Preserving Abstraction for Operation Invocations with Repeated Arguments

Greg Kulczycki, Murali Sitaraman, William F. Ogden, and Bruce W. Weide

Technical Report RSRG-08-03
School of Computing
100 McAdams
Clemson University
Clemson, SC 29634-0974 USA

May 2008

Preserving Abstraction for Operation Invocations with Repeated Arguments

Gregory W. Kulczycki
Computer Science, Virginia Tech, Falls Church, VA 22043 USA
gregwk@vt.edu

Murali Sitaraman
Computer Science, Clemson University, Clemson, SC 29634 USA
murali@cs.clemson.edu

William F. Ogden and Bruce W. Weide
Computer Science and Engineering, The Ohio State University, Columbus, OH 43210 USA
ogden@cse.ohio-state.edu, weide@cse.ohio-state.edu

Abstract: When arguments of non-trivial types are repeated in an operation invocation, alternative but otherwise functionally equivalent implementations often produce different results. To know what outcome to expect, a programmer is forced to examine the procedure body of the implementation currently in use. This clearly violates procedural abstraction and thwarts modular reasoning. It occurs because aliasing is introduced when repeated arguments are passed to procedures by copying references. A simple parameter-passing scheme in which initial values are passed for second and subsequent repeated arguments eliminates this problem. This scheme can be implemented efficiently, and it interacts well with by-value parameter passing. It localizes the effects of calls, and permits the introduction of a uniform proof rule for all procedure calls, simplifying formal modular analysis as well as informal yet rigorous reasoning that respects procedural abstraction.

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

1. Introduction

All popular programming languages allow arguments to be repeated in calls. A repeated-argument instance is either explicit, as in the call `P(U, U)`, or implicit, as in the call `P(A[i], A[j])` where `A` is an array and `i` equals `j`, and the call `P(G)` where `G` is a global variable modified within the body of `P`. Instances of repeated arguments in calls cannot be reliably detected at compile time, so such calls, and their untoward consequences, have become a standard "feature" of nearly all imperative programming languages.

The most serious complication arising from calls with repeating arguments is the violation of procedural abstraction: even when two different implementations of an operation always yield the same results for independent arguments, they can easily yield different results when arguments are repeated. This means that the effects of a call with repeated arguments cannot be determined without examining the body of the called operation, which directly violates the principle of procedural abstraction. As a result, it thwarts the ability of developers to reason, formally or informally, about a system in a modular fashion using only the interface descriptions for operations.

The violation of procedural abstraction caused by repeated arguments might be viewed as a special instance of the general unresolved aliasing problem that currently exists in all popular imperative languages. It is widely recognized that aliasing seriously impairs the intuitive comprehension as well as the formal specification and verification of software [5][12][24]. Previous research on eliminating or controlling aliasing has quite sensibly focused on its most common cause – the aliasing of implicit or explicit pointers through reference assignments of mutable objects [3][10][23]. However, the aliasing caused by repeated arguments is independent of such aliasing in that it persists even when a language designer or a disciplined software engineer has avoided explicit assignment of references everywhere else.

Repeated argument aliasing arises because, for efficiency reasons, non-trivial objects are passed by copying references. When arguments are then repeated in a call, their formal parameters become aliased within the body of the procedure. This happens in every well-known imperative language from FORTRAN I – which included no reference variables or reference assignments – to Java,

because they all pass certain arguments by copying references, and they all allow repeated arguments. The repeated argument aliasing problem is both pervasive and separable from the general reference assignment aliasing problem, so we shall subsequently focus exclusively on solving it.

This paper introduces a new parameter-passing scheme that preserves procedural abstraction in the presence of calls with repeated arguments. A key language design objective is to make the scheme efficient, yet simple, so that it permits all traditional calls yet gives them uniform semantics. A key software engineering objective is that, for potential repeated-argument calls, the scheme should require neither special-case assertions from software specifiers nor special-case implementations from programmers.

Section 2 presents a simple example of a call with repeated arguments that shows how traditional parameter passing breaks procedural abstraction. Section 3 introduces the new parameter-passing scheme for eliminating the problem. It describes an efficient implementation and explains how the scheme interacts appropriately with parameters passed by value. The scheme makes it possible to introduce in Section 4 a single proof rule for all procedure calls with the virtue that calls without repeated arguments have their traditional effects, yet calls with repeated arguments do not introduce aliasing and hence do not thwart procedural abstraction and modular reasoning. Section 5 contains a summary of related work and our conclusions.

