Clean Semantics for Calls with Repeated Arguments

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

Technical Report RSRG-05-01
Department of Computer Science
100 McAdams
Clemson University
Clemson, SC 29634-0974 USA

March 2005

Copyright © 2005 by the authors. All rights reserved.

Clean Semantics for Calls 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 are repeated in a procedure call, aliasing arises and plays havoc with specification and reasoning. To prevent this aliasing, which renders the semantics of a language unclean even when all other sources of aliasing are avoided, we propose a simple new scheme for parameter passing that uses initial values for passing second and subsequent repeated arguments and localizes the effects of calls.

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

1. Introduction

In most imperative languages, explicit referencing mechanisms (e.g., pointers) enjoy a prominent role because they seem to provide simple and efficient ways to solve a number of computational problems. Unfortunately, the simplicity is somewhat illusory in that unbridled copying of references creates aliasing and seriously impedes the intuitive comprehension as well as the formal specification and verification of software [5][11][12][16][20]. To capture the full range of problems related to aliasing, we introduce in Section 2 the simple notion of *clean semantics*.

When pursuing languages with clean semantics, the obvious problem is aliasing of explicit pointers arising from reference assignments for mutable objects. Fortunately, this problem is widely recognized and has been addressed in previous research. Some approaches for object-oriented languages are based on unique references, and they provide a destructive read operator as an alternative to assignment [3][19]. Others provide exchange statements or a swapping operator for data transfer [10].

This paper addresses a more subtle obstacle to clean semantics for imperative languages. This is the aliasing that arises when procedures are called with repeated arguments and parameters are passed by copying references. Such a repeated-argument instance may be obvious, as with an explicit call p(x, x). But it may also be implicit when indexed array variables or global variables are passed as arguments, as in p(a[i], a[k]) when i equals k, or in p(g) when g is a global variable modified within the body of p. Consequently, repeated argument problems aren't generally resolvable at compile time.

Within the body of a procedure that has been called with repeated arguments passed by reference copying, two or more different formal parameters become aliased. This remains an independent, unresolved aliasing problem because it does not disappear even when a language designer or a disciplined software engineer avoids explicit assignment of references. It exists in every well-known imperative language from FORTRAN I—which includes no reference variables or reference assignments—to Java, because they pass parameters by copying references and allow arguments to be repeated. As early as 1978, while considering the scenario in which a global variable is passed by reference as a parameter to a procedure that also uses it directly, Cook [4] concluded that permitting procedure calls with repeated arguments renders Hoare logic unsound.

The contribution of this paper is to introduce a new scheme for parameter passing that makes it possible to reason soundly about programs without being concerned about aliasing from repeated arguments or about loss of efficiency. Pragmatically, this eliminates a subtle source of programming errors—making the result important to both the software engineering and the programming language communities. The scheme is discussed in Section 3. The section presents an efficient implementation of the scheme and explains how it interacts appropriately with parameter passing by value. The scheme makes it possible to introduce in Section 4 a single proof rule for all procedure calls: calls without repeated arguments are unaffected, yet calls with repeated arguments do not introduce aliasing. Section 5 contains a summary of related work and our conclusions.

2. Localizing the Effects of Calls

To capture a critical characteristic that makes it comparatively easy to reason about procedures in a modular way, we introduce the intuitive, yet formalizable, notion of *clean semantics*. A semantics *S* for a programming language *L* is clean if it has the following two properties. First, it is *variable-based*: *S* presents the state of a program as just the aggregate abstract values of all the variables defined at that point in the program. Second, it is *effect-restricted*: For any invocation of any operation p, the state change prescribed by *S* does not modify the values of any variables that are not *syntactically targeted* by that invocation, as explained below.

The variable-based property is simply a requirement that the semantics *S* uses the standard notion of state space. The effect-restricted property is a requirement that the semantics of a call to p be localized to variables syntactically targeted by that invocation—a subset of the explicit parameters to the call along with the global variables in the scope of p that the code for p appears, from its syntax, to be affecting. Having a small predictable set of affected variables is what makes it

relatively easy to reason about a call. Conversely, if aliasing or some other side-effecting mechanism can cause changes to variables that aren't syntactically targeted, then reasoning becomes quite difficult because it is no longer local. To illustrate the notion of syntactically targeted variables, consider a simple example in Pascal:

```
var g, h: integer;
procedure p( var x, y: integer; z: integer );
begin
    x := x + z;
    g := h + y;
end;
```

Although there are several variables in scope in the body of p, p's code syntactically targets only its first formal parameter x and the global variable g for modification[1]. Therefore, the effect of an invocation p(a, b, c+17) must be restricted to the corresponding actual parameter a and the global g. If a variable other than a or g is modified in any way, then the call to p is not effect-restricted. In general, the code for an operation such as p may involve calls to other operations, and therefore, the formal definition of syntactically targeted variables involves inductively aggregating all the non-local variables targeted by all called operations. For the built-in primitive operations (such as the integer assignments used here), what's syntactically targeted is obvious.

