

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

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

Technical Report RSRG-03-01¹
Department of Computer Science
451 Edwards Hall
Clemson University
Clemson, SC 29634-0974 USA

January 2003

Copyright © 2003 by the authors. All rights reserved.

¹ Also published as Technical Report 02-13, Department of Computer Science, Iowa State University, Ames, Iowa 50011-1041.

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

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, Bruce W. Weide
Computer and Information Science, The Ohio State University, Columbus, OH 43210-1277 USA
ogden@cis.ohio-state.edu, weide@cis.ohio-state.edu

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

Abstract

A fundamental complexity in human understanding and reasoning about imperative, object-based software systems has to do with the need to distinguish references and values of objects. It is possible to eliminate this complexity by (deep) copying values of all mutable objects, but this is too inefficient for typical, non-trivial objects. The problem of minimizing the impact of the reference-value distinction without resorting to value copying manifests itself when objects are repeated as parameters to procedures. From a software engineering perspective, we consider alternative strategies to address the repeated argument problem ranging from ones that disallow repeated arguments to more permissive ones; from ones that do not require any new programming language mechanisms to ones that need new features. We introduce a parameter passing approach that neither requires the reference-value distinction nor value copying to handle repeated arguments. We present a specification-aware, unrestricted, proof rule schema for procedure calls that is suitable for verification using alternative parameter passing techniques, separately or in combination.

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

1. Introduction

To facilitate efficient data movement, large objects in modern programming languages are usually referred to indirectly. In some languages, this indirection is introduced implicitly, as with references in Java, C#, Lisp, and Smalltalk. Such languages often feature an *indirect model* of storage, in which compound data structures (objects, records, arrays, etc.) are allocated on the heap and referred to indirectly through references. By contrast, in languages with a *direct model* of

storage, such as Pascal, Ada, C, and C++, such compound data structures are allocated on the stack and denoted directly by variable names.¹ In languages with a direct model of storage, indirection needs to be programmed explicitly, with pointers. We will refer to both kinds of explicit and implicit pointers as *references*. Both have the same operational effect on execution performance: it takes constant time to copy a reference, whereas copying an object's entire representation is generally more expensive. Furthermore, it is not possible to provide automatic deep copying (or deep comparison) that works appropriately for all arbitrary objects [18]. Hence we wish to avoid or minimize copying.

References, though seemingly necessary for efficient computing, complicate both understanding and reasoning about program behavior. As early as 1973, Hoare warned of references that “their introduction into high-level languages has been a step backward from which we may never recover” ([22], p.37). And in 1976, Kieburtz explained why we should be “programming without pointer variables” ([29], p.95). Reasoning concerns about references—formal and informal—typically focus on the aliasing they can cause. Indeed, it is not just actual aliasing, but the mere potential for aliasing that complicates specification and verification. In Cook's seminal paper on the soundness and completeness of Hoare logic he suggests that procedure call rules are the most difficult to make sound because the rules must account for (or carefully avoid) the aliasing caused by repeated arguments [10]. In their paper on object aliasing, Hogg et. al. point out that the possibility of aliasing can make it “annoyingly difficult to prove the simple Hoare formula $\{x = \text{true}\} y := \text{false} \{x = \text{true}\}$ ” ([25], p.11). The need to account for reference behavior complicates specifications as well [48][50]. Logics and formal specification languages with reference semantics can be used to make specifications appear less cluttered, but they cannot prevent the reasoning complexities caused by indirection and aliasing [1][34][47].

References also complicate reasoning for practicing programmers. This is partly why Koenig and Moo argue for rethinking how C++ is taught in [30]. Their introductory C++ textbook introduces vectors, strings, and structures long before even mentioning pointers, because “pointers are a slippery subject to master, and beginners have a much easier time dealing with the value-like classes” such as vectors, strings, and structs (p.27). The orthodox canonical form is a popular C++ idiom in which programmers override assignment and copy constructors to perform deep copies, allowing them to think of objects as values rather than references [11]. Value semantics is explained as a fundamental principle in the design of the C++ Standard Template Library in [42].

The complexity caused by references is especially problematic in languages with an indirect model of storage, where indirection and aliasing often make it essential to distinguish between references and values to ensure sound reasoning. To avoid the complexity of this reference-value distinction, practicing programmers and students routinely use a simplified clean view of storage. In the *clean view*, one just ignores indirection and pretends that reference variables directly denote the values (i.e., representations) of the objects to which they refer.

Figure 1. illustrates three different views of a variable s whose type is a bounded stack of trees. Figure 1(a) shows the actual representation of s as a reference to an elaborate structure. The view in

¹ This terminology comes from the first edition of the book *Essentials of Programming Languages* [16]. While C and C++ use the direct model, these languages take the value of an array to be a pointer to its first element when used in expression contexts, making them somewhat resemble the indirect model in the treatment of arrays.

Figure 1(b) is clean because it suppresses the fact that s is a reference to this representation data structure and that the contents array contains references. The view in Figure 1(b), however, is implementation-dependent, unlike the view in Figure 1(c) where s is viewed as a mathematical string (or sequence) of entries. The view in Figure 1(c) is both clean and abstract.

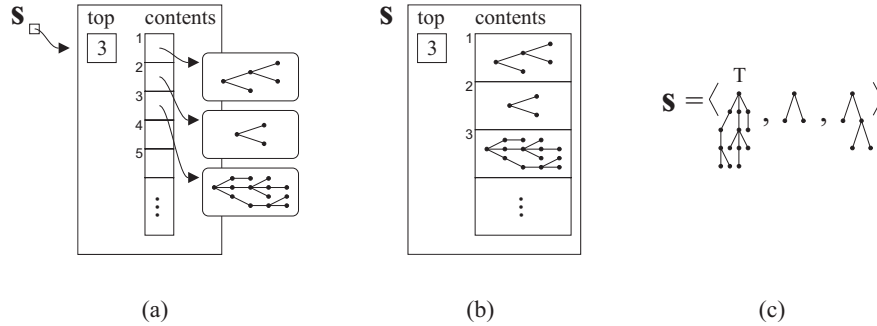


Figure 1. Three views of a bounded stack of trees.

1.1. Why Clean Steps Are Necessary for the Clean View to Be Sound

Although the clean view of objects as values (with or without abstraction) is desirable for programming and reasoning about programs, it is *unsound* when mutable objects are aliased. The clean view can be used as a basis for reasoning only when the steps of a program are *clean*, that is, when they do not introduce the need for references in reasoning. Steps that introduce aliasing necessarily introduce the need to distinguish references and values in reasoning, therefore they are not clean. For example, suppose that the reference variable s in Figure 1 is assigned to another variable t using reference assignment (denoted $t = s$ in Java). Immediately after the assignment, both variables point to the same representation, so the actual view and the clean view agree that s and t have the same value. Later, however, if the representation associated with s is modified, then the actual view and the clean view will *not* agree. In the clean view only the value of s changes, while the value of t remains the same. However, in reality there is just one shared representation that changes, leading to an inconsistency between the actual view shown in Figure 2(a) and the clean view shown in Figure 2(b).

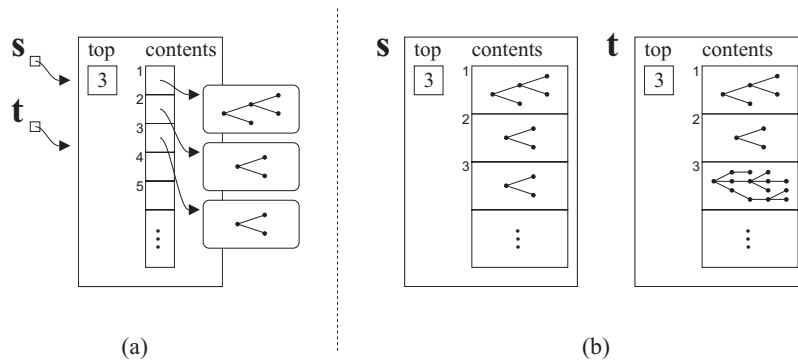


Figure 2. Two views of aliased bounded stacks of trees

Reference assignment is the most direct way to introduce aliasing, but it can also occur through *repeated storage of references* when a reference is passed as a parameter, as in the Java call `s.push(x)` where `s` is a stack object containing references to its entries. After the call, `x` will be aliased to the stack's top element. Likewise, aliasing can be introduced when an aggregate object returns a reference to one of its internal objects, as in the Java assignment `x = s.top()` where the method `top` does not remove the element from the stack.

Another common source of aliasing stems from repeated arguments. Repeated arguments occur when two (or more) references to the same object are passed in a single call to a procedure, which can lead to aliasing among the formal parameters of the called procedure. They may occur explicitly, as in the call `p(u, u)`, or implicitly, when two array elements are passed to a procedure, as in the call `p(a[i], a[j])`. In the later case, there may be no way to statically detect, in general, the aliasing that may result if $i = j$. The same kind of implicit passing of repeated arguments may happen when a user-defined collection is accessed in languages such as Java.

A subtler kind of implicit repeated argument is also possible. This is aliasing that occurs between actual arguments and global variables that are accessible within the called procedure. For example, in the call `f(x)`, if the procedure `f` also has direct access to the variable `x`, then the procedure's environment will have an alias between the global `x` and `f`'s formal parameter. We consider this to be a case of repeated arguments because we consider the global variables used by a procedure to be implicit arguments to the procedure [37]. We refer to the potential for contradiction and confusion in specifications and implementations, when arguments are repeated, as the *repeated argument problem*.

1.2. Contributions

The larger software engineering objective of our research is to address all sources of aliasing, minimize the need for the reference-value distinction, and minimize the complexity of reasoning in imperative programs. We illustrate how the reference-value distinction complicates software engineering activities such as specification and verification, using aliasing from repeated arguments

as an illustrative issue. To understand the novelty of our contributions, some background information on procedure call verification in the context of repeated arguments is useful.