2. How Repeated Arguments Break Procedural Abstraction

To see clearly how calls with repeated arguments make otherwise functionally equivalent implementations behave differently and violate procedural abstraction, consider an example of a transform operation on matrices (two-dimensional arrays) implemented in a typical language. The operation's signature is `Transform_by(C, X: Matrix)`, and its abstract behavior is described informally as: "Given square matrices `C` and `X` of equal size, the `Transform_by` operation multiplies coefficient matrix `C` times matrix `X` and returns the result in `X`. It leaves the value of `C` unchanged."

Consider two different implementations for the `Transform_by` operation. One implementation is given in Figure 1 using the particular notation of RESOLVE [21]. It stores intermediate results in a

temporary variable XCol and then uses this to overwrite the columns of X. An alternative implementation uses the same code as the first, except that the loops run from n down to 1 rather than 1 up to n. Both implementations yield the same correct result when Transform_by is called on any two distinct matrices, as in a call Transform_by(K, M). However, if you repeat the coefficient matrix, as in the call Transform_by(K, K) with $K = \begin{bmatrix} 1 & 2 \\ 3 & 4 \end{bmatrix}$, the implementation in Figure 1 yields $K = \begin{bmatrix} 7 & 22 \\ 15 & 46 \end{bmatrix}$, while the alternative implementation yields $K = \begin{bmatrix} 31 & 10 \\ 69 & 22 \end{bmatrix}$. Since in repeated argument cases the value of K differs depending on the implementation, the particular operation body must be consulted to determine which result to expect, and procedural abstraction is violated.

```
Procedure Transform_by(C, X: Matrix);
    Var i, j, k: Integer;
    Var n := Size(C);
    Var XCol: Array 1..n of Integer;
    For j := 1 to n do
        For i := 1 to n do
            XCol(i) := C(i, 1)*X(1, j);
            For k := 2 to n do
                XCol(i) := XCol(i) + C(i, k)*X(k, j);
            end;
        end;
        For i := 1 to n do
            X(i, j) := XCol(i);
        end;
    end;
end Transform_by;
```

Figure 1. An implementation of the Transform_by operation with increasing for-loops

This example applies to any traditional imperative language, since they all pass arrays by copying references. It demonstrates that the traditional parameter passing mechanism is plainly broken for repeated arguments.

3. An Initialization Based Scheme for Parameter Passing

A simple solution to the problem raised by repeated arguments in operation invocations is to use initialized variables for second and subsequent arguments that are repeated. With such a scheme, the effective semantics for an invocation with repetitions such as `P(U, U, U)` is the same as that for the repetition-free "`Temp1: T; Temp2: T; P(U, Temp1, Temp2);`" where `T` is the type of `U`, and `Temp1: T, Temp2: T` denotes the declaration and initialization of variables of type `T`. Similarly, if `F` is a function, `F(U, U, U)` returns the result of `F(U, Temp1, Temp2)`. The effect of this scheme is that the value of a repeated argument after a call is the value the *first* parameter would have, provided initial values were supplied for the second and subsequent repeated arguments.

Consider the effect of this scheme in the call `Transform_by(K, K)`. This is equivalent to `Transform_by(K, Temp)` where `Temp` is a Matrix having an initial value. In this case, the coefficient matrix `K` simply remains unchanged. The scheme always produces consistent, predictable behavior, but in general, it won't always be quite so benign.

A less repeated-argument friendly design for a transform operation might put the coefficient matrix second, as in `Transform(X, C: Matrix)`. The operation would then transform the first parameter `X` to be `C` times `X`. As before, a call `Transform_by(K, K)` is equivalent to the call `Transform(K, Temp)` where `Temp` is an initial-valued Matrix. If the initial value for type Matrix were the identity matrix, then *all* correct implementations of the `Transform_by` operation would still preserve the value of `K`. This effect can be predicted by consulting the specification because the product of `K` and the identity matrix is `K`. If the initial value for type Matrix were the zero matrix, then `K` would become the zero matrix (the product of `K` and the zero matrix) for all implementations. If Matrix type objects were allowed a range of initial values, then the range of possible results for `K` would be dictated by that initial value range.

