ACL — Eliminating Parameter Aliasing with Dynamic Dispatch

Gary T. Leavens and Olga Antropova

TR #98-08a
July 1998, revised February 1999

Keywords: reference parameter aliasing, global variable aliasing, multi-body procedures, dynamic dispatch, static dispatch, program verification, ACL language, alias-free programs, compiler optimizations, call-by-value and call-by-result patterns.

1997 CR Categories: D.3.1 [*Programming Languages*] Formal Definitions and Theory — semantics; D.3.3 [*Programming Languages*] Language Constructs and Features — control structures, procedures, functions, and subroutines; D.3.m [*Programming Languages*] Miscellaneous — dynamic dispatch, multiple dispatch; F.3.1 [*Logics and Meanings of Programs*] Specifying and Verifying and Reasoning about Programs — logics of programs.

Submitted for publication.

Department of Computer Science
226 Atanasoff Hall
Iowa State University
Ames, Iowa 50011-1040, USA

ACL — Eliminating Parameter Aliasing with Dynamic Dispatch

Gary T. Leavens[1] and Olga Antropova[2]

[1] Department of Computer Science, Iowa State University
226 Atanasoff Hall, Ames, Iowa 50011-1040 USA
http://www.cs.iastate.edu/~leavens/index.html
leavens@cs.iastate.edu
phone: +1 515 294 1580, fax: +1 515 294 1580
[2] Engineering Animation, Inc.,
2321 North Loop Dr., Ames IA, 50010 USA
olga@eai.com

Abstract. We have designed and prototyped a new approach for eliminating reference parameter aliases. This approach allows procedure calls with overlapping call-by-reference parameters, but guarantees that procedure bodies are alias-free. It involves writing multiple bodies for a procedure: up to one body for each possible aliasing combination. Procedure calls are dispatched to the appropriate procedure body based on the alias combination that occurs among the actual parameters and imported global variables; errors are generated if there is no corresponding body. This approach makes writing verifiable client code simpler, since clients do not need to write code to determine the aliasing combination among actuals. Furthermore, since procedure bodies are free of aliases, their static analysis and verification is easier.
The prototype language we have designed to explore these ideas incorporates some features to limit the number of alternative procedure bodies that a programmer must write.

Stream: foundations and methodology.
Mini-Track: languages and tools.
Keywords: reference parameter aliasing, global variable aliasing, multibody procedures, dynamic dispatch, static dispatch, program verification, ACL language, static analysis, call-by-reference.

1 Introduction

Two or more names that reference the same location are *aliases*. In this paper we concentrate on aliases generated by reference parameters. (We discuss how other kinds of aliasing are treated in the related work section below.)

Aliases and mutation make writing programs and reasoning about their correctness more difficult [13, 23]. The main problem is that verification of procedure correctness when aliasing is possible involves separate proofs for all possible aliasing combinations among formal parameters, and among formal parameters and global variables [12]. Note that there are, in general, an exponential number of such combinations.

Some compiler optimizations become impossible in the presence of aliasing [1, p. 648]. The main reason for this is that static analyses tend to lose precision in the presence of aliasing; again this is because of the exponential number of potential aliasing combinations. Loss of precision in static analysis results in slower executable code. For this reason, much research has concentrated on conservative flow analysis to statically detect aliases.

1.1 Problems Caused by Parameter Aliasing

Two kinds of aliasing can happen because of parameter passing with reference parameters. First, the same object may be passed twice as an actual parameter; for example, if a matrix multiplication procedure, `mm`, takes three reference parameters, then the procedure call `mm(a, a, a)` aliases the corresponding formals. Second, if a global variable is passed as an actual parameter by reference, then the global and formal become aliases. For example, in Figure 1 the formal parameter names `a` and `b` in are aliases within `size`'s body, as the type "`&int`" indicates that the parameter `b` is passed by reference.

```
var size: int = 10;
array x: int[10] = [1, 2, 3, 4, 5, 6, 7, 8, 9, 10];
proc sum(a: &int[], b: &int) imports (size) {
  var i: int = 1;
  b := 0;
  while (i < size + 1) {
    b := b + a[i];
    i := i + 1
  }
}
call sum(x, size)
```

Fig. 1. Aliasing of a global variable.