Noting the difficulty with reasoning in the presence of aliasing, Hoare introduced parameter passing rules with the provision that “the programmer is willing to observe a certain familiar and natural *discipline* in his use of parameters” ([21], p.102). In his approach, programmers are responsible for ensuring that parameters are independent of one another. They may neither repeat arguments nor introduce aliasing, directly indirectly. Cook’s rules improve upon Hoare’s rules and allow global variables to be passed as parameters to procedures [10], but still avoid the aliasing caused by repeated arguments. While his rules allow global variables to be passed as parameters, there are restrictions to prevent argument repetition. Similarly, Ernst extended Hoare’s rules by permitting constructs involving limited dependence (such as $p(x, f(x))$ where f must be a function) and globals that were not updated within the procedure body [14]. Guttag, Horning, and London extended the logic of Euclid [37] with procedure call rules [19] which, as elsewhere in language, permitted pointers in a restricted way.

Cartwright was the first to introduce parameter passing rules for Hoare logic that permitted aliasing by making the reference-value distinction explicit [8]. In effect, variables passed as parameters to a procedure were considered abstract addresses that were kept track of in a global mapping of variables to values. While the rules themselves are not that complicated, the authors noted that they were “cumbersome to use in practice” (p.131) because pre- and postconditions had to account for all possible cases of potential aliasing between variables. Inspired by Cartwright’s approach, Gries explains parameter passing in terms of a simple multiple assignment rule in [17], and outlines how these rules could be extended to handle aliasing from argument repetition. Crank and Felleisen present and compare formal semantics for alternative parameter passing techniques, including parameter passing by reference and value [12]. Their conclusion that “using call-by-value [...] seems the most attractive choice” (p.10) from a reasoning perspective is consistent with the clean view presented in this paper. Using ease of value-based reasoning as motivation, Harms and Weide propose swapping an efficient alternative to value copying in [20], but they preclude repeated arguments.

Given this background, we make the following contributions in this paper:

- We explain various strategies to address the repeated argument problem ranging from ones that disallow repeated arguments to more permissive ones; from ones that do not require any new programming language mechanisms to ones that need new features. We discuss the merits and drawbacks from the perspectives of software engineering and languages.
- We present an approach for parameter passing—other than call-by-value—that allows repeated arguments, but does not introduce aliasing or require a reference-value distinction; the approach makes a procedure call with repeated arguments a clean step.
- We present a specification-aware, unrestricted, proof rule for procedure calls. An interesting aspect of this contribution is a single procedure call proof rule schema that works for all the parameter passing approaches we consider, and that is flexible enough to combine various parameter passing approaches.

To support a fairly standard practice of programming, we present the results in an imperative setting. We do not discuss non-imperative solutions such as functional programming, although we explain the utility of some ideas (e.g., linear typing), that are taken from a non-imperative context.

1.3. Where Aliasing Appears Unavoidable

While aliasing introduces complexity, an accurate view of objects with aliasing is necessary in reasoning about low-level modules that deal with manipulation of references for efficient processing of list or tree-like structures [29]. In these modules, aliasing is used intentionally to share storage, avoid copying, and allow multiple access paths. Thus in such modules adopting a clean view that ignores references would be detrimental and make reasoning more complex. On the other hand, most modules in a large program are based on reusing abstract objects, and in these higher-level modules the clean view offers significant software engineering benefits. To separate the modules where reasoning with the clean view is sound from those where it is not, data abstraction is crucial. In particular, one must abstract away the details of sharing and references found in the implementation of low-level modules by using abstract data types. For example, one can view a doubly linked list abstractly as a pair of strings or sequences [46][44], and in this abstract view there is no need to think about the sharing implicit in the representation.

Such abstract views of data can be guaranteed to be sound, using recent work on alias-controlling type systems [2][24][43][9][41]. This work centers on alias encapsulation, and allows control of all references into and out of the representation of an object. Thus this research can be used to enforce statically the separation between higher-level modules with a clean view and lower-level modules that do not support this view. Since the number of such low-level modules would presumably be small, the additional annotations required by these type systems would not be as much of a drawback as they would be if one tried to use them in all modules. In this paper we focus on the more common higher-level client modules, where a clean view is both desirable and possible. Alternatively, the reader can imagine two kinds of references: unique references used by high-level client modules, and non-unique references used in low-level modules. The focus of this paper is on the former. A discussion of the latter may be found in [31].

1.4. Organization of this Paper

The rest of the paper is organized into the following sections. Section 2 pertains to the first contribution and it considers alternative responses for handling repeated arguments. In the process it highlights the complexity introduced by the reference-value distinction. Section 3 corresponds to the second contribution. It discusses parameter passing approaches that avoid the reference-value distinction and related software engineering issues. Section 4 discusses proof rules for formal verification. Section 5 contains a discussion of related work and our conclusions.

2. Understanding the Impact of the Reference-Value Distinction

2.1. An Example Problem

To illustrate the impact of the reference-value distinction on software engineering, we consider an example that raises the repeated argument question. Figure 3 contains an informal specification of a (global) procedure, named `transferTop`, which manipulates `Stack` objects. In this specification, `#s` in the ensures clause (postcondition) denotes the old value of `s` (i.e., the value of `s` in the pre-state).

```
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. Informal specification of the `transferTop` procedure

Now consider the code in Figure 4 to implement the `transferTop` procedure².

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

Figure 4. An implementation of the `transferTop` procedure

Suppose that `transferTop` is called with stacks `u` and `v` as parameters in a clean environment. Also suppose that `u` is a stack of integers, which we write abstractly as the string $\langle \underline{7}, 6, 9 \rangle$, where the underlined entry (i.e., 7) is the top, and suppose that `v` is another stack with abstract value $\langle 8 \rangle$. Tracing through the call `transferTop(u,v)` using this sound and abstract view of stacks yields the table given in Figure 5 from which it appears that the code is correct for this case.

Pre-state	<code>u = $\langle \underline{7}, 6, 9 \rangle$ and $v = \langle 8 \rangle$</code>
	<code>Object x; x = u.pop();</code>
State after Pop	<code>u = $\langle \underline{6}, 9 \rangle$ and $v = \langle 8 \rangle$ and $x = 7$</code>
	<code>v.push(x);</code>
Post-state	<code>u = $\langle \underline{6}, 9 \rangle$ and $v = \langle \underline{7}, 8 \rangle$</code>

Figure 5. Example tracing of the code in Figure 4

Now we consider the case when the same object is used as a repeated argument, as in the call `transferTop(u,u)`. The first issue here is a suitable interpretation of the specification. For example, we can substitute argument `u` in the ensures clause of `transferTop` for both formal parameters in the specification in Figure 3. This leads to “`u` is unchanged except that it has lost its top entry, and `u` is

² 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.

unchanged except that it has acquired the top entry of #u as its new top entry.” That is, this interpretation of the specification says that stack u would have both lost its top entry and also that it is unchanged, because its top entry becomes its former top entry. Under this interpretation, the ensures clause becomes equivalent to “false”, and it cannot be satisfied by any implementation. Of course, the implementation in Figure 4 does not meet this specification.

There are several possible responses to this situation, as we describe in the following subsections. A key issue here is whether a procedure call with potentially repeated arguments can be treated as a clean step. That is, does a specific approach for handling repeated argument passing require a reference-value distinction in reasoning about procedure calls? To determine this, we assume that all steps leading up to a call with repeated arguments are clean, which effectively means that no aliases exist in the state before the procedure is called. Therefore, the only potential source of aliasing is when arguments are repeated in the call itself. This assumption allows us to compare all approaches uniformly.

2.2. A First Response

The most obvious response to the problem is to introduce a reference-value distinction explicitly in the specification as shown in Figure 6. In Figure 6, t is a reference to the actual argument object, while Contents(t) denotes its value³. This distinction is necessary to specify different behaviors for the different cases of when s and t are aliases and when they are not. When using this specification, to understand and reason about calls to transferTop one must make a distinction between references and values. What is worse, this complexity cannot be limited to just procedures such as transferTop where repeated arguments are an obvious issue. Doing so would require that different models be used for the specification of different methods for a single abstract data type, which would cause too many complications in practice. Furthermore, clients will need to distinguish between references and values so that they may reason about calls to procedures, such as transferTop, whose specifications make this distinction; hence it will be necessary to express the specification in a way that supports such reasoning. Finally, in languages with subtyping, such as Java, Stack would be a subtype of Object, and so even procedures like push would have the possibility of repeated arguments (i.e., the stack and the object being pushed on the stack could be the same). For all these reasons, the reference-value distinction pervades the specification shown in Figure 7.

```

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

```

³ It is possible to make the distinction of Contents less apparent in the notation, for example, by using *s to denote the contents of Stack s. It is also possible to allow the context to determine when a name stands for a reference to an object and when it stands for its value. We have not done so here so that the issues are more apparent. Also, it makes the procedure call proof rules more straightforward—even in the presence of aliasing—as illustrated later in this paper.

Figure 6. Specification of transferTop with an explicit reference-value distinction

The formal specification in Figure 7 is given in the RESOLVE notation [45] for a simplified Stack type. To introduce a notion of addresses, this specification uses an external memory manager facility, `Memory_Manager_Fac`. This external facility defines a mathematical type `Location` (informally an address) as a mathematical set. The only thing about `Location` of interest here 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 mapping `Contents` describes the value of a stack as a mathematical string of entries, given a stack location, thus making explicit the reference-value distinction.

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

public Stack();
  updates Contents;
  ensures Contents(this) =  $\Lambda$  and ( $\forall r$ : Stack if  $r \neq$  this then Contents( $r$ ) = #Contents( $r$ ));

public void push(Object x);4
  updates Contents;
  ensures this = #this and Contents(this) =  $\langle \#x \rangle \circ$  #Contents(this) and
    ( $\forall r$ : Stack, if  $r \neq$  this then Contents( $r$ ) = #Contents( $r$ ));

public Object pop();
  updates Contents;
  requires Contents(this)  $\neq \Lambda$ ;
  ensures this = #this and #Contents(this) =  $\langle$ pop() $\rangle \circ$  Contents(this) and
    ( $\forall r$ : Stack, if  $r \neq$  this then Contents( $r$ ) = #Contents( $r$ ));
```

Figure 7. A specification of a Stack component that accounts for references

⁴ In principle, it is necessary to distinguish the references and values of all objects, such as x . While doing so might help motivate the issues in this paper further, it provides no additional insight. Therefore, to keep the presentation focused, we make the reference-value distinction for only Stack objects in this paper.