The same scheme of using initialized variables works for other repeated argument situations. Consider an operation `Transform_by_G(X: Matrix)` that transforms `X` by some global matrix `G`. A call `Transform_by_G(G)` would result in the coefficient Matrix `G` remaining unchanged. This is because the scheme treats any global variables as fixed first parameters and consequently views a global used as an actual parameter as an argument repetition, rendering the effect of this call to be

"Temp: Matrix; Transform_by_G(Temp);". For a record R, a call P(R, R.Field3) has the same effect as "Temp: T_{Field3}; P(R, Temp);", i.e., the effect of the call on the earlier record type parameter dominates that on the later field parameter. If the parameters were reversed, the effect on the designated field would override any modifications to the remainder of the record. Passing an array and one of its elements as arguments produces analogous effects.

3.1. Efficient Implementation

Although we want to simplify reasoning about programs by keeping calls from introducing references into the programmer's world, a compiler can still use a modified version of parameter passing by reference copying to realize the proposed scheme efficiently, while minimizing initialization overhead – as long as this usage remains completely hidden within the language implementation. If a compiler can detect statically that a call does not involve repeated arguments, then as usual it can generate code for passing parameters by copying references because there is no possibility of introducing aliasing. Examples include calls such as P(U) where U is *not* a global variable in the scope of P and calls of the form P(U, V). When repeated arguments are detected statically, as in the calls P(U, U) or P(A, A[i]), the compiler can generate code for "Temp: T; P(U, Temp);" or "Temp: T; P(A, Temp);", respectively.

When repeated arguments are uncertain, as in the call P(A[i], A[j]), the compiler can use the following technique. To marshal each potentially repeated actual parameter, it generates code first to copy the reference value of the argument to the parameter stack and then to replace the prior reference with a special invalid address. For example, suppose that A[i] and A[j] are array-variable arguments in the invocation of an operation with signature P(X, Y: T). The compiler generates code to copy the reference in location A[i] to the parameter stack and to replace that reference in location A[i] with a special invalid address. For the subsequent argument A[j], it generates code to check for that special invalid address. When j equals i, this check succeeds and signals that a repeated argument has been encountered. Accordingly, the generated code creates an initial value of the appropriate type and places a reference to its location on the parameter stack together with a special invalid address as the location to which it is to be returned. Upon return, parameters with the special invalid address as their designated return location are simply finalized. Figure 2 illustrates this process when $i \neq j$ and $i = j$.

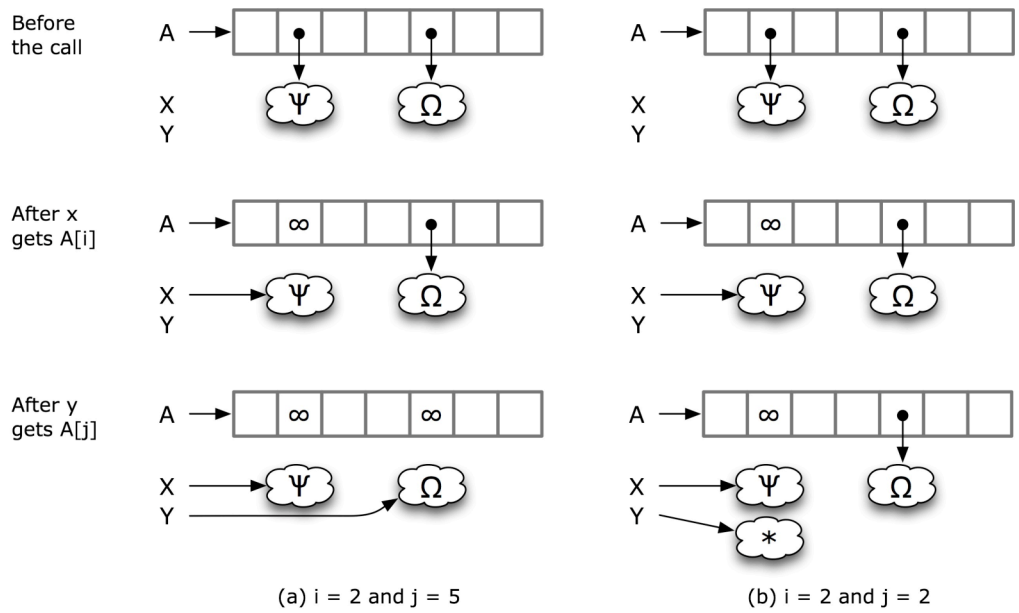

Figure 2. Representations of the program state during parameter passing in the call P(A[i], A[j]) when (a) i ≠ j (b) i = j. Variables X and Y are the formal parameters of P, ∞ is a special invalid address, and Ψ and Ω symbolize objects in the array. * symbolizes the initialized object created when ∞ is recognized as invalid.