As illustrated by the above examples, a major problem with parameter aliasing is that programmers often forget that formal parameters may become aliases. However, the result of a procedure call may depend on the procedure implementation and combination of aliases at run-time.

In many contemporary programming languages parameter aliases are common. For example, C++ [29] has call-by-reference. Other object-oriented languages such as Smalltalk [11] and Java [4], and even mostly-functional languages such as ML [21] and Scheme [28], manipulate objects indirectly, through implicit references. In such languages assignment as well as parameter passing may cause aliasing.

1.2 Related Work and Treatment of Pointers

In this paper we take a language design approach to eliminating aliasing. This follows the design of Euclid [18, 20, 27, 32], which is a variant of Pascal [13, 19, 31] designed to aid program verification. Eliminating aliasing was an important design goal of Euclid. According to its authors, Euclid "... demonstrated that it is possible to completely eliminate aliasing in a practical programming language" [27, p. 16].

Pointer variables and pointer assignments are allowed in Euclid, but pointers are considered to be indexes into a "collection" of objects of the same type. Collections "are explicit program variables that act like the 'implicit arrays' indexed by pointers" ([27], p. 14). Thus the key ideas all relate to how to eliminate aliasing for arrays.

Arrays in Euclid use the "direct model" [10, Chapter 6], like Pascal and Ada [2, 5, 17], and thus distinct declarations of arrays do not overlap. However, Euclid does have call-by-reference.

The approach taken to eliminating aliases resulting from reference parameters is to prohibit procedure calls when the actual parameters overlap. This includes structured data passed along with a component (e.g., an array `a` and its element `a[1]`). When array elements `a[`i`]` and `a[`j`]` are passed as parameters, the requirement is that $i \neq j$. Often i and j are computed by expressions and it is not possible to determine statically whether these expressions yield distinct results. So Euclid requires the compiler "to generate a legality assertion to guarantee their distinctness" [27, p. 14]. This legality assertion is checked at run-time.

For global variables, Euclid requires explicit importation of those that are used by a procedure. Like parameters, imported globals should not overlap with the actual parameters.

Recent work on eliminating aliasing in object-oriented languages by Utting extends Euclid's idea of collections [30]. In Utting's work, complex objects (possibly sharing memory locations) are viewed as a set of disjoint collections (local stores) of homogeneous objects. Local stores are treated as arrays and pointers as indexes. This reduces the problem of dealing with even object-oriented pointer structures to the problem of dealing with arrays. For procedure calls, the requirements are similar to those in Euclid: actuals should be non-overlapping.

1.3 Problem with previous approaches

The way these previous approaches treat parameter aliases has a major disadvantage: the requirement that the parameters must be non-overlapping is too burdensome. For example, it is not uncommon in programming to make a procedure call such as `p2(a[i], a[j], a[k])`, where `p2` takes three reference parameters.

If the programming language prohibits procedure calls with overlapping actuals, then the author of such client code must either:

- prove that no aliases are possible (that a[i], a[j], and a[k] do not overlap), or
- write code to check for potential overlaps and call a different procedures depending on the aliasing combination among a[i], a[j], and a[k]. An example of such code is given in Figure 2.

In some cases it is not possible to decide statically whether the actuals overlap. This may happen if the variables i, j, and k depend on the user input. In other cases, it may just be too difficult to prove that there is no overlap, because the values of i, j, and k are the results of complex computations.

```
if (i == j && j == k) {
   call p2_123(a[i])
} else if (i == j) {
   call p2_12(a[i], a[k])
} else if (i == k) {
   call p2_13(a[i], a[j])
} else if (j == k) {
   call p2_23(a[i], a[j])
} else {
   call p2(a[i], a[j], a[k])
}
```

Fig. 2. Hand-coded analysis of aliasing combinations.

If one cannot prove that the actuals do not overlap, then Euclid essentially forces one to write alias analysis code like that in Figure 2. If this is not done, then there is the possibility that actuals may overlap at run-time, and then the legality assertion that the Euclid compiler generates will abort the program to prevent the aliasing.