Based on this modeling, the rest of the specification describes `Stack`'s public operations. The constructor gives an unoccupied location as the value to a newly created stack, `this`. Immediately after initialization, `Contents(this)` is the empty string, denoted by Λ . The operations `push` and `pop` change `Contents` at location `this`. In the specifications “`o`” denotes string concatenation. The specification makes the impact of the reference-value distinction apparent because it asserts explicitly that when the contents of “`this`” is changed, then the contents of all other stack variables remain unaffected. This assertion is also called the frame property in the literature [5][41]. In this and all other specifications in this paper, variables that are *not* mentioned in the parameter list or **updates** clause are assumed to remain unchanged.

```

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

```

Figure 8. A formal specification of the `transferTop` procedure

Given the specification for `Stack` in Figure 7, it is possible to reason formally that the code in Figure 4 is correct with respect to the formal specification of the `transferTop` procedure in Figure 8. In Figure 8, the superscript “`R`” is a postfix operator that reverses a string, and `|Contents(s)|` is the length of the string `Contents(s)`.

There are two general observations regarding a proof of correctness. One is that it would be case-based, the two cases being when `s = t` and when `s ≠ t`. Distinguishing these cases requires a reference-value distinction. The other observation is that the proof would keep two kinds of information for each variable, its direct value (which is a location or reference) and its indirect value (the location's contents—in this case the string value of a stack). These complexities in formal analysis are the result of allowing steps that are not clean (here, passing possibly repeated references as arguments). They foreshadow corresponding difficulties in software understanding and maintenance. This is apparent from the Appendix where formal reasoning involving `transferTop` is given using two approaches, one that requires the reference-value distinction, and another that does not.

Some of these difficulties are inherent in current programming languages, which often make implicit distinctions between references and values by context. For example, the left side of an assignment statement is a reference context, and the right side is a value context. Furthermore, some operations (like `push` on stacks) use and affect the values of objects, and some, like the “`==`” primitive in Java, work on the references directly. Some of the advantages in notation enjoyed by Eiffel [39], JML [34] and similar specification languages result from exploiting this contextual distinction between references and values.

2.3. A Disciplined Response

One step towards eliminating the need for a reference-value distinction in specification and reasoning is to preclude repeated arguments. This approach, when generalized to solving the larger reference-value distinction problem, amounts to preventing aliases from occurring in the first place. The manner in which repeated arguments are prevented has an effect on the flexibility of the language. We distinguish and describe two forms of the disciplined approach: a rigid version and a relaxed version.

```
uses Clean_Stack_Template;  
public void transferTop(Stack s, Stack t);  
  requires |s| > 0;  
  ensures sR o t = #sR o #t and |s| = |#s| - 1
```

Figure 9. A specification of the transferTop procedure under the rigid discipline

In the *rigid* version of the disciplined approach, the language disallows any constructs that could potentially lead to repeated arguments during at run-time. For example, the language would disallow calls such as transferTop(u,u). One of the main advantages of this approach is that it makes it possible to avoid a reference-value distinction in specifications. In particular, the case distinction made in the specification shown in Figure 8 is no longer necessary, because the case where s and t are aliased cannot arise. This simplification is illustrated by the specification given in Figure 9. Unlike the specification in Figure 8, the uses of s and t that appear in this specification denote values rather than references. Furthermore, there is no Contents variable appearing in an **updates** clause. This is not because a Contents variable exists but does not change, it is because there *is no* Contents variable. Since there is no need to make a distinction between references and values, there is no need to introduce a Contents variable that maps references to values. Under our assumption that all steps leading up to a procedure call were clean, we know that no prior steps forced us to introduce a Contents variable. Hence, we can use a simpler, value-based specification of Stack such as the one shown in Figure 10.

```
/* Clean_Stack_Template */  
class Stack;  
  uses String_Theory, Memory_Manager_Fac;  
  type Stack is modeled by Str(Object);  
  
public Stack();  
  ensures this =  $\Lambda$ ;  
  
public void push(Object x);  
  ensures this =  $\langle \#x \rangle$  o #this;  
  
public void pop(Object x);  
  requires this  $\neq$   $\Lambda$ ;  
  ensures #this =  $\langle x \rangle$  o this;
```

Figure 10. A clean specification of a Stack component

To always use this value-based specification with the discipline of avoiding repeated arguments (and other clean approaches), other sources of aliasing such as reference assignment must be avoided. Reference assignment constitutes a step that is not clean, and it would force us to introduce a Contents variable into our model of the program.

A significant drawback of the rigid disciplined approach is that it restricts certain desirable language features. For example, in the call `transferTop(a[i],a[j])`, where `a` is an array of stacks, a compiler cannot, in general, determine if `i` will equal `j` during execution. Since the rigid approach needs to ensure that repeated arguments can never occur during execution, it must be conservative and prohibit array variables as arguments. Furthermore, the rigid approach would likely be incompatible with user-defined collection types that can return object references (as in Java), since statically detecting them would probably require programmer-supplied annotations and a conservative type system. Alternatively, one could use a whole-program static analysis.

Unlike the rigid version, a *relaxed* version of the disciplined approach precludes repeated arguments at run-time. For example, in Euclid [36], attempting to pass repeated arguments by reference results in the program aborting execution. Typically this approach requires making a distinction between references and values during static analysis. That is, unless the language can guarantee that all references returned by expression evaluation are unique, a run-time determination will not avoid the reference-value distinction in reasoning, since then to statically verify total correctness one must (statically) reason about avoiding passing repeated arguments, and to do so one must reason about references. Therefore, precluding repeated arguments only avoids the reference-value distinction if some other mechanism guarantees that all references are unique.

Targeted forms of the disciplined approach may incorporate aspects of both the rigid and relaxed approaches to avoid a reference-value distinction with only limited restrictions on flexibility. For example, a language that required expressions to return values rather than references might still allow array variables. Static detectability would then require proof rule families which would require a client to avoid all possible combinations of array indices that resulted in repeated arguments. In the call `transferTop(a[i],a[j])` the verification system would add an obligation that `i` cannot equal `j`. Calls such as `transferTop(u,u)` could still be caught statically by the compiler. In general, there will always be a tradeoff in these disciplined approaches between reasoning complexity (caused by the reference-value distinction) and flexibility of the language.

2.4. Alias Dispatching Response

A third response to the repeated argument problem is a combination of the first two responses, and it requires a new language feature. In the alias dispatching approach [33], one writes several “multibodies” for a procedure, each of which has no aliasing among its formal parameters. Together these multibodies implement a procedural abstraction that can handle repeated arguments. For example, one could implement the specification of Figure 8 as follows.

```
public void transferTop(Stack s, Stack t) imports() {  
    Object x;           // multibody that is run when s ≠ t  
    x = s.pop();  
}
```

```

    t.push(x);
} alias(s,t) {
    skip;           // multibody that is run when s = t; t cannot be used in this body
}

```

Figure 11. Handling of repeated arguments in an alias dispatching language

Alternatively, one can change the specification to rule out aliasing among arguments, as in the specification of Figure 12. This specification is implemented by the code shown in Figure 4, because in this approach, missing bodies signal an error when they are invoked in response to a call.

```

uses Ref_Val_Stack_Template;
public void transferTop(Stack s, Stack t);
requires  $s \neq t$  and  $|\text{Contents}(s)| > 0$ ;
ensures  $\text{Contents}(s)^R \circ \text{Contents}(t) = \#\text{Contents}(s)^R \circ \text{Contents}(\#)$  and
 $|\text{Contents}(s)| = \#\text{Contents}(s) - 1$  and
 $(\forall r: \text{Stack} \text{ if } r \neq s \text{ and } r \neq t \text{ then } \text{Contents}(r) = \#\text{Contents}(r))$ ;

```

Figure 12. A specification that explicitly prohibits repeated arguments

Unfortunately, as seen from this example, specifications written to preclude aliases also involve a reference-value distinction. Furthermore, since the alias dispatching approach by itself does not rule out repeated storage of references, it needs to be based on `Ref_Val_Stack_Template` (Figure 7). The alias dispatching approach can be seen as a variant of both the standard and disciplined approaches. It is a variant of the standard approach because it allows repeated arguments in calls where the programmer has written a multibody to handle the pattern of repeated arguments that appears among the actuals in the call. It can also be seen as an aid to careful reasoning with the first approach, because it helps make explicit in the code cases that are apparent in the specification (and because the coding will point out cases that the specification should consider).

Another way to see the alias dispatching approach is as a relaxation of the disciplined approach. Like the disciplined approach, it restricts the aliasing patterns that may be used in calls to a procedure, but instead of just restricting the caller to the pattern where there are no repeated arguments, it allows the specification of several patterns of repeated arguments for each procedure. However, unlike the rigid disciplined approach, it does not, by itself, support a clean view of objects.

3. Techniques for Avoiding the Reference-Value Distinction

The essential problem with passing references as arguments is that it introduces aliasing. Once aliasing is introduced, by either repeated arguments or repeated storage of references, reasoning must distinguish between references and values because the clean treatment of objects as values is no longer sound.

3.1. The Unique References Approach

The *unique references approach* allows each object to be referred to by at most one reference. It is the most direct attack on the aliasing problem, and hence largely solves the reference-value distinction problem.

One implementation of the unique references approach uses linear types [4][40]. The basic idea of a linear type system is simple: the reference stored in a variable of a linear type can only be read once, and the act of reading it places a null reference in the variable. For example, suppose that s is a Stack variable with $\langle 3,6,2 \rangle$ as its abstract value. After s is read, it will be a null reference. There is also a special *Replica* action on a reference variable; this action leaves the variable unchanged, but copies its value. There are variants of linear types, such as Boyland’s alias burying approach [7], which provide more flexibility than the semantics we have described, but this will suffice for our purposes.