In the implementation outlined above, initialization overhead for the larger objects typically found in the heap is incurred only for truly repeated arguments. A compiler might further defer initialization until an object is actually accessed, thereby avoiding the initialization cost entirely if a repeated argument happens not to be used in a given call. However, any optimization that avoids initializations is only employable on objects for which initialization induces no side-effects.

3.2. Combining the Initialization Scheme with Parameter Passing by Value

The original by-value mode for passing parameters was well known for not introducing aliasing and not introducing problems for modular reasoning. This happened because it worked by creating a new object to pass whose value was that produced by the functional expression that was the actual parameter. When the actual parameter was a simple variable U, the effective functional expression used was Clone(U), which created a deep copy of U's value. So as long as a cloning operation was available for the object type and its performance parameters were acceptably low, the aliasing problem for repeated by-value arguments was eliminated.

For performance reasons, operations on non-trivially complex objects frequently involve updates to the parameter objects, so our scheme for preventing aliasing must be employed, and it must interact appropriately with other parameters being passed in a by-value mode. The potential difficulty arises when an object passed by reference copying might be a repeated argument and that object is also involved in some of the expressions for by-value mode parameters. Clearly, the by-value mode arguments must be evaluated before the initialization scheme is employed on other parameters. In the Transform_by operation example, if the coefficient matrix parameter C were given by-value mode, then the repeated argument call Transform_by(U, U) would effectively become Transform_by(Clone(U), U) and the resulting matrix U would have #U * #U as its value, where #U is the prior value of U.

4. Benefits for Formal Specification and Reasoning

This section presents formal proof rules for parameter passing that incorporate the initialization scheme. This ensures, for example, that the code in Figure 1 can be proved to be correct with respect to a formal specification, such as the one below, whether or not arguments might be repeated.

```
Operation Transform_by(C, X: Matrix);
    ensures C = #C and X = C * #X;
```

Symbols #C and #X denote the incoming values of C and X respectively.

Now we are ready to discuss a general specification-based proof rule for handling operation invocations as shown in Figure 3. Without loss of generality, we consider a call with only two potentially repeated arguments. The Context within which this rule is to be applied must include the specification of the called operation, P, as shown. The notation assertive_code is a placeholder for all 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 in order for an assertion Q to be confirmed true after the call.

Context \ assertive_code; $\%U_X$: T1; $\%V_Y$: T2; $\%U_X$ <– U; $\%V_Y$ <– V;

Confirm pre[X⇝$\%U_X$, Y⇝$\%V_Y$] **and** (∀ $?U_X$: T1, ∀ $?V_Y$: T2,

post[#X⇝$\%U_X$, X⇝$?U_X$, #Y⇝$\%V_Y$, Y⇝$?V_Y$] ⇒ Q'[U⇝$?U_X$][V⇝$?V_Y$]);

Context \ assertive_code; P(U, V); **Confirm** Q;

where (**Operation** P(X: T1; Y: T2); **requires** pre; **ensures** post;) ∈ Context and Q' is Q with substitutions for $\%U_X$, $\%V_Y$, $?U_X$, and $?V_Y$ to avoid name conflicts.

Figure 3. A general proof rule for verification of operation invocations

The rule introduces two local verification variables, specially named $\%U_X$ and $\%V_Y$, to which the values of the corresponding actual arguments are transferred. The names U and V have been subscripted with names of the formal parameters X and Y to avoid naming conflicts in the repeated argument case P(U, U), which is important for an automated verification system. The hypothesis of the rule uses the initializing transfer operation V <– W, which gives the value of W to V and gives W an initial value of its type, to give appropriate values to $\%U_X$ and $\%V_Y$. The transfers allow the traditional invocation hypothesis to be applied in the remainder of the rule, as if P were being called with guaranteeably distinct parameters. When the arguments are in fact not repeated, the transfers make no difference. With a repeated argument call P(U, U), the second argument will effectively be the desired initial value because of the sequence $\%U_X$ <– U; $\%U_Y$ <– U.