In Figure 2, the procedures p2_123, p2_12, p2_13, and p2_23 are variants of p2 that handle different combinations of aliases. The idea would be that they handle each combination of aliases, and that they all achieve the same postcondition.

Unless some of these combinations can be statically ruled-out, similar alias analysis code is needed in all places where p2 is called. This repeated writing by clients of checking code is the main problem we solve in this paper. We believe that it is an important technical problem in making Euclid-style aliasing prohibitions practical.

1.4 Call-by-Value-Result is not a Solution

A language with call-by-value-result may seem to be a solution to the aliasing problem. This mechanism does eliminate aliases among actuals and globals during a call, since it makes a copy of each into fresh, non-overlapping storage.

From the point of view of program verification, however, the main problem is that in the presence of aliasing it may be impossible to reconcile the desired postconditions for different value-result parameters. Because of this, proof rules for languages with value-result and result parameters "usually" consider passing the same location to multiple result parameters to be "invalid" [23, p. 57]. That is, the actual parameters passed by value-result or result must not overlap, because copying back the results will not work in a verifiable way unless the actuals do not overlap. Hence, to reason about such a language one would need the same prohibitions on parameter aliasing as in Euclid.

In short call-by-value-result is not a solution to the problem of eliminating parameter aliasing, as the use of this mechanism also requires the prohibition of overlap among actual parameters. Since call-by-value-result is less efficient for passing parameters, we choose to focus on reference parameters.

1.5 Overview

The following sections describe our new approach to the problem of eliminating aliasing due to reference parameters. We also discuss our prototype implementation of a language that uses this approach — the programming language ACL (short for "Aliasing Controlling Language"). We look at programs in ACL, and discuss the results and implications of the approach. The conclusion section summarizes the paper and discusses directions for future work.

2 Prohibiting Aliases in Procedures

Our approach is a new way to avoid the aliasing caused by parameter passing. In particular it eliminates the aliasing due to reference parameters. It is different from the related work discussed above in that it automates calling an appropriate procedure body based on the aliasing combination that occurs dynamically. It does this using a variant of multimethod dispatch [6] [7] [22]. (Languages that use multimethod dispatch include CLOS [26] and Cecil [8].)

2.1 Our Approach: Dispatch Based on Aliasing Patterns

To allow for dynamic dispatch to different procedure bodies, programmers can write multiple alternative bodies for a procedure—up to one for each possible combination of aliases among the parameters and global variables.

The programmer need not, however, write all of the exponential number of bodies for each possible aliasing combination. If an omitted combination only involves aliases among constant reference parameters, then ACL will automatically generate the missing body, as explained below. Other omitted combinations cause an abort of the program if they occur at run time, as in Euclid; hence programmers do not need to write alternative bodies for aliasing combinations that are ruled out by preconditions.

To avoid unnecessary alias combinations with global variables, we adopt Euclid's idea of explicitly importing global variables in procedures [27]. (Functions and procedure names are implicitly available in procedure bodies since they cannot be aliased to variables in the language that we study.)

In general, dynamic dispatch must be used to find the appropriate procedure body to execute since the concrete alias combination among the parameters often cannot be determined until run-time. However, in many cases the aliasing combination is evident statically, and so static dispatch is possible as an optimization.

2.2 ACL explained

An ACL program consists of a command. Typically, this command is a "block" command; such a command is a sequence of declarations followed by a body, which is a sequence of commands. (The grammar of ACL is presented in Appendix A.) For example, the program in Figure 3 declares a global variable `a` and a procedure `swap`. Comments extend from `--` to the end of a line. The `swap` procedure has an alternative body which follows `| (x alias y)`. The body of the program calls `swap`, and the call executes the alternative body, since both actuals are the same. Note that this alternative achieves the desired postcondition by doing nothing.

```
var a: int = 1;
proc swap(x:&int, y:&int)
  -- ensures x == old(y) && y == old(x);
{
  var temp: int = x;
  x := y;
  y := temp
}
| (x alias y) {
   skip
}
call swap(a, a)
```

Fig. 3. The swap procedure in a small ACL program.