When a variable of a linear type is repeated as an argument, the value of all but the first of the repeated arguments become null references, because a variable can only be read only once. This prevents aliasing among the corresponding formals. Similarly, if a global variable is passed as an argument, then its value is also turned into a null reference, which in turn prevents aliasing between the global and the corresponding formal parameter. Assuming arguments are evaluated from left to right, in the call to `transferTop(u,u)`, the second formal receives a null reference. This execution produces an error when the code (as in Figure 4) attempts to pop the second stack.

Is it possible to specify when a variable is the null reference but avoid making a reference-value distinction? Technically, the answer is *yes*—one can introduce a specification primitive, `defined(v)`, which is true precisely when the variable v does not contain a null reference. With this primitive one can specify `transferTop` in a way that both prohibits repeated arguments and avoids references, as shown in Figure 13.

```
/* uses a modified version of Clean_Stack_Template */  
public void transferTop(Stack s, Stack t);  
    requires defined(s) and defined(t) and |s| > 0;  
    ensures defined(s) and defined(t) and sR o t = #sR o #t and |s| = |#s| - 1
```

Figure 13. Specification under the unique references approach

Though it may be technically possible through the use of such syntactic sugar to avoid a reference-value distinction with this approach, the added complexity of reasoning would be the same whether one populates specifications with a multitude of $x \neq \text{null}$ assertions or a multitude of `defined(x)` assertions. This problem can be so considerable that a key use of ESC/Java is to detect potential violation of null variable accesses in specifications [36]. It is important to note that the specifications of components such as the Stack in `Clean_Stack_Template` (Figure 10) would have to be amended to handle all the potentially undefined variables or null references. Thus, the unique references approach is not an ideal way to eliminate the reference-value distinction.

As noted above, the specification in Figure 13 actually precludes repeated arguments because the second repeated formal, t , will receive a null value, which violates the `requires` clause. One could allow repeated arguments by permitting t to be undefined, but one would have to modify the

ensures clause so that t did not appear ($\#t$ could still appear), creating a specification of dubious worth. Another feature of this approach is that the order that parameters are passed to the procedure is important when arguments are repeated. This difference from previous approaches is also a characteristic of the swapping and call-by-value-result approach.⁵ Since this ordering is important only when arguments are repeated, and the linear typing approach effectively precludes repeated arguments, we leave a discussion of this feature to the next subsection.

3.2. The Swapping Approach

Swapping has been suggested as an alternative to assignment to minimize aliasing as early as 1976 in the form of a “value exchange” operation in [29] and more recently in [20][40]. This is because swapping two variables never introduces new aliases. The idea of swapping is simple. If s and t are two Stack objects with abstract values $\langle 3,6,2 \rangle$ and $\langle 7,7,9,1,2 \rangle$ respectively, swapping s with t leads to their values being interchanged. Swapping takes constant time independent of the sizes of objects because, in the implementation, only references to the two objects need to be swapped.

Swapping can also be used for efficient parameter passing. More importantly, call-by-swapping makes it possible to handle repeated arguments cleanly. The mechanics of call-by-swapping (relevant for language implementers) are simple. In principle, the actual objects are swapped with the corresponding formal objects that have been initialized with values appropriate for their types. (In practice, most initializations turn out to be unnecessary and can be optimized out by a compiler.) The called procedure is executed on the formal objects, and on return the formals are swapped with the actual objects again. The swapping of formals and actuals takes place sequentially in left-to-right order on call and right-to-left order on return. The order has no bearing in the common case when arguments are not repeated (but does matter when they are).

From a user perspective call-by-swapping is even more straightforward: When no arguments are repeated, call-by-swapping is as simple as call-by-value-result; when arguments *are* repeated, the first formal gets the value of the repeated argument on call (the others get initial values), and the repeated argument gets the value of the first formal on return. To see why this is the case, consider the call $\text{transferTop}(u,u)$. When the procedure is called, u 's value goes to the first corresponding formal, s . All subsequent corresponding formals (in this case only t), get initial values for their type (because of left-to-right order of swapping with initial objects). On return, the repeated argument gets the value of the first corresponding formal. Values of subsequent corresponding formals become irrelevant because they are not returned to the caller. To illustrate this more formally, consider the value-based specification of transferTop and its implementation shown in Figure 14.

```

/* uses Clean_Stack_Template */
public void transferTop(Stack s, Stack t)
  requires |s| > 0;
  ensures sR o t = #sR o #t and |s| = |#s| - 1;
{
  Object x;
  s.pop(x);
  t.push(x);
}

```

⁵ Call-by-value-result is not considered here because of its inefficiency.

Figure 14. Specification and implementation for swapping approach

Using the user perspective in the interpretation of the specification for the call `transferTop(u,u)`, we see that the procedure requires $|u| > 0$ and it ensures $u^R \circ t = \#u^R \circ \Lambda$ **and** $|u| = |\#u| - 1$. On simplification, if the requires clause holds on call to `transferTop`, then on return w is the same as it was except that it is missing its top (which was transferred to stack t and never returned). This is exactly what the implementation in Figure 14 does when parameters are passed by swapping—the top of s (which has u 's incoming value) is popped into x , and then pushed on the initialized stack t which is not returned to the caller.

The implementation in this example is appropriate for a language with call-by-swapping. It differs from the implementation given in Figure 4, which is more appropriate for a language such as Java that relies on reference copying for parameter passing and data movement. This underscores the fact that language design decisions can impact programming style. Just as programmers had to get used to a certain style of programming with Java, programmers would have to adapt to a style of programming compatible with call-by-swapping.

While the swapping approach is designed to work seamlessly whether or not arguments are repeated, it is possible to disallow argument repetition through specification, if so desired. The important observation here is that this goal can be accomplished without a reference-value distinction. For example, consider the variation of the `transferTop` procedure, given in Figure 15, where the roles of s and t are reversed. This procedure behaves as expected when there is no repetition of arguments, but the call `transferTopTo(u,u)` is illegal because the second argument t gets the initial stack value (Λ) which is a violation of the requires clause $|t| > 0$. This is justifiably so because otherwise the pop operation will be invoked on t , the empty stack.

```
public void transferTopTo(Stack s, Stack t)
  requires |t| > 0;
  ensures tR o s = #tR o #s and |t| = |#t| - 1;
{
  Object x;
  t.pop(x);
  s.push(x);
}
```

Figure 15. Precluding repeated arguments without introducing references

Without involving references, call-by-swapping gives a meaning to a specification and its implementation in a way that makes the implementation's correctness independent of whether or not callers repeat arguments. Swapping allows arguments to be repeated or precluded at the discretion of specifiers without a need to introduce a reference-value distinction. Implementers can write a single implementation without concern for potential argument repetition. Callers of procedures, however, will need to understand the behaviors of their calls when they repeat arguments as in the case of every approach discussed in this paper.

A variation of the swapping approach is necessary to handle parameter passing in the presence of subtyping. The symmetry of swapping raises an issue in transferring the value of a subtype object to a super type object, when the formal parameter is of the super type and the actual argument is of a subtype [40]. To address this problem, temporary objects that correspond to the dynamic types of

the actual arguments will be created and swapped, instead of creating objects of the formal parameter's supertype. When a subtype object is repeated, the first formal argument will have the value of the object whereas the second and subsequent arguments will have an initial value of the subtype (or a copy of the original value in copy-based variations). Importantly, no reference-value distinction is necessary in handling subtypes.

3.3. Clean Views and Clean Steps Revisited

The call-by-swapping approach to parameter passing makes a procedure call a clean step because it allows a client to maintain a clean view of objects. As previously noted, aliasing in high-level modules is typically introduced by assignment or parameter-passing or some combination of the two. Therefore, the ability to minimize the reference-value distinction depends not only on the parameter-passing approach, but on the approach to general data movement as well. Many of the parameter passing approaches described in this paper correspond to specific data movement mechanisms that have similar properties in terms of efficiency and complexity. For example, the unique references approach corresponds to using destructive reads for data movement—it is efficient and it introduces null references. The swapping approach corresponds to using swapping for data movement—it is efficient and avoids aliasing. Alternative approaches to data movement can be used with their corresponding parameter passing approaches to help minimize the reference-value distinction. Data movement mechanisms that are not as well-known may yet inspire other approaches to parameter passing and language design [49]. In the next section, we will present a rule template for parameter passing that is parameterized by a data movement mechanism.

4. Verification of Correctness

This section (together with the Appendix) formalizes the impact of the reference-value distinction on formal reasoning. Since we have taken the route of accounting for reference-value distinctions explicitly in the specifications, the proof rules themselves are quite simple regardless of the parameter passing approach employed; in fact, the same template is shown to work for all approaches. Some classical procedure call rules preclude aliasing among arguments [21]. Some others encourage clients to distinguish between cases in which there is no potential for aliasing (making the proofs themselves much easier) and cases in which aliasing is a possibility [8][17].

4.1. Proof Rules

The first rule, given in Figure 16, is a common rule for verification of correctness of a procedure. The precondition of the procedure is assumed and the postcondition is to be confirmed following the code for the procedure. Here, `assertive_body` includes the statements and assertions (e.g., loop invariants) in the code for `p`. The `Context` prefix in the rule serves as a placeholder for any external information that is necessary for verification, such as the specifications of other procedures called by `p`. We simplify the discussion by only dealing with the case of two formal parameters.

Context/ **assume** pre; assertive_body; **confirm** post;
Context/ **void** p(T1 x, T2 y); **requires** pre; **ensures** post; {assertive_body}

Figure 16. A proof rule for procedure body verification

We illustrate the application of the rule through the verification of correctness for an implementation of `transferTop` in the Appendix for both call-by-swapping and call-by-reference-copying approaches. While the same rule works for showing correctness under all parameter passing approaches, the specifications and/or implementations in approaches that have a reference-value distinction typically need to be case-based to handle repeated arguments. The corresponding complexity is then seen in the reasoning.