2.1. Data Abstraction

To illustrate the ideas in the presence of data abstraction, suppose that the state space of a calling program involving stacks of graphs includes stack variables s, t and graph variable x, as in Figure 1.

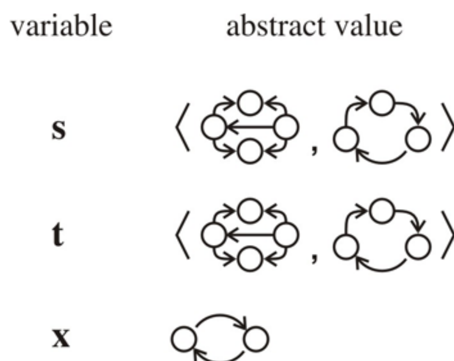

Figure 1. A state space in a program containing graphs and stacks of graphs

In the figure, s and t denote two distinct stack-of-graph objects that happen to have the same value, so the effect of a call t.push(x) would be restricted to t and x. If instead, s and t were aliased, then the same call would not be effect-restricted, since it would also modify s, which is not syntactically

[1] Here, we make this observation based on a direct examination of the body of p. However, in a language that supports specifications, the same information may be ascertained from the header of p alone.

targeted by the call. Stacks s and t might be aliased either because an explicit reference assignment occurred prior to the call to push or because push is being called within a repeated-argument context, such as when an invocation transferTop(u, u) has been made to the code in Figure 2. In this case, the call t.push(x) affects s, and the call s.pop() affects t, so neither call is effect-restricted.

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

Figure 2. A procedure that can introduce aliasing through repeated arguments

Regardless of its source, if aliasing were present, then the view of variables as denoting values in Figure 1 would lead to unsound reasoning about the code in Figure 2. The unsoundness in reasoning remains even if transferTop is never employed with repeated arguments in a calling program. This is because for modular reasoning, it is essential to know that, in all possible usages, the code in Figure 2 would satisfy its specification. If formal parameters might be repeated, then the reasoning needs to account for this possibility. The same problem arises when a global variable targeted by a procedure is passed as an argument in its call. Consider a procedure void transferGlobalTop(Stack t) that transfers the top element of a global stack g to t. A call transferGlobalTop(g) would make g and t aliases, so the internal call t.push(x) would not be effect-restricted. Here, the global variable g is effectively a fixed argument to any transferGlobalTop call, so that transferGlobalTop(g) behaves the same as transferTop(g, g).

2.2. Semantic Space of Variables

The definition of clean semantics deliberately allows the possibility of the "abstract" value of a variable being a location (reference) in a global store. Such abstractions are necessary at a minimum when one wishes to specify a component whose operations and objects capture reference behavior [12], or in other situations where software engineers want to make indirection explicit. In the extreme, however, the definition would make it possible to view all program variables as locations, and consequently have the global store be syntactically targeted by every programming operation. This possibility is illustrated in Figure 3, where the global Store is a *conceptual* variable. Using the "abstraction" here, every call becomes trivially effect-restricted. A call t.push(x) semantically affects only the global Store and not the location s, even though it is aliased with t. However, the advantages of abstraction have been lost. So while the purpose of restricting effects of a call was to localize and simplify reasoning, the introduction of a global store has led to an undesirable, opposite effect.