ACL is designed to have a small prototype implementation, but yet to be expressive enough to investigate the problem of eliminating reference parameter aliases using our approach.

ACL has integer and boolean literals and variables. It has arrays, because as described in the related work, the treatment of arrays is the key to dealing with aliases. Arrays in ACL use the direct model, like Euclid, Pascal and Ada. ACL has no pointers, but they could be added as explained in the related work section above.

The language has both functions and procedures. In ACL expressions and functions do not have side-effects. Procedures may modify the store but do not return values directly.

ACL has value parameters, reference parameters, and constant reference parameters. Value parameters are the default. Reference parameters are signaled by an ampersand (`&`) before the formal's type. Constant reference parameters are signaled by the keyword `const`. For example, a formal parameter may be declared as `const &int[]`, which is a constant reference to an array of integers.

These kinds of parameters are allowed for functions as well. Since ACL's functions have no side effects, they cannot observe aliasing. Thus no restrictions on aliasing or on the import of global variables are made within functions.

Procedures in ACL Procedures are the key feature of ACL. A procedure has a *header* containing a formal parameter list and an optional list of imported global variables, a main procedure body (for the case without any aliases), and zero or more alternatives, separated by vertical bars (`|`). Each *alternative* has a list of lists of aliases, which describes what aliasing combination it handles, and an alternative body, which is executed when that combination occurs among the actuals. In Figure 3, `swap` has one alternative, which handles the combination that occurs when both parameters are aliased.

Within a procedure body or alternative body, no two names are aliases. To guarantee this, names other than the first mentioned in an alias list cannot be used in the corresponding alternative body. For example, in Figure 3, `y` cannot be used within the alternative body.

An array element can be an alias to a formal parameter. For example, consider Figure 4, which is a procedure that computes the sum of the array `a` and stores it in `b`. Note that `size` is a value parameter and thus does not appear in alias lists. There is no import list, so no global variables are available in any of the procedure's bodies. Parameter `b` is of the same type as the elements of array `a`. Thus it is possible for the actual parameter initializing `b` to be an element of `a`. (Note that `a` is *not* declared as a constant reference parameter.)

ACL uses a declared index to allow an aliased element of an array to be named. For example, consider the alternative's alias list of Figure 4, where the declared index `j` allows the alias `b` to be named as `a[j]` within the alternative's body. The index `j` is declared by the programmer in the alias list[1]. ACL allocates an integer variable for such declared index, and initializes them to the correct index value at run-time. For example, in Figure 4 when `b` is an alias for an

[1] Such names must be distinct from other formals, imported globals, and other such declared indexes.

```
proc sum(a: &int[], b: &int, size: int)
  -- ensures b is the sum of a[0]..a[size-1]
{
  var i: int = 0;
  b := 0;
  while (i < size) {
    b := b + a[i];
    i := i + 1
  }
}
| (a[j:int] alias b) {
  var s:int;
  call sum(a, s, size);
  a[j] := s
}
```

Fig. 4. Summing an array in ACL.

element of `a` the alternative body is used, and then `j` is initialized so that `a[j]` denotes the same location as the actual parameter `b`.

Since all the elements of an array can be expressed using its name and subscripts, the name of an array and a declared index suffice for naming all aliases. For example, in the alternative body of Figure 4, the element `b` of array `a` can be referred to as `a[j]`. However, since the appropriate index value for `j` is unknown until run-time, if the variable aliased to an element were listed first in an alias list (e.g., `b alias a[j:int]`), then no other element of the array could be used in the corresponding alternative body. Therefore, ACL forces programmers to list an array (along with any declared indexes) first in each sublist of aliases in which arrays occur.

To show how imported global variables affect the alternatives for a procedure, we present in Figure 5 a procedure `sumGlobal` that uses a global variable `size`. An imported global variable is treated in much the same way as a reference parameter. However, there is one difference in that an atomic global variable cannot overlap with any array parameters, due to the semantics of ACL. For example, the alias combinations `a[j:int] alias size` and `a[j:int] alias b alias size` are not possible since the imported integer variable `size` is not part of any array.

