u_s$; Stack $\%u_t$;
$\%u_s$ <– u; $\%u_t$ <– u;
confirm $|\%u_s| > 0$ **and**
∀?u_s: Stack, ∀?u_t: Stack,
if $?u_s^R \circ ?u_t = \%s^R \circ \%t$ **and** $|?u_s| = |\%u_s| - 1$ **then** $?u_s = \Lambda$;

By the proof rule for the initializing transfer operator (Figure 11 in the main paper) we can replace all instances of $\%u_t$ in the confirm statement with u, and simultaneously replace all instances of u in the confirm statement (if there were any) with Λ. This reduces our assertive code to:

Stack $\%u_s$; Stack $\%u_t$;
$\%u_s$ <– u;
confirm $|\%u_s| > 0$ **and**
∀?u_s: Stack, ∀?u_t :Stack,
if $?u_s^R \circ ?u_t = \%u_s^R \circ u$ and $|?u_s| = |\%u_s| - 1$ **then** $?u_s = \Lambda$;

By a second application of the proof rule for initializing transfer this becomes:

assume |u| = 1;
Stack $\%u_s$; Stack $\%u_t$;
confirm |u| > 0 **and**
    $\forall ?u_s$: Stack, $\forall ?u_t$:Stack,
    if $?u_s^R \circ ?u_t = u^R \circ \Lambda$ **and** $|?u_s| = |u| - 1$ **then** $?u_s = \Lambda$;

Two applications of a proof rule for variable declarations allow us to replace instances of the declared variables that appear in the confirm statement (if there were any) with their initial values, reducing the assertive code to:

assume |u| = 1;
confirm |u| > 0 **and**
    $\forall ?u_s$: Stack, $\forall ?u_t$:Stack,
    if $(?u_s^R \circ ?u_t = u^R \circ \Lambda$ **and** $|?u_s| = |u| - 1)$ **then** $?u_s = \Lambda$;

Once the assertive code is reduced to an assume statement followed by a confirm statement, we can replace it by an implication in which the assume clause is the antecedent and the confirm clause is the consequent.

$(|u| = 1) \Rightarrow$ (|u| > 0 **and**
    $\forall ?u_s$: Stack, $\forall ?u_t$:Stack,
    if $?u_s^R \circ ?u_t = u^R \circ \Lambda$ **and** $|?u_s| = |u| - 1$ **then** $?u_s = \Lambda$);

When |u| = 1 it follows that |u| > 0. It also follows that $|?u_s| = |u| - 1$ reduces to $|?u_s| = 0$, which implies that $?u_s = \Lambda$. Therefore, the expression is true.

Proof #4: This proof of correctness for transferTop uses the straightforward stack specification given in Figure 8 of the main paper. We want to prove the assertive program:

```
{ void pop(Object x) } U { void push(Object x) }\
public void transferTop(Stack s, Stack t)
    requires |s| > 0;
    ensures s^R ∘ t = #s^R ∘ #t and |s| = |#s| - 1;
{
    Object x;
    s.pop(x);
    t.push(x);
}
```

By a proof rule for procedure body declaration this reduces to:

```
{ void pop(Object x) } U { void push(Object x) } \
assume |s| > 0;
Object x;
s.pop(x);
t.push(x);
confirm s^R ∘ t = #s^R ∘ #t and |s| = |#s| - 1;
```

By the proof rule for procedure call (Figure 10 in the paper) on push this reduces to:

{ **void** pop(Object x) } \
assume |s| > 0;
Object x;
s.pop(x);
Stack $\%t_s$; Object $\%x_x$;
$\%t_s$ <– t; $\%x_x$ <– x;
confirm $\forall ?t_s$: Stack, **if** $?t_s = \langle \%x_x \rangle \circ \%t_s$ **then** ($s^R \circ ?t_s = \#s^R \circ \#t$ **and** |s| = |#s| - 1);

By two applications of the proof rule for initializing transfer and two applications of a proof rule for variable declaration this reduces to:

{ **void** pop(Object x) } \
assume |s| > 0;
Object x;
s.pop(x);
confirm $\forall ?t_s$: Stack, **if** $?t_s = \langle x \rangle \circ t$ **then** ($s^R \circ ?t_s = \#s^R \circ \#t$ **and** |s| = |#s| - 1);

By the proof rule for procedure call on pop this reduces to:

assume |s| > 0;
Object x;
Stack $\%s_s$; Object $\%x_x$;
$\%s_s$ <– s; $\%x_x$ <– x;
confirm $|\%s_s| > 0$ **and**
    $\forall ?x_x$: Object, $\forall ?s_s$: Stack, **if** $\%s_s = \langle ?x_x \rangle \circ ?s_s$ **then**
    ($\forall ?t_s$: Stack, **if** $?t_s = \langle ?x_x \rangle \circ t$ **then** ($?s_s^R \circ ?t_s = \#s^R \circ \#t$ **and** $|?s_s| = |\#s| - 1$));

By two applications of the proof rule for initializing transfer and three applications of a proof rule for variable declaration this reduces to:

assume |s| > 0;
confirm |s| > 0 **and**
    $\forall ?x_x$: Object, $\forall ?s_s$: Stack, **if** $s = \langle ?x_x \rangle \circ ?s_s$ **then**
    ($\forall ?t_s$: Stack, **if** $?t_s = \langle ?x_x \rangle \circ t$ **then** ($?s_s^R \circ ?t_s = \#s^R \circ \#t$ **and** $|?s_s| = |\#s| - 1$));

Reducing the assume-confirm statement sequence to an implication and simultaneously eliminating the # symbols (since in the confirm clause, s is the same as #s and t is the same as #t) reduces this to:

$|s| > 0 \Rightarrow$ (|s| > 0 **and** $\forall ?x_x$: Object, $\forall ?s_s$: Stack, **if** $s = \langle ?x_x \rangle \circ ?s_s$ **then**
    ($\forall ?t_s$: Stack, **if** $?t_s = \langle ?x_x \rangle \circ t$ **then** ($?s_s^R \circ ?t_s = s^R \circ t$ **and** $|?s_s| = |s| - 1$)));

It suffices to show that each of the following three assertions are true for all $?x_x$: Object, $?s_s$: Stack, and $?t_s$: Stack:

1. $|s| > 0$ **implies** $|s| > 0$;
2. $|s| > 0$ **and** $s = \langle ?x_x \rangle \circ ?s_s$ **and** $?t_s = \langle ?x_x \rangle \circ t$ **implies** $?s_s^R \circ ?t_s = s^R \circ t$;
3. $|s| > 0$ **and** $s = \langle ?x_x \rangle \circ ?s_s$ **and** $?t_s = \langle ?x_x \rangle \circ t$ **implies** $|?s_s| = |s| - 1$;

The first assertions is trivially true. The third is true because $s = \langle ?x_x \rangle \circ ?s_s \Rightarrow s^R = ?s_s^R \circ \langle ?x_x \rangle$, so that the equation on the right of the implication becomes $?s_s^R \circ \langle ?x_x \rangle \circ t = ?s_s^R \circ \langle ?x_x \rangle \circ t$. The third is true because $s = \langle ?x_x \rangle \circ ?s_s \Rightarrow |s| = 1 + |?s_s|$.