The rule uses the notation ⇝ for logical substitution, and it requires two conjuncts to be proved. First, the precondition pre of P needs to be proved, after replacing the formals X and Y with the actuals. Next, the assertion Q needs to be confirmed, assuming that the postcondition post of P holds (with proper substitutions). Since the specification of P may be relational, any of a number of post-state values may result for the same input values. So the second conjunct states that as long as the post-state values of the arguments satisfy the post-condition, Q must hold. The formal output parameter names $?U_X$ and $?V_Y$ denote possible post-state values; they must replace actual arguments U and V before Q is confirmed so that the names used in the two sides of the implication are consistent. The ordering of the substitutions in Q reflects the fact that the value of the first formal parameter is used when arguments are repeated; in other words, the substitutions take place sequentially in two left to right steps ([ ], [ ]), not in parallel ([,]). If arguments are repeated and Q just involves U, for example, then only the output value of the first parameter $?U_X$ matters.

Context \ assertive_code; **Confirm** ∀ ?V: T, (T.**Is_Initial**(?V) ⇒ Q'[V⇝?V][U⇝V]);

Context \ assertive_code; U <– V; **Confirm** Q;

where Q' is Q with substitution for ?V to avoid name conflicts.

Figure 4. A proof rule for the initializing transfer operator

Because the proof rule for operation invocations introduces the initializing transfer operator, a proof rule for it is needed, and this is given in Figure 4. Here, T.**Is_Initial**(X) is a predicate that tells whether its argument X of type T has an initial value for the type T. We use a predicate instead of asserting, for example, M = Matrix.**Initial_Value** because a constructor operation may provide any one of multiple possible initial values. If the **ensures** clause of the constructor is omitted for a type T, then an initial object of that type is allowed to have any abstract value and the predicate T.**Is_Initial**(X) is always true. While this may not be desirable in most cases, it does not cause any difficulties for the formal system. Since we assume it is possible to initialize variables of any type using a declaration such as Temp: T, every type must have a default public constructor having no arguments – a practice widely followed in component design.

Using the rules given in this section, a proof of correctness for the implementation of a TransferTop operation on Stack objects, and a proof for a piece of code that calls TransferTop with repeated arguments, are given in [14]. Those examples illustrate the use of the same operation invocation rule for both the common calling situation, in which no arguments are repeated, and the case when arguments are repeated.

5. Related Work and Conclusions

As early as 1978, Cook [4] noted that permitting procedure calls with repeated arguments renders Hoare logic unsound. The vast body of literature on aliasing and its software engineering consequences are summarized in [12][14][24]. While some have sought to avoid aliasing all together, several others have concentrated on merely constraining the aliasing that can result from reference assignment. Systems involving unique references – in which there is only one reference to each object – often form the basis of such research [3]. They allow unique references to be "temporarily passed to methods without being consumed," using a borrowing mechanism ([3],

p. 1). Unfortunately, borrowing violates uniqueness and thus necessitates using global stores to capture its semantics.

Some previous treatments of parameter aliasing preclude indexed array variables and global variables as arguments to calls, and thus eliminate repeated arguments syntactically [10]; others propose to introduce a less restrictive (but more expensive) preclusion strategy and eliminate them semantically [17]. Whereas procedure call rules in [4] and [6] assume no argument repetition, some rules that allow repeated arguments introduce references to handle them [2]. Gries introduces a proof rule that handles repeated arguments in [8], but it requires a correctness proof for a version of the procedure in which potentially aliased formal parameters have the same identifier. For example, a repeated argument call to Transform_by would require that the procedure body be proved correct with respect to its specification if the formal parameter `X` is replaced everywhere by `C`. Explicit specification of aliasing for operations with potential repeated arguments is another topic that has received attention [16]. Crank and Felleisen compare formal semantics of alternative parameter passing techniques from a reasoning perspective, including parameter passing by reference and value [5], and they conclude (p.10), "using call-by-value … seems the most attractive choice." Their conclusion is consistent with the motivation for avoiding the aliasing caused by repeated arguments in this paper. However, by having the initialization scheme as the default approach and by-value parameter passing as an available option, the approach presented here provides a solution that avoids aliasing but is also efficient and much more general.

We have presented a way of resolving the problem of providing procedural abstraction for all operation invocations, while permitting calls with repeated arguments. Different implementations produce identical and predictable behaviors under the proposed scheme. The formal reasoning benefits of our approach within the context of a clean semantics are documented in [14], where formal proofs of calls with repeated arguments using both the initialization scheme and the conventional reference-copying scheme are presented. Results from an experimentation with the ideas in a Java-like language are given in [13]. The simple scheme we have proposed solves a subtle, yet long-standing obstacle to parameter alias avoidance and prevention. When it is integrated with techniques for avoiding other sources of aliasing among mutable objects, it can lead

to languages that support procedural abstraction – and hence, modular reasoning – in all cases, and can make it easier to specify, develop, and maintain correct software.