Figure 5 also demonstrates the use of `const` reference parameters. In the header of `sumGlobal`, `a` is declared as a constant reference parameter. As in C++, `const` references cannot be assigned to within a procedure, nor can they be passed as non-const arguments to other procedures. If a constant reference and a non-constant reference both appear in an alias sublist, then the name used in the body is treated as a constant reference by the ACL type checker. In Figure 5,

```
var size: int = 10;

proc sumGlobal(a: const &int[], b: &int) imports (size)
  -- requires b is not aliased to an element of a
  -- ensures b is the sum of a[0]..a[old(size)-1]
{
  var i:int = 0;
  while (i < size) {
    b := b + a[i];
    i := i + 1
  }
}
| (b alias size)
{
  var s: int;
  call sumGlobal(a, s);
  b := s
}
```

Fig. 5. Summing an array with an imported global variable `size`.

this means that if `a` and `b` overlap, then there is no way to use an alternative body like the one in Figure 4, since that body assigns to `a`, which is `const` in Figure 5. Since `a` cannot be modified, the precondition in the specification of `sumGlobal` is needed. Because calls where `b` and `a` overlap are ruled out by this precondition, `sumGlobal` has no need for an alternative body for when `a` and `b` overlap; if they do overlap at run-time, then ACL will abort the program.

It is also possible to declare that imported globals are `const`. This has the effect of treating them like constant reference parameters, as opposed to being treated like reference parameters. If we modified `sumGlobal` in Figure 5 to declare `size` as a `const` import, then the precondition would also have to state that `size` and `b` cannot be aliased. This would mean that the alternative body in Figure 5 would become illegal, as the alias of `b` and `size` would cause `b` to be treated as a `const`. In that case the precondition would have to be adjusted.

ACL gives a warning when all possible alias combinations for a procedure are not handled in alternative bodies. One use for such warnings is to help decide that some reference or imported parameters should be declared using `const`. Another use is to check that the precondition prohibits all aliases that are not expected. Of course, the programmer may decide that the right thing is to simply write additional alternative bodies to handle some aliasing combinations.

Specification and Verification of ACL Procedures The method for specifying procedures in ACL we envision is that all the procedure bodies in a procedure work together to implement the same behavioral specification. This supports client code, in that clients should not have to worry about getting different results from calls with different aliasing combinations.

Of course, clients are responsible for not calling a procedure with an aliasing combination that violates the procedure's precondition. This is unchanged from the situation in other languages, such as Euclid. What makes ACL unique is that ACL makes it easy to write procedures that have very forgiving specifications with respect to aliasing. For example, in Figure 4, there is no precondition relating to the aliasing of the actuals at all. The `size` argument is copied, as it is passed by value, and overlaps among the other arguments are handled by alternative bodies.

To verify the correctness of an ACL procedure, one verifies that:

– each body is correct and
– that there are enough procedure bodies so that the disjunction of the predicates that describe their aliasing combinations is implied by the precondition.

Verifying the second of these conditions can be assisted by the ACL type checker. If ACL issues no warnings about a procedure having missing bodies, for example, then the second condition is trivially satisfied.

In doing the verification of a body, one can assume the conjunction of the precondition and a predicate that describes the specific aliasing pattern for that body. Note that code in each body has no aliasing at all. Calls to other procedures in the body can be treated using the specifications of those procedures; there is no need to prove any additional conditions on aliasing among the actuals passed to such procedures, other than any such conditions that are part of their preconditions.

2.3 Patterns

Often alternative procedure bodies follow common patterns. We discuss these patterns through a slightly larger example: matrix multiplication.

An ACL matrix multiplication procedure, `mm`, is given in Figure 6. It takes three matrices as reference parameters, `a`, `b`, and `c`, multiplies `b` by `c`, and stores the result in `a`. For simplicity we assume that the size of all dimensions of all the matrices are given by the value parameter `size`. A helping procedure, `copyMatrix`, is presented in Figure 7.