First, we present a straightforward rule for procedure calls in Figure 17 for the (rigid) disciplined approach in which repeated arguments are syntactically precluded. The Context for this rule must include the specification the called procedure, `p`. In Figure 17, 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 obligates that two conjunctions be proved. The first of these is the `requires` clause of `p`, which needs to be proved when the formals, `x` and `y`, are replaced by the actuals, `a` and `b`. In addition, 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 output values may result for the same input parameters. Therefore, the second conjunction states that as long as the output values of the arguments satisfy the post-condition, `Q` must be confirmed. The two verification variables `?a` and `?b` to denote any possible output value. The formal output names `?a` and `?b` replace actual arguments `a` and `b` before `Q` is confirmed so that the names used in the two sides of the implication are consistent.

Context\ assertive_code; **confirm** pre [x ← a, y ← b] **and**
 $\forall ?a: T1, \forall ?b: T2, \text{post}[\#x \leftarrow a, x \leftarrow ?a, \#y \leftarrow b, y \leftarrow ?b] \Rightarrow Q'[a \leftarrow ?a, b \leftarrow ?b];$
Context\ assertive_code; p(a, b); **confirm** Q;
where Q' = Q with substitutions for ?a, and ?b to avoid name conflicts.

Figure 17. A proof rule for verification of procedure calls without repeated arguments

We next present a more general proof rule template (Figure 18) that is suitable for verification when parameters are passed by reference copying, value copying, unique references, or the swapping approach. This rule is parameterized by a transfer operator, written “<-,” that stands for the transfer operator used to transfer the value from actual arguments to temporary ones. For the copying approaches, “<-” stands for “=” (or sometimes “:=”), the assignment operator. For the unique references approach it denotes a destructive read. For the swapping approach, it stands for “:=,” the swap operator.

```

Context\ assertive_code; T1 %ax; T2 %by; %ax ← a; %by ← b;
confirm pre [x ← %ax, y ← %by] and
∀?ax: T1, ∀?by: T2, post[ #x ← %ax, x ← ?ax, #y ← %by, y ← ?by] ⇒ Q'[a ← ?ax][ b ← ?by];
Context\ assertive_code; p(a, b); confirm Q;
  where Q' = Q with substitutions for %ax, %by, ?ax, and ?by to avoid name conflicts.

```

Figure 18. A general proof rule template for verification of procedure calls

Parameter passing by all the approaches can be thought of as involving the creation of two initialized local variables, named %a_x and %b_y to which the values of the corresponding actual arguments are transferred. These verification variables are given special names beginning with %. 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. When no arguments are repeated and the variables a and b are distinct, it is easy to see that the variable declarations and the transfer statements have no impact other than use of the names %a_x and %b_y in lieu of a and b in subsequent assertions.

The formal output names ?a_x and ?b_y replace actual arguments a and b (in that order), before Q is confirmed so that the names used in the two sides of the implication are consistent; the ordering in which results are returned is reflected indirectly 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. More specifically, if no argument is repeated (or precluded from repetition) where a result is expected, then the ordering does not matter. In the swapping and unique references approaches, the order in which parameters are passed also has an impact on the resulting state.

A syntactic variation of the rule for function call verification is given in Figure 19. It is needed, for example, in the function assignment statement “x = s.pop();” of Figure 4. Unlike a mathematical function, this rule allows a function to return one of many outputs as is possible in Java.

```

Context\ assertive_code; T1 %ax; T2 %bf; %ax ← a; %bf ← b;
confirm pre [x ← %ax] and
∀?ax: T1, ∀?bf: T2, post[ #x ← %ax, x ← ?ax, f() ← ?bf] ⇒ Q'[b ← ?bf][ a ← ?ax];
Context\ assertive_code; b ← f(a); confirm Q;
  where Q' = Q with substitutions for %ax, %bf, ?ax, ?bf to avoid name conflicts, and where f()
  denotes the returned value of f.

```

Figure 19. A general proof rule template for verification of function calls

To more fully illustrate the use of the rules, we have included a proof of correctness for transferTop in the Appendix, and verify a client program that calls transferTop with repeated arguments. The use of the procedure call rule for the common calling situation is illustrated in verification of calls to the push and pop operations in the transferTop code. The use of the same call rule when arguments are repeated is seen in the proof of the client program. The proofs are shown using both call-by-reference-copying and call-by-swapping approaches to illustrate the impact of the reference-value distinction.

4.2. Hybrid Variations

This section considers hybrid variations of the proof rules for parameter passing given above.⁶ The procedure call proof rule is set up so that all but the first result parameter is ignored when an argument is repeated in multiple result positions. It is easy to reverse this, i.e., make the last result be the interesting one, ignoring others, if one chooses to do so by simply rewriting the substitutions in the last conjunction as $Q'[b \leftarrow ?b_y][a \leftarrow ?a_x]$; Another alternative is to follow a disciplined approach when the same argument appears as multiple result parameters. A system in such a case may give a warning or selectively disallow repetition in a rigid or relaxed fashion. Variations more specific to the swapping and unique references approach include use of parameter modes to override the strict left-to-right order in which arguments are passed to procedures; for example, expressions may be evaluated and transferred before other variables are transferred regardless of their relative ordering in the list of parameters.

In the case of small objects, such as integers, value copying is cheap. Therefore, such small objects should be copied and passed by value whether or not arguments are repeated. A multiplication operation call on integers such as $i = \text{mult}(j, j)$ should result in i that is equal to $j * j$. To achieve this, languages can introduce specialized parameter modes to allow value copying. In a Java-like version of RESOLVE, for example, the operation signature would be written as $\text{mult}(\text{evaluates Integer } x; \text{evaluates Integer } y): \text{Integer}$; and a compiler would interpret the above call as $i = \text{mult}(\text{Replica}(j), \text{Replica}(j))$ because the evaluates mode would alert the compiler to expect expressions. When variables are encountered where expressions are expected, the Replica operation (similar to clone in Java) is automatically invoked. The language allows variables to be passed for evaluates mode parameters only when there is a replication operation for the type of the variable. Another variation is to use value replication whenever an argument is repeated if a corresponding replication operation is available.

4.3. Summary of Approaches

This paper can be used to provide a framework for a specification and/or programming language designer within which to engineer parameter passing, so that one gets efficiency and the expected effects in typical situations, and can trade-off such considerations against complexity. Table 1 summarizes the approaches presented in this paper. When the approaches are combined, obviously, we would also expect a combination of benefits and drawbacks.

⁶In principle, a language could introduce features to allow specifiers to dictate how each parameter should be passed. Though such an approach gives maximum flexibility, it also introduces the greatest potential for confusion.

Approach	Benefits	Drawbacks
Copy-by-Value-Result	Clients may treat all objects as values.	Copy operators not available for all non-trivial objects; value copying is inefficient.
Call-by-Value (using reference variables)	Standard approach in programming languages such as Java.	References are introduced in reasoning about repeated arguments.
Disciplined (Rigid)	Clients may treat all objects as values.	Array variables and global variables cannot be passed as parameters.
Disciplined (Relaxed)	Closer to the standard approach.	References are introduced in (static) reasoning about repeated arguments.
Alias Dispatch	Avoids aliasing among formal parameters in a multibody.	References are introduced in reasoning about repeated arguments. Requires a new language feature.
Unique References (Linear Typing)	Client may treat all defined objects as values.	Instead of a general reference-value distinction, a distinction between defined and undefined (null) objects is introduced.
Swapping	Client may treat all objects as values.	Requires a new language feature.

Table 1. A summary of alternative techniques to handle repeated arguments

5. Related Work and Summary

In the 1970’s, when pointers were first being introduced into high-level languages, a handful of papers appeared warning of their detrimental effects on reasoning [22][29]. Hoare introduced recursive data structures as a partial alternative to references in [23], and Kieburtz proposed replacing pointers with two low-level data structures—lists and trees—that could encapsulate their most important behavior [29]. Reasoning about references typically involves considering a *common store* that holds object values. Some languages, like Euclid, divide this common store into collections [27] to aid reasoning, but the reference-value distinction is still needed. Luckham and Suzuki, working on the specification and verification of Pascal programs, modeled the common store explicitly in specifications [38]. More recently, Weide and Heym illustrate the use a common store to specify and reason about Java components [48]. But all of this work introduces a reference-value distinction and does not support the clean view. In this paper, the “Contents” function serves as access to the common store.

With the introduction of popular object-oriented programming languages that employed an indirect model of storage, reasoning about references and the aliasing they caused took on a renewed significance. A substantial body of work has gone into developing specification languages and logics geared toward object-oriented programs [1][13][34][35][41]. Some, like JML [34] and Object-Z [13], use reference semantics to capture object identity, but reasoning in the face of potential aliasing remains a problem. Smith suggests using alias control techniques to make such reasoning systems more tractable [47], but such techniques do not themselves support a clean view. 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” (p.1). This provides evidence for our goal of supporting a clean view more directly.

Hogg et. al. surveyed the problem of object aliasing in [25]. Though encapsulation was characterized as one of the strengths of object-oriented programming [39], Hogg called the notion that objects provide encapsulation “the big lie of object-oriented programming” ([24], p.271), pointing to aliasing between modules as evidence to the contrary. Consequently, a number of proposed solutions to the aliasing problem have focused on encapsulation and alias control in object-oriented languages. Hogg’s *islands* [24] 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 [43]. Their method is aimed primarily at container objects and is aimed at preventing the exposure of an object’s *representation* (typically its state variables) and ensuring that the object does not depend on the mutable state of any of its *arguments* (typically the “contents” of a container object). 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]. None of these alias control techniques, however, eliminates aliasing, none eliminates repeated arguments, and thus all do not directly support a clean view of software. Numerous techniques for pointer analysis have been proposed, but the computational difficulty of alias detection is high [28][32]. Such techniques may aid reasoning by helping to avoid extra case analysis when doing verification, but they do not themselves eliminate the reference-value distinction.

Aliasing appears unavoidable in some situations, like the implementation of typically linked data structures such as lists and trees. Hoare suggested that languages introduce recursive data structures to take care of these cases [23], and Kieburtz proposed list and tree recursive data types so that “computations involving address constants as data can be pushed down to a very low level of abstraction” ([29], p. 96). These approaches borrow from functional programming, but techniques for encapsulation of aliasing suggest that similar results may be obtained using methods more familiar to imperative programmers [2][15][24]. Examples of lists that can be implemented with references while allowing their objects to be treated as values can be found in [44][46]. Various low-level components that encapsulate the reference behavior of typically linked data structures have been proposed [31][44]. Such “pointer” components are not clean because they provide an indirect model storage, but since their functionality is targeted to linked data structures they are less likely to be abused than traditional—and more general—pointers.