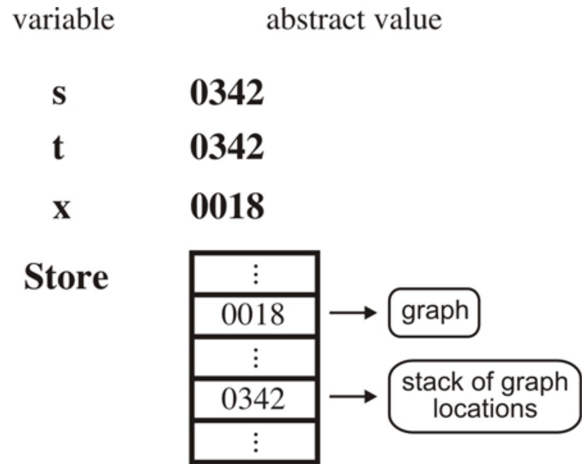

Figure 3. A less desirable state space for Stack variables

Although frame properties might be used to restrict the locations where the global store is allowed to change [1][13][15][16], languages that permit aliased references to mutable objects via whatever mechanism *require* a view of all sophisticated variables as references into a global store in order to achieve clean semantics and sound reasoning—even if the only source of aliasing is calls with repeated arguments. Whereas prior work (e.g., [15][16]) has focused on capturing such indirection with all its specificational complexity, our objective is to avoid it as much as possible. Therefore, we supplement the goal of achieving clean semantics with that of also allowing variables to take abstract values directly in appropriately rich semantic spaces (as in Figure 1), not just in restricted spaces of locations and pointers.

Using data abstraction specifications, we have shown elsewhere [7][17] that it is possible to capture the behavior of programming types such as lists and graphs using a rich space of mathematical models such as strings and sets of edges rather than references. Neither global stores nor locations are necessary to capture the effects of calls in specifications. At the same time, these types can be implemented correctly using references internally by establishing appropriate correspondences between implementations and specifications [18]. This approach limits the need for programmers to reason about references to only a few unavoidable implementations.

3. A Parameter Passing Scheme with Clean Semantics

A simple solution to the problem raised by calls with repeated arguments is to use initialized variables for second and subsequent arguments that are repeated. Operationally, this semantics for a procedure call such as p(u, u, u) is the same as the semantics for {T temp1, temp2; p(u, temp1, temp2);} where T is the type of u, and T temp1, temp2 denotes the declaration and initialization of variables of type T. Similarly, if f were a function, f(u, u, u) would return the result of f(u, temp1, temp2). With this scheme, the call transferTop(u, u) behaves as though it were {Stack temp; transferTop(u, temp);}. The code in Figure 2 would then effectively produce the calls u.pop() and temp.push(x), both of which are effect-restricted. The call transferTop(u, u) has the net effect of removing the top entry of u. This conclusion can be reached purely from the *specification* of

transferTop—without consulting its body. This key feature of the proposed parameter-passing scheme, that makes it suitable for formal analysis, is explained in Section 4.

The same semantics of using initialized variables works for other repeated argument situations. For example, the call transferGlobalTop(g) removes the top entry of g. This is because we treat the global variable g as the fixed first parameter and view its use as an actual parameter as an argument repetition, rendering the effect of the call to be {Stack temp; transferGlobalTop(temp);}. For a record r, a call p(r, r.field3) has the same effect as {T_{field3} temp; p(r, temp);}, i.e., the effect of the call on the first record parameter dominates that on the subsequent field parameter. If the parameters were reversed, the effect on the field would override the modification to that field in the record. Passing an array and one of its elements as parameters to a call leads to similar effects.

3.1. Efficient Implementation

Even though we want to simplify reasoning about calls by keeping references from appearing in the programmer's world, a compiler can still use a modified version of parameter passing by reference to realize the proposed scheme efficiently and minimize initialization overhead—as long as this detail remains strictly internal to 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 {T temp; p(u, temp);} or {T temp; p(a, temp);}, respectively. The compiler can also generate analogous code for simple non-heap parameters in uncertain cases when static analysis can neither rule out repeated arguments nor show that they exist.

When the arguments are heap-resident and aliasing is uncertain, the compiler can use the following optimization. For each aliasing-uncertain actual parameter, it generates code first to copy the heap referencing value to the parameter stack and then to replace the prior reference with a special invalid address, thereby preserving the uniqueness of references into the heap. For example, suppose that a[exp1] and a[exp2] are two heap-resident arguments to a call, where exp1 and exp2 are index-valued expressions. The compiler generates code to copy the heap reference in location a[exp1] to the parameter stack and to replace that reference in location a[exp1] with a special invalid address. For the subsequent argument a[exp2], it generates code to check for that special invalid address. When exp2 equals exp1, then the checking succeeds and signals that repeated arguments exist. So instead of transferring a[exp2] to the parameter stack, the generated code creates and transfers an initial value of the appropriate type. On return, parameters are transferred back (in reverse order) from the parameter stack.

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 may 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 has no specified side-effects.