A *substitution* pattern, which occurs in the `(b alias c)` case of `mm`, allows use of call-by-reference when an aliasing combination is known to be harmless. In such a case, the code from the main body can be reused by substituting the first name in an alias list for the others in its list. The *substitution* pattern results in efficient code, since copies of the parameters are not made.

ACL automatically generates code using the substitution pattern applied to the main body, when the only aliased formals in an omitted body are declared as constant reference parameters. For example, if the formals `b` and `c` of `mm` were declared as constant references, then the `(b alias c)` case could be omitted. Of course, in that case, the precondition of `mm` would have to be adjusted to reflect

```
proc mm(a:&int[][], b:&int[][], c:&int[][], size:int)
  -- requires size > 0 && (* a, b, c are square size x size matricies *);
  -- ensures (* a is the product of old(b) and old(c) *);
{
  var i:int = 0;  var j:int;  var k:int;
  while (i < size) {
    j := 0;
    while (j < size) {
      k := 0;  a[i][j] := 0;
      while (k < size) {
        a[i][j] := a[i][j] + b[i][k] * c[k][j];  k := k + 0
      };
      j := j + 1
    };
    i := i + 1
  }
}
| (b alias c) {
  var i:int = 0;  var j:int;  var k:int;
  while (i < size) {
    j := 0;
    while (j < size) {
      k := 0;  a[i][j] := 0;
      while (k < size) {
        a[i][j] := a[i][j] + b[i][k] * b[k][j];  k := k + 0
      };
      j := j + 1
    };
    i := i + 1
  }
}
| (a alias b) {
  array temp: int[size][size];  call copyMatrix(temp, a, size);
  call mm(a, temp, c, size)
}
| (a alias c) {
  array temp: int[size][size];  call copyMatrix(temp, a, size);
  call mm(a, b, temp, size)
}
| (a alias b alias c) {
  array temp: int[size][size];
  call mm(temp, a, a, size);
  call copyMatrix(a, temp, size)
}
```

Fig. 6. Matrix multiplication in ACL.

```
proc copyMatrix(a:&int[][], b:&int[][], size:int)
  -- requires size > 0 && (* a, b are square size x size matricies *);
  -- ensures forall (i:int, j:int) 0 <= i < size && 0 <= j < size
  --                             => a[i][j] == old(b[i][j]);
{
  var i:int = 0;
  var j:int = 0;
  while (i < size ) {
    while (j < size) {
      a[i][j] := b[i][j];
      j := j + 1
    };
    i := i + 1;
    j := 0
  }
}
| (a alias b) {
  skip
}
```

Fig. 7. Copying a matrix in ACL.

the requirement that `a` not overlap with `b` or with `c`, and the last three bodies in `mm` could then also be eliminated. By not declaring `b` and `c` to be constant reference parameters, `mm` has a more forgiving specification.

A *call-by-value* pattern is found in the alternative bodies of `mm` for the alias combinations `(a alias b)` and `(a alias c)`. In this pattern, the code copies the aliased variables into locally declared variables, and then calls the procedure recursively with new actual parameters. This recursive call, since it has a different aliasing combination, is handled by a different body.

A *call-by-result* pattern occurs when one of the aliased variable serves as an accumulator for a result. This pattern occurs in the last alternative body of Figure 6, and in Figures 4 and 5. The pattern is to declare the local variable, call the procedure with a local variable, and copy the result of the computation back into the aliased variable.

A variation of the result-pattern is the *value-result-pattern*. In this pattern the initialization of the result variable is also needed for the computation.

ACL offers flexibility in choosing the appropriate pattern (parameter passing mechanism) for different alternative bodies. Whereas in other languages with call-by-value or call-by-value-result, copies are always made, regardless of the aliasing combination. This makes ACL's procedures more efficient than their counterparts in other languages.

Finally, there is the *error* pattern, which occurs when an aliasing combination violates the procedure's precondition. This is often needed when two or more reference parameters of the same type are acting as result parameters. By omitting the corresponding body, the programmer tells ACL to generate errors when such combinations occur.