The use of reference semantics can be confined to low-level components only if aliasing can be avoided in higher-level components. In languages with reference variables, aliasing is introduced routinely through assignment and parameter passing. While we have focused on parameter passing in this paper, some approaches such as linear typing and swapping provide a more general solution to data movement. Baker’s linear type system relies on destructive reads for data movement [4]. Once a variable x reads y ’s value, y becomes null. Baker’s system applies to functional languages, but others have proposed destructive reads in an imperative setting [24][40]. Hogg’s paper on object aliasing mentions the swapping approach [20] as a way to prevent aliasing but notes that it requires a paradigm shift in programming, cautioning that “it is unclear whether this paradigm can mesh well with mainstream object oriented programming techniques” ([25], p.8). In response to this, Hollingsworth et al. talk about their success with building a commercial product using the RESOLVE/C++ discipline. Out of 250 components in the system, only 7 used the assignment operator, the rest using a swap operator. They note that “the ‘different paradigm’ that one must learn to use swapping effectively is but a minor variant of the traditional imperative programming

style” ([26], p.13). Deep-copying remains an alternative in languages that provide both value objects and reference objects. Almeida recommends it for balloon types and points to optimizations like lazy evaluation to lessen the performance cost [2], and C++ programmers employ it routinely for value-like objects [11][42].

It is clear that current programming practice in languages such as Java and C++ introduces complexity into specifications and reasoning due to an unavoidable reference-value distinction. Using the example of repeated arguments to procedures, we have shown that there are alternative approaches to parameter passing that can avoid this reference-value distinction without resorting to copying. We are optimistic that through such mechanisms the impact of the reference-value distinction can be minimized, allowing programmers to devote more energy to the complexity inherent in the program, and less to the complexity inherent in the language.

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] C.A.R. Hoare, "Procedures and Parameters: An Axiomatic Approach," E. Engeler (ed.), *Proceedings Symposium on Semantics of Algorithmic Languages*, 1971, pp. 102-116.
- [22] 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.
- [23] C.A.R. Hoare, "Recursive Data Structures," *International Journal of Computer and Information Science*, vol. 4, no. 2, 1975, pp. 105-132.
- [24] J. Hogg, Islands: "Aliasing Protection in Object-Oriented Languages," *Proceedings OOPSLA '91*, volume 26 of *ACM SIGPLAN Notices*, 1991, pp. 271-285.

- [25] 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>
- [26] 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.
- [27] J. Horning, "A Case Study in Language Design," F.L. Bauer and M. Broy (eds.), *Program Construction*, LNCS 69, Springer-Verlag, New York, 1979.
- [28] 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.
- [29] R.B. Kieburtz, "Programming Without Pointer Variables," In *Proc SIGPLAN '76 Conf. on Data: Abstraction, Definition and Structure*, 1976. ACM Press.
- [30] A. Koenig and B. E. Moo, "Rethinking How to Teach C++," *Journal of Object-Oriented Programming*, Feb 2001, pp. 25-27.
- [31] G. Kulczycki, M. Sitaraman, W.F. Ogden, and J.E. Hollingsworth, "Capturing the Behavior of Linked Data Structures," *Proceedings RESOLVE 2002 Workshop*,
<http://people.cs.vt.edu/~edwards/RESOLVE2002/proceedings/Kulczycki/>, 2002.
- [32] W. Landi, "Undecidability of Static Analysis," *ACM Lett. Program. Lang. Syst.*, vol. 1, no. 4, Dec. 1992, pp. 323-337.
- [33] 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.
- [34] 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.
- [35] K.R.M. Leino and G. Nelson, *Data Abstraction and Information Hiding*, tech. Report 160, Compaq Systems Research Center, Palo Alto, CA, 2000.
- [36] 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.
- [37] 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.
- [38] 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.
- [39] B. Meyer, *Object-oriented Software Construction*, 2nd ed., Prentice-Hall, New York, 1997.

- [40] 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.
- [41] P. Müller, *Modular Specification and Verification of Object-Oriented Programs*, Springer-Verlag, 2002, vol 2262 Lecture Notes in Computer Science.
- [42] D.R. Musser, G.J. Derge, and A Saini, *STL Tutorial and Reference Guide*, 2nd ed., Addison-Wesley, 2001.
- [43] 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.
- [44] W.F. Ogden, *The Proper Conceptualization of Data Structures*, The Ohio State University, 2000.
- [45] 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.
- [46] 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.
- [47] G. Smith, "Reasoning about Object-Z Specifications," *Proceedings of Asia-Pacific Software Engineering Conference*, 1995, pp. 489-497.
- [48] 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-heyms.pdf>
- [49] 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>
- [50] J.M. Wing, "A Specifier's Introduction to Formal Methods," *IEEE Computer*, vol. 23, no. 9, Sept. 1990, pp. 8-24.

Appendix A Formal Verification with and without a Reference-Value Distinction

This appendix contains the following four proofs:

1. A proof of client code with repeated arguments using the clean specification of `transferTop` in Figure 14 and the call-by-swapping approach to parameter passing.
2. A proof of correctness for `transferTop` using the specification and implementation in Figure 14, the specification for `Stack` given in `Clean_Stack_Template` (Figure 10), and the call-by-swapping approach to parameter passing.
3. A proof of client code with repeated arguments using the specification of `transferTop` in Figure 8 and the call-by-reference-copy approach to parameter passing.
4. A proof of correctness for `transferTop` using the specification in Figure 8, the implementation in Figure 4, the specification for `Stack` given in `Ref_Val_Stack_Template` (Figure 7), and the call-by-reference-copy approach to parameter passing.

Proof #1: A proof of client code with repeated arguments using the clean specification of `transferTop` in Figure 14 and the call-by-swapping approach to parameter passing.

We want to prove the following assertive program:

```
assume |u| = 1;
transferTop(u, u);
confirm u =  $\Lambda$ ;
```

We need to prove:

```
{transferTop}\
assume |u| = 1;
transferTop(u, u);
confirm u =  $\Lambda$ ;
```

By the proof rule for procedure call (Figure 18) this reduces to:

```
assume |u| = 1;
Stack %us; Stack %ut;
%us := u; %ut := u;
confirm |%us| > 0 and
   $\forall ?u_s: \text{Stack}, \forall ?u_t: \text{Stack},$ 
  if ?usR  $\circ$  ?ut = %sR  $\circ$  %t and |?us| = |%us| - 1 then ?us =  $\Lambda$ ;
```

By the proof rule for the swap statement⁷ this reduces to:

⁷ The proof rule for swap statement is shown below, where $Q[s \leftarrow t, t \leftarrow s]$ denotes simultaneous substitution:
 Context/ assertive_code; **confirm** $Q[s \leftarrow t, t \leftarrow s]$;
 Context/ assertive_code; s := t; **confirm** Q;

```

assume |u| = 1;
Stack %us; Stack %ut;
%us := u;
confirm |%us| > 0 and
  ∀?us: Stack, ∀?ut: Stack,
  if ?usR ◦ ?ut = %usR ◦ u and |?us| = |%us| - 1 then ?us = Λ;

```

By another application of the proof rule for the swap statement this reduces to:

```

assume |u| = 1;
Stack %us; Stack %ut;
confirm |u| > 0 and
  ∀?us: Stack, ∀?ut: Stack,
  if (?usR ◦ ?ut = uR ◦ %us and |?us| = |u| - 1) then ?us = Λ;

```

The proof rule for variable declarations allows us to replace instances of the declared variables that appear in the confirm statement with their initial values.

```

confirm |u| > 0 and
  ∀?us: Stack, ∀?ut: Stack,
  if (?usR ◦ ?ut = uR ◦ Λ and |?us| = |u| - 1) then ?us = Λ;

```

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 ⇒ (|u| > 0 and
  ∀?us: Stack, ∀?ut: Stack,
  if ?usR ◦ ?ut = uR ◦ Λ and |?us| = |u| - 1 then ?us = Λ);

```

Since $|u| = 1$ we know that $|u| > 0$, and $|?u_s| = |u| - 1 \Rightarrow |?u_s| = 0$, which occurs when $?u_s = \Lambda$.

Proof #2: A proof of correctness for transferTop using the specification and implementation in Figure 14, the specification for Stack given in Clean_Stack_Template (Figure 10), and the call-by-swapping approach to parameter passing.

We need to prove:

```

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

```

By the proof rule for procedure body declaration (Figure 16) this reduces to:

```
{push, pop}\
assume |s| > 0;
Object x;
s.pop(x);
t.push(x);
confirm sR ◦ t = #sR ◦ #t and |s| = |#s| - 1;
```

By the proof rule for procedure call (Figure 18) on push this reduces to:

```
{pop}\
assume |s| > 0;
Object x;
s.pop(x);
Stack %ts; Object %xx;
%ts := t; %xx := x;
confirm ∀?ts: Stack, if ?ts = ⟨%xx⟩ ◦ %ts then (sR ◦ ?ts = #sR ◦ #t and |s| = |#s| - 1);
```

By the proof rule for swap statement and the proof rule for variable declaration this reduces to:

```
{pop}\
assume |s| > 0;
Object x;
s.pop(x);
confirm ∀?ts: Stack, if ?ts = ⟨x⟩ ◦ t then (sR ◦ ?ts = #sR ◦ #t and |s| = |#s| - 1);
```

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

```
assume |s| > 0;
Object x;
Stack %ss; Object %xx;
%ss := s; %xx := x;
confirm |%ss| > 0 and
  ∀?xx: Object, ∀?ss: Stack, if %ss = ⟨?xx⟩ ◦ ?ss then
  (∀?ts: Stack, if ?ts = ⟨?xx⟩ ◦ t then (?ssR ◦ ?ts = #sR ◦ #t and |?ss| = |#s| - 1));
```

By the proof rule for swap statement and the proof rule for variable declaration this reduces to:

```
assume |s| > 0;
confirm |s| > 0 and
  ∀?xx: Object, ∀?ss: Stack, if s = ⟨?xx⟩ ◦ ?ss then
  (∀?ts: Stack, if ?ts = ⟨?xx⟩ ◦ t then (?ssR ◦ ?ts = #sR ◦ #t and |?ss| = |#s| - 1));
```

In the confirm clause, s is the same as #s and t is the same as #t, so this reduces to:

```
assume |s| > 0;
confirm |s| > 0 and ∀?xx: Object, ∀?ss: Stack, if s = ⟨?xx⟩ ◦ ?ss then
```

$(\forall ?t_s: \text{Stack}, \text{if } ?t_s = \langle ?x_x \rangle \circ t \text{ then } (?s_s^R \circ ?t_s = s^R \circ t \text{ and } |?s_s| = |s| - 1));$

Reducing the assume-confirm statement sequence to an implication yields:

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

It suffices to show that each of the following is true for all $?x_x: \text{Object}$, $?s_s: \text{Stack}$, and $?t_s: \text{Stack}$:

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 $s = \langle ?x_x \rangle \circ ?s_s \Rightarrow |s| = 1 + |?s_s|$.

Proof #3: A proof of client code with repeated arguments using the specification of transferTop in Figure 8 and the call-by-reference-copy approach to parameter passing.

We want to prove the following assertive program:

```
{transferTop}\
assume Contents(u) = ⟨a⟩;
transferTop(u, u);
confirm Contents(u) = ⟨a⟩;
```

Contents is a mathematical global variable. Such variables are treated similar to parameter, but they are not programming variables so they are not declared or transferred. Figure 20 shows a proof rule that has been extended to account for an updated mathematical global variable G:

```
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, ∀?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\ assertive_code; P(a, b); confirm Q;
  where Q' = Q with substitutions for %a_x, %b_y, ?a_x, ?b_y, and ?G to avoid name conflicts.
```

Figure 20. A modified general proof rule template that includes a mathematical global

Applying the proof rule in Figure 20 to the assertive code reduces it to:

```
assume Contents(u) = ⟨a⟩;
Stack %u_s; Stack %u_t;
%u_s = u; %u_t = u;
confirm |Contents(%u_s)| > 0 and
  ∀?u_s: Stack, ∀?u_t: Stack, ∀?Contents: Location → Str(Object),
  %u_s = ?u_s and %u_t = ?u_t and
```

if $?u_s \neq ?u_t$ **then** $(?Contents(?u_s)^R \circ ?Contents(?u_t) = Contents(?u_s)^R \circ Contents(?u_t)$ **and**
 $|?Contents(?u_s)| = |Contents(?u_s)| - 1)$ **and**
if $?u_s = ?u_t$ **then** $?Contents(?u_s) = Contents(?u_s)$ **and**
 $(\forall r: Stack, \text{if } r \neq ?u_s \text{ and } r \neq ?u_t \text{ then } ?Contents(r) = Contents(r))$ **implies**
 $?Contents(?u_s) = \langle a \rangle;$

The proof rule for reference assignment results in the replacement of $?u_s$ and $?u_t$ with u , and the proof rule for variable declaration has no affect. The resulting assertive program looks similar to one that would have obtained if the simpler proof rule (Figure 17) were used in the last step. In the next proof, we will take a shortcut and combine the proof rule and these steps into one.

assume $Contents(u) = \langle a \rangle;$
confirm $|Contents(u)| > 0$ **and**
 $\forall ?u_s: Stack, \forall ?u_t: Stack, \forall ?Contents: Location \rightarrow Str(Object),$
 $?u_s = ?u_s$ **and** $?u_t = ?u_t$ **and** (
if $?u_s \neq ?u_t$ **then** $(?Contents(?u_s)^R \circ ?Contents(?u_t) = Contents(?u_s)^R \circ Contents(?u_t)$ **and**
 $|?Contents(?u_s)| = |Contents(?u_s)| - 1)$ **and**
if $?u_s = ?u_t$ **then** $?Contents(?u_s) = Contents(?u_s)$ **and**
 $(\forall r: Stack, \text{if } r \neq ?u_s \text{ and } r \neq ?u_t \text{ then } ?Contents(r) = Contents(r))$ **implies**
 $?Contents(?u_s) = \langle a \rangle;$

Reducing the assume-confirm sequence to an implication yields:

$(Contents(u) = \langle a \rangle) \Rightarrow (\forall ?u_s: Stack, \forall ?u_t: Stack, \forall ?Contents: Location \rightarrow Str(Object),$
 $u = ?u_s$ **and** $u = ?u_t$ **and**
(if $?u_s \neq ?u_t$ **then** $(?Contents(?u_s)^R \circ Contents(?u_t) = Contents(?u_s)^R \circ Contents(?u_t)$ **and**
 $|?Contents(?u_s)| = |Contents(?u_s)| - 1)$ **and**
(if $?u_s = ?u_t$ **then** $?Contents(?u_s) = Contents(?u_s))$ **and**
 $(\forall r: Stack, \text{if } r \neq ?u_s \text{ and } r \neq ?u_t \text{ then } ?Contents(r) = Contents(r))$ **implies**
 $?Contents(?u_s) = \langle a \rangle;$

The consequent contains the assertions $u = ?u_s$ and $u = ?u_t$, which enables us to reduce this to:

$(Contents(u) = \langle a \rangle) \Rightarrow (\forall ?Contents: Location \rightarrow Str(Object),$
(if $u \neq u$ **then** $(Contents(u)^R \circ Contents(u) = Contents(u)^R \circ Contents(u)$ **and**
 $|?Contents(u)| = |Contents(u)| - 1)$ **and**
(if $u = u$ **then** $Contents(u) = Contents(u))$ **and**
 $(\forall r: Stack, \text{if } r \neq u \text{ and } r \neq u \text{ then } ?Contents(r) = Contents(r))$ **implies**
 $?Contents(u) = \langle a \rangle;$

Since $u = u$ we can further reduce this to:

$(Contents(u) = \langle a \rangle) \Rightarrow (\forall ?Contents: Location \rightarrow Str(Object),$
 $?Contents(u) = Contents(u)$ **and**
 $(\forall r: Stack, \text{if } r \neq u \text{ and } r \neq u \text{ then } ?Contents(r) = Contents(r))$ **implies**
 $?Contents(u) = \langle a \rangle;$

Substituting $\langle a \rangle$ for $Contents(u)$ yields:

$\forall ?\text{Contents}: \text{Location} \rightarrow \text{Str}(\text{Object}), ?\text{Contents}(u) = \langle a \rangle$ **and**
 $(\forall r: \text{Stack}, \text{if } r \neq u \text{ and } r \neq u \text{ then } ?\text{Contents}(r) = \text{Contents}(r))$ **implies**
 $?\text{Contents}(u) = \langle a \rangle;$

Which is true since $?\text{Contents}(u) = \langle a \rangle$ is on the left hand side of the implication.

Proof #4: A proof of correctness for `transferTop` using the specification in Figure 8, the implementation in Figure 4, the specification for `Stack` given in `Ref_Val_Stack_Template` (Figure 7), and the call-by-reference-copy approach to parameter passing.

We need to prove:

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

By the proof rule for procedure body declaration (Figure 16) this reduces to:

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

By the proof rule for procedure call (Figure 20) for `push`, reference assignment, and variable declaration this reduces to:

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

Using the fact that $?t_s = t$, we can reduce this to:

```

{pop}\
assume |Contents(s)| > 0;
Object x;
x = s.pop();
confirm  $\forall ?\text{Contents}: \text{Location} \rightarrow \text{Str}(\text{Object}),$ 
  ( $?\text{Contents}(t) = \langle x \rangle \circ \text{Contents}(t)$ ) and
   $\forall r: \text{Stack}, \text{if } r \neq t \text{ then } ?\text{Contents}(r) = \text{Contents}(r)$ ) implies
  ( $s = \#s$  and  $t = \#t$  and
  (if  $s \neq t$  then  $(?\text{Contents}(s))^R \circ ?\text{Contents}(t) = \#\text{Contents}(s)^R \circ \#\text{Contents}(t)$ ) and
   $|\text{Contents}(s)| = |\#\text{Contents}(s)| - 1$ ) and
  (if  $s = t$  then  $?\text{Contents}(s) = \#\text{Contents}(s)$ ) and
  ( $\forall r: \text{Stack}, \text{if } r \neq s$  and  $r \neq t$  then  $?\text{Contents}(r) = \#\text{Contents}(r)$ ));

```

We need to apply a proof rule for function assignment that handles mathematical variables. The rule is given in Figure 21.

```

Context\ assertive_code; T1 %ax; T2 %bf; %ax ← a; %bf ← b;
confirm pre [x ← %ax] and
 $\forall ?a_x: T1, \forall ?b_f: T2, \forall ?G: T3, \text{post}[\#x \leftarrow \%a_x, x \leftarrow ?a_x, f() \leftarrow ?b_f, \#G \leftarrow G, G \leftarrow ?G]$ 
 $\Rightarrow Q'[b \leftarrow ?b_f][a \leftarrow ?a_x][G \leftarrow ?G];$ 


---


Context\ assertive_code; b ← f(a); confirm Q;
  where Q' = Q with substitutions for %ax, %bf, ?ax, ?bf, and ?G to avoid name conflicts, and
  where f() denotes the returned value of f.

```

Figure 21. A general proof rule template for verification of function calls

To avoid naming conflicts, we rename $?\text{Contents}$ as $??\text{Contents}$ before applying the rule:

```

{pop}\
assume |Contents(s)| > 0;
Object x;
x = s.pop();
confirm  $\forall ??\text{Contents}: \text{Location} \rightarrow \text{Str}(\text{Object}),$ 
  ( $??\text{Contents}(t) = \langle x \rangle \circ \text{Contents}(t)$ ) and
   $\forall r: \text{Stack}, \text{if } r \neq t \text{ then } ??\text{Contents}(r) = \text{Contents}(r)$ ) implies
  ( $s = \#s$  and  $t = \#t$  and
  (if  $s \neq t$  then  $(??\text{Contents}(s))^R \circ ??\text{Contents}(t) = \#\text{Contents}(s)^R \circ \#\text{Contents}(t)$ ) and
   $|\text{Contents}(s)| = |\#\text{Contents}(s)| - 1$ ) and
  (if  $s = t$  then  $??\text{Contents}(s) = \#\text{Contents}(s)$ ) and
  ( $\forall r: \text{Stack}, \text{if } r \neq s$  and  $r \neq t$  then  $??\text{Contents}(r) = \#\text{Contents}(r)$ ));

```

By the proof rule for function assignment this reduces to:

assume $|\text{Contents}(s)| > 0$;
 Object x ;
 Stack $\%s_s$; Object $\%x_{\text{pop}}$;
 $\%s_s = s$; $\%x_{\text{pop}} = x$;
confirm $\text{Contents}(s) \neq \Lambda$ **and** $(\forall ?s_s: \text{Stack}, \forall ?x_{\text{pop}}: \text{Object}, \forall ?\text{Contents}: \text{Location} \rightarrow \text{Str}(\text{Object}),$
 $?s_s = \%s_s$ **and** $\#\text{Contents}(?s_s) = \langle ?x_{\text{pop}} \rangle \circ ?\text{Contents}(?s_s)$ **and**
 $(\forall r: \text{Stack}, \text{if } r \neq ?s_s \text{ then } ?\text{Contents}(r) = \#\text{Contents}(r))$) **implies**
 $(\forall ??\text{Contents}: \text{Location} \rightarrow \text{Str}(\text{Object}),$
 $(??\text{Contents}(t) = \langle ?x_{\text{pop}} \rangle \circ ?\text{Contents}(t)$ **and**
 $\forall r: \text{Stack}, \text{if } r \neq t \text{ then } ??\text{Contents}(r) = ?\text{Contents}(r))$ **implies**
 $(?s_s = \#s$ **and** $t = \#t$ **and**
 $(\text{if } ?s_s \neq t \text{ then } (??\text{Contents}(?s_s)^R \circ ??\text{Contents}(t) = \#\text{Contents}(?s_s)^R \circ \#\text{Contents}(t)$ **and**
 $|??\text{Contents}(?s_s)| = |\#\text{Contents}(?s_s)| - 1))$ **and**
 $(\text{if } ?s_s = t \text{ then } ??\text{Contents}(?s_s) = \#\text{Contents}(?s_s))$ **and**
 $(\forall r: \text{Stack}, \text{if } r \neq ?s_s \text{ and } r \neq t \text{ then } ??\text{Contents}(r) = \#\text{Contents}(r))$);

By the proof rule for reference assignment and variable declaration, this reduces to:

assume $|\text{Contents}(s)| > 0$;
confirm $\text{Contents}(s) \neq \Lambda$ **and** $(\forall ?s_s: \text{Stack}, \forall ?x_{\text{pop}}: \text{Object}, \forall ?\text{Contents}: \text{Location} \rightarrow \text{Str}(\text{Object}),$
 $?s_s = s$ **and** $\#\text{Contents}(?s_s) = \langle ?x_{\text{pop}} \rangle \circ ?\text{Contents}(?s_s)$ **and**
 $(\forall r: \text{Stack}, \text{if } r \neq ?s_s \text{ then } ?\text{Contents}(r) = \#\text{Contents}(r))$) **implies**
 $(\forall ??\text{Contents}: \text{Location} \rightarrow \text{Str}(\text{Object}),$
 $(??\text{Contents}(t) = \langle ?x_{\text{pop}} \rangle \circ ?\text{Contents}(t)$ **and**
 $\forall r: \text{Stack}, \text{if } r \neq t \text{ then } ??\text{Contents}(r) = ?\text{Contents}(r))$ **implies**
 $(?s_s = \#s$ **and** $t = \#t$ **and**
 $(\text{if } ?s_s \neq t \text{ then } (??\text{Contents}(?s_s)^R \circ ??\text{Contents}(t) = \#\text{Contents}(?s_s)^R \circ \#\text{Contents}(t)$ **and**
 $|??\text{Contents}(?s_s)| = |\#\text{Contents}(?s_s)| - 1))$ **and**
 $(\text{if } ?s_s = t \text{ then } ??\text{Contents}(?s_s) = \#\text{Contents}(?s_s))$ **and**
 $(\forall r: \text{Stack}, \text{if } r \neq ?s_s \text{ and } r \neq t \text{ then } ??\text{Contents}(r) = \#\text{Contents}(r))$);

Using the fact that $?s_s = s$ simplifies the assertive code to:

assume $|\text{Contents}(s)| > 0$;
confirm $\text{Contents}(s) \neq \Lambda$ **and** $(\forall ?x_{\text{pop}}: \text{Object}, \forall ?\text{Contents}: \text{Location} \rightarrow \text{Str}(\text{Object}),$
 $\#\text{Contents}(s) = \langle ?x_{\text{pop}} \rangle \circ ?\text{Contents}(s)$ **and**
 $(\forall r: \text{Stack}, \text{if } r \neq s \text{ then } ?\text{Contents}(r) = \#\text{Contents}(r))$) **implies**
 $(\forall ??\text{Contents}: \text{Location} \rightarrow \text{Str}(\text{Object}),$
 $(??\text{Contents}(t) = \langle ?x_{\text{pop}} \rangle \circ ?\text{Contents}(t)$ **and**
 $\forall r: \text{Stack}, \text{if } r \neq t \text{ then } ??\text{Contents}(r) = ?\text{Contents}(r))$ **implies**
 $(s = \#s$ **and** $t = \#t$ **and**
 $(\text{if } s \neq t \text{ then } (??\text{Contents}(s)^R \circ ??\text{Contents}(t) = \#\text{Contents}(s)^R \circ \#\text{Contents}(t)$ **and**
 $|??\text{Contents}(s)| = |\#\text{Contents}(s)| - 1))$ **and**
 $(\text{if } s = t \text{ then } ??\text{Contents}(s) = \#\text{Contents}(s))$ **and**
 $(\forall r: \text{Stack}, \text{if } r \neq s \text{ and } r \neq t \text{ then } ??\text{Contents}(r) = \#\text{Contents}(r))$);

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

$|Contents(s)| > 0 \Rightarrow Contents(s) \neq \Lambda$ **and** $(\forall ?x_{pop}: Object, \forall ?Contents: Location \rightarrow Str(Object),$
 $Contents(s) = \langle ?x_{pop} \rangle \circ ?Contents(s)$ **and**
 $(\forall r: Stack, \text{if } r \neq s \text{ then } ?Contents(r) = Contents(r)))$ **implies**
 $(\forall ??Contents: Location \rightarrow Str(Object),$
 $(??Contents(t) = \langle ?x_{pop} \rangle \circ ?Contents(t)$ **and**
 $\forall r: Stack, \text{if } r \neq t \text{ then } ??Contents(r) = ?Contents(r))$ **implies**
 $(s = s$ **and** $t = t$ **and**
 $(\text{if } s \neq t \text{ then } (??Contents(s))^R \circ ??Contents(t) = Contents(s)^R \circ Contents(t)$ **and**
 $|??Contents(s)| = |Contents(s)| - 1))$ **and**
 $(\text{if } s = t \text{ then } ??Contents(s) = Contents(s))$ **and**
 $(\forall r: Stack, \text{if } r \neq s \text{ and } r \neq t \text{ then } ??Contents(r) = Contents(r)))$);

This complex expression has the form:

$\forall s, t: Stack, \forall ?x_{pop}: Object, \forall ?Contents, ??Contents: Location \rightarrow Str(Object),$
 $|Contents(s)| > 0 \Rightarrow Contents(s) \neq \Lambda$ **and**
 $(\text{first_antecedent implies (second_antecedent implies consequent)});$

Therefore, it suffices to prove $\forall s, t: Stack, \forall ?x_{pop}: Object,$ and $\forall ?Contents, ??Contents: Location \rightarrow Str(Object),$ that:

1. $|Contents(s)| > 0 \Rightarrow Contents(s) \neq \Lambda$
2. $|Contents(s)| > 0$ **and** *first_antecedent* **and** *second_antecedent* \Rightarrow *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** $s \neq t$ **then** $(??Contents(s))^R \circ ??Contents(t) = Contents(s)^R \circ Contents(t)$ **and**
 $|??Contents(s)| = |Contents(s)| - 1$
3. **if** $s = t$ **then** $??Contents(s) = Contents(s)$
4. $\forall r: Stack, \text{if } r \neq s$ **and** $r \neq t$ **then** $??Contents(r) = Contents(r)$

Assertion (1) is trivially true.

To deal with assertion (2), assume that $s \neq t$. Using the string equalities from the *first_antecedent* and the *second_antecedent* we can simplify the string equality in assertion (2) as follows:

$??Contents(s)^R \circ ??Contents(t) = Contents(s)^R \circ Contents(t)$
apply string equality from second_antecedent
 $??Contents(s)^R \circ \langle ?x_{pop} \rangle \circ ?Contents(t) = Contents(s)^R \circ Contents(t)$
apply string equality from first_antecedent
 $??Contents(s)^R \circ \langle ?x_{pop} \rangle \circ ?Contents(t) = (\langle ?x_{pop} \rangle \circ ?Contents(s))^R \circ Contents(t)$
apply string manipulation
 $??Contents(s)^R \circ \langle ?x_{pop} \rangle \circ ?Contents(t) = ?Contents(s)^R \circ \langle ?x_{pop} \rangle \circ Contents(t)$

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

$$|??\text{Contents}(s)| = |\text{Contents}(s)| - 1$$

apply string equality from first_antecedent

$$|??\text{Contents}(s)| = |\langle ?x_{\text{pop}} \rangle \circ ?\text{Contents}(s)| - 1$$

apply string properties

$$|??\text{Contents}(s)| = 1 + |?\text{Contents}(s)| - 1$$

Again, the frame property in the *first_antecedent* allows us to conclude that $??\text{Contents}(s) = ?\text{Contents}(s)$. 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*.