Acknowledgments

This research was funded in part by the National Science Foundation grant CCR-0113181 and DMS-0701187. Our special thanks go to Gary Leavens for his insightful comments in the early stages of the development of this paper. We would also like to thank members of our research groups for their suggestions.

References

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

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

[3] D.G. Clarke and T. Wrigstad, "External Uniqueness," *Procs. 10th International Workshop on Foundations of Object-Oriented Languages,* New Orleans, LA, January 2003; available at: http://www.cis.upenn.edu/~bcpierce/FOOL/FOOL10.html.

[4] S.A. Cook, "Soundness and Completeness of an Axiom System for Program Verification," *SIAM Journal of Computing 7(1)*, 1978, pp. 70-90.

[5] E. Crank and M. Felleisen, "Parameter Passing and the Lambda Calculus," *Procs. 18th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, 1991, pp. 233-244.

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

[7] G.W. Ernst, R.J. Hookway, and W.F. Ogden, "Modular Verification of Data Abstractions with Shared Realizations," *IEEE Transactions on Software Engineering 20(4)*, 1994, pp. 288-207.

[8] D. Gries and G. Levin, "Assignment and Procedure Call Proof Rules," *ACM Transactions on Programming Languages and Systems 2(4)*, 1980, pp. 564-579.

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

[10] D.E. Harms and B.W. Weide, "Copying and Swapping: Influences on the Design of Reusable Software Components," *IEEE Transactions on Software Engineering 17 (5)*, pp. 424-435.

[11] C.A.R. Hoare and J. Misra: "Verified Software: Theories, Tools and Experiments," *IFIP Working Conference on Verified Software: Theories, Tools and Experiments (VSTTE)*. Zurich, 2005. http://vstte.ethz.ch/

[12] J. Hogg, D. Lea, A. Wills, D. deChampeaux, and R. Holt, "The Geneva Convention on the Treatment of Object Aliasing," *OOPS Messenger 3(2)*, pp. 11-16.

[13] G. Kulczycki and J. Vasudeo: "Simplifying Reasoning about Objects with Tako," *Proceedings of the Specification and Verification of Component-Based Systems Workshop*. Portland, OR, 2006.

[14] G. Kulczycki, *Direct Reasoning*, Ph. D. Dissertation, Department of Computer Science, Clemson University, Clemson, SC 29634-0974, May 2004.

[15] 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, pp. 175-188.

[16] G.T. Leavens, O. *Antropova: ACL – Eliminating Parameter Aliasing with Dynamic Dispatch*. Tech Report 98-08a, Department of Computer Science, Iowa State University, 1998.

[17] 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 10 (1)*, 1978, pp. 1-26.

[18] B. Meyer, "On to components," *IEEE Computer 32(1)*, 1999, pp. 139-140.

[19] P. Müller and A. Poetzsch-Heffter, "Modular Specification and Verification Techniques for Object-Oriented Software Components," *Foundations of Component-Based Systems*, eds. G.T. Leavens and M. Sitaraman, Cambridge University Press, 2000, 137-159.

[20] P. O'Hearn, J. Reynolds, and H. Yang, "Local Reasoning about Programs that Alter Data Structures," *Procs. 15th Intl. Workshop on Computer Science Logic*, 2001, pp. 1-19.

[21] 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," *Procs. Sixth International Conference on Software Reuse – LNCS 1844*, Springer Verlag, 2000, pp. 266-283.

[22] M. Sitaraman, B. W. Weide, and W. F. Ogden, "Using Abstraction Relations to Verify Abstract Data Type Representations," *IEEE Transactions on Software Engineering 24(3)*, pp. 157-170.

[23] P. Wadler, "Linear Types Can Change the World!," *IFIP TC 2 Working Conference on Programming Concepts and Methods*, 1990, pp. 347-359.

[24] B.W. Weide and W.D. Heym, "Specification and Verification with References," *Procs. ACM OOPSLA Workshop on Specification and Verification of Component-Based Systems*, 2001; available at: http://www.cs.iastate.edu/~leavens/SAVCBS/papers-2001.