2.4 Non-Patterns

The non-pattern cases are the main procedure body and cases where knowledge of the aliasing combination may be used to advantage. One example occurs in the alternative bodies for `copyMatrix` and in `swap`. Since the desired postcondition is already achieved by the aliasing combination for these alternatives, nothing needs to be done, and so the code, very efficiently, just does a `skip` command.

There are other examples, such as equality comparison tests and sequential search procedures, where the result is given immediately for some aliasing combinations. In general, the programmer can take advantage of the work already done by dynamic dispatch, using the predicate that characterizes an aliasing combination in conjunction with the precondition to achieve higher efficiency.

The ability to have programmers write non-parttern cases is an advantage of our approach over having various parameter mode annotations that would always automatically generate the appropriate code.

3 Implementation Issues

An earlier version of ACL has been implemented as a semantic interpreter by the second author [3]. It is written in the purely functional language Haskell [16], which allows the interpreter to closely resemble a denotational semantic definition of the language. That version of ACL did not include defined indexes, constant references, or the automatic generation of bodies for omitted aliasing combinations (either for constant reference overlap or error cases). The implementation of that version of the language is available from the following URL.

`ftp://ftp.cs.iastate.edu/pub/techreports/TR98-07/`

A type checker is included, which uses some clever tricks in Haskell to make the code look like the type inference rules for the language. These type inference rules were proved to satisfy a subject-reduction property [3].

Instead of discussing more details of that effort, we would like to make note of an algorithmic issue for the implementation of dynamic dispatch on aliasing combinations.

3.1 Static Dispatch

Recall that in may cases calls to procedures can be statically dispatched, as a compiler would be able to determine the aliasing combination statically. ACL should allow easier static determination of aliasing than most languages, because all names in a procedure body are known to be distinct. Hence the aliasing combination among actual parameters could often be determined statically. Even in some cases where the dispatch cannot be wholly static, one can statically construct a partial aliasing combination. Then at run-time this partial alias combination could be completed as necessary.

3.2 Dynamic Dispatch

When static alias analysis is not possible, the concrete combination of aliases among actual parameters must be determined at run-time, and used to select the appropriate procedure body. One way to do this would be to use pattern-matching techniques, which simultaneously analyze the aliasing combination and find the correct procedure body. To do this the procedure bodies would be statically organized in a decision tree with the parameter addresses comparisons as tests. The time complexity of pattern-matching dispatch would be $O(n \cdot \log n)$, where n is number of reference parameters and imported global variables[2].

3.3 Efficiency of ACL dispatch compared to Euclid

The necessity of dynamic dispatch could be considered a disadvantage of the multi-body procedures approach. We claim, however, that ACL programs need be no slower in the worst case than equivalent Euclid programs [27]. This claim is true despite the fact that in Euclid's procedure calls always statically dispatch to exactly one body.

To see the truth of this claim, recall that, unless one can statically prove otherwise, for correctness additional code similar to the code used by ACL to do dynamic dispatch must be written in a Euclid program at the point of each procedure call (as in Figure 2). We assume equally "smart" compilers for Euclid and ACL, so that whenever the Euclid compiler can prove what the aliasing combination of a call is, then the ACL compiler could also carry out the same proof. If they both know the aliasing combination for a call statically, they can both compile code that jumps directly to the appropriate body, resulting in equal speed.

If the aliasing combination cannot be discovered statically, then there must be more than one aliasing combination that seems to be possible (to both compilers). In this case the Euclid compiler would insert an assertion to check for aliasing at run-time; in the worst case for this assertion (when it finds no aliasing), its running time would be the same as the worst case time needed for dynamic dispatch in ACL.

The Euclid programmer can help the Euclid compiler and avoid the possibility of having a call aborted because of aliasing by writing alias analysis code by hand. But such code can be no better in the worst case than an ACL compiler could do. Thus, given an ACL compiler that is as sophisticated as a Euclid compiler, ACL programs need be no slower in the worst case than Euclid programs.

[2] To calculate this we took $O(n^n)$ as an upper bound on the number of the procedure bodies. Then traversing a