3.2. Combining the Initialization Scheme with Value Parameter Passing

Passing parameters by value does not create aliasing, poses no problem for clean semantics, and often is desirable for simple objects. Other parameter passing schemes are necessary because value copying cannot be implemented efficiently for non-trivial types and cannot in general even be mechanized correctly [9]. This section explains how the initialization scheme can be integrated into a language that allows by-value parameter passing as an option. For an example, consider a function declaration Tuple tupleSum(Tuple x, Tuple y). If repeated-argument calls, such as tupleSum(t, t), are anticipated but the default semantics are not what's wanted, then a specifier can override the semantics with a declaration such as: Tuple tupleSum(**eval** Tuple x, Tuple y). Here, the **eval** parameter mode denotes *expression evaluation mode*. Following this declaration, a call tupleSum(t, t) would be interpreted by the compiler as tupleSum(t.replica(), t), provided there is a special replica operation available for the **eval** mode parameter. If there is no such replica operation, then the call tupleSum(t, t) would be deemed erroneous. We prefer the naming **eval** over the traditional **val** because it clarifies that when a variable is passed where an expression is expected, it is just a shorthand for an implicit call to the replica operation.

The example illustrates that an operation may have some parameters in **eval** mode, while others are in default mode. In order to prevent initializations from affecting the results of expressions, **eval** arguments are evaluated before the initialization scheme is used to pass other arguments to the call.

4. Benefits for Formal Specification and Reasoning

This section explains the benefits of the initialization scheme for parameter passing in formal reasoning about procedure calls. We begin with a formal specification that captures the abstract notion of stacks in Figure 1.

```
class Stack;
        uses String_Theory;
        type Stack is modeled by Str(Object);
public Stack();
        ensures this = Λ;
public void push(Object x);
        updates this, x;
        ensures this = ⟨#x⟩ ○ #this;
public Object pop( );
        updates this;
        requires this ≠ Λ;
        ensures #this = ⟨pop()⟩ ○ this;
```

Figure 4. Specification of a Stack component

4.1. Example Specification

The specification in Figure 4 presents an abstraction of Stack variables as mathematical strings of objects. The assertion below the constructor public Stack() ensures that the *initial value* for every new Stack variable is Λ, i.e., the empty string. The *ensures* clause of push states that the value of this Stack after the operation will be the string concatenation (denoted by ○) of the incoming value of x (denoted by #x) and the incoming value of this Stack (denoted by #this). The pop function *requires* that the stack not be empty. In its *ensures* clause, the notation pop() denotes the object that is removed and returned by the function, and ⟨pop()⟩ denotes the string that contains that single object.

Using this specificational framework, a formal specification of the transferTop operation is given in Figure 5. The operation requires that the first stack not be empty. In the ensures clause, α^R denotes the reversal of string α, and |α| denotes the length of α. Using these operators, the specification states that the top entry from stack s is transferred to t. The initialization semantics makes it unnecessary for the specification to make any mention of repeated arguments or aliasing, yet makes its effect in the face of repeated arguments clear: By substituting Λ (the initial Stack value) for the input value of the second repeated argument, i.e., #t in the ensures clause, a caller can see that transferTop(u, u) results in a Stack u with its top entry removed—the same actual effect of the code in Figure 2.

```
uses Stack;
public static void transferTop(Stack s, Stack t);
        updates s, t;
        requires s ≠ Λ;
        ensures s^R ○ t = #s^R ○ #t and |s| = |#s| - 1;
```

Figure 5. A specification of the transferTop operation

4.2. Formal Reasoning about Calls

The code in Figure 2 can be proved to be a correct realization of the specification in Figure 5 whether or not arguments might be repeated. To enable such formal reasoning, we give a general proof rule for procedure calls. Suppose that v <– w denotes an initializing transfer operator that gives the value of w to v, and gives w an initial value of its type. Using this operator, the semantics of a call of the general form p(a, b) can be given as that of {T t1, t2; t1 <– a; t2 <– b; P(t1, t2); b <– t2; a <– t1;}. When a and b are distinct, it is easy to see that this sequence has the desired effect. When arguments are repeated, as in p(a, a), the semantics is equivalent to {T t1, t2; t1 <– a; t2 <– a; P(t1, t2); a <– t2; a <– t1;} and has the net effect of {T t2; p(a, t2);} because after the first transfer a gets an initial value and it is transferred to t2. After the call, the value of the first parameter becomes the value of a because of the reverse order of transfer on return: a <– t2; a <– t1. Given this idea, we discuss the proof rule for procedure calls shown in Figure 6.

Without loss of generality, we consider a call with only two arguments. The Context for this rule must include the specification of the called procedure, p, as shown. 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 for an assertion Q to be confirmed after the call. The rule introduces two local verification variables, specially named $\%a_x$ and $\%b_y$, to which the values of the corresponding actual arguments are transferred. The names a and b have been subscripted with names of the formal parameters x and y to avoid naming conflicts in the repeated argument case p(a, a) (which is important if an automated verification system is used).

(**void** p(T1 x, T2 y); **requires** pre; **ensures** post;) $\in$ Context;

Context \ assertive_code; T1 $\%a_x$; T2 $\%b_y$; $\%a_x$ <– a; $\%b_y$ <– b;

confirm pre[$x \rightsquigarrow \%a_x$, $y \rightsquigarrow \%b_y$] **and** ($\forall$ $?a_x$: T1, $\forall$ $?b_y$: T2,

post[$\#x \rightsquigarrow \%a_x$, $x \rightsquigarrow ?a_x$, $\#y \rightsquigarrow \%b_y$, $y \rightsquigarrow ?b_y$] $\Rightarrow$ Q'[$a \rightsquigarrow ?a_x$][$b \rightsquigarrow ?b_y$]);

Context \ assertive_code; p(a, b); **confirm** Q;

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

Figure 6. A general proof rule for verification of procedure calls

The rule uses the notation $\rightsquigarrow$ for substitution, and it requires two conjuncts to be proved. First, the precondition of p needs to be proved, after replacing the formals x and y with the actual arguments. Next, the assertion Q needs to be confirmed, assuming that the postcondition of p holds (with proper substitutions). Since the specification of p may be relational, multiple post-state values may result for the same input values. So the second conjunct states that as long as the post-state values of the arguments satisfy the post condition, Q must hold. The formal output names $?a_x$ and $?b_y$ denote possible post-state values; they must replace actual arguments a and b 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 steps ([ ], [ ]), not in parallel ([,]). When arguments are repeated and Q only involves a, for example, notice that the output value of the first parameter $?a_x$ will affect Q, but $?b_y$ will not. In the absence of repetition, the order of result return has no impact on the resulting state.

The semantics of the initializing transfer operator is given in Figure 7. Here, T.**is_initial**(x) is a predicate that tells whether its argument x of type T has an initial value for type T. For example, the predicate Stack.**is_initial**(s) is true if s is a Stack variable with the abstract value Λ—the postcondition of Stack's default constructor in Figure 4. We use a predicate instead of asserting, for example, s = Stack.**initial_value**, because a constructor operation may provide one of multiple possible initial values. If the ensures clause of the constructor is omitted for a type T, 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 our formal system. Since we assume it is possible to initialize variables of any type using a declaration such as T temp, every type must have a default public constructor with no arguments—a practice widely followed in component design.

Context \ assertive_code;

confirm ($\forall$ initial_T: T, T.**is_initial**(initial_T) $\Rightarrow$ Q'[b⇝initial_T][a⇝b]);

Context \ assertive_code; a <– b; **confirm** Q;

where Q' = Q with substitution for initial_T to avoid name conflicts.

Figure 7. Semantics of the initializing transfer operator

Using the rules given in this section, a proof of correctness for the implementation of transferTop and a proof for a piece of code that calls transferTop with repeated arguments are shown in [12]. The examples illustrate the use of the same procedure call 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

The vast body of literature on aliasing and its software engineering consequences are summarized in [11][12][20]. 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. Our initialization scheme for parameter passing preserves unique references even when repeated arguments occur and thereby circumvents one of the prime motivations for introducing borrowing.

Some previous treatments of 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 [14]. Whereas procedure call rules in [4] and [6] assume no argument repetition, rules that allow repeated arguments introduce references to handle them [2][8]. 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 clean semantics in this paper. But we differ in proposing the initialization scheme as the default approach with by-value parameter passing as an option.

We have explained one way of resolving the problem of providing clean semantics for procedure calls, including those with repeated arguments. Formal reasoning benefits of a clean semantics over one that requires objects to be viewed as references are documented in [12], where formal proofs of calls with repeated arguments using the two approaches are presented. In addition, [12] includes an experimental implementation of the scheme for calls using initial variables. The scheme we have proposed is implicit and simple, and it solves a subtle, yet long-standing obstacle to alias avoidance and prevention. When it is integrated with techniques for avoiding other sources of aliasing among

mutable objects, a new programming paradigm, in which languages have clean and rich semantics and in which it is easier to specify, develop, and maintain correct software, becomes possible.

Acknowledgments

This research is funded in part by the National Science Foundation grant CCR-0113181. Our special thanks go to Gary Leavens for his insightful comments throughout 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] 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.

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

[13] 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.

[14] 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.

[15] 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.

[16] 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.

[17] 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.

[18] 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.

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

[20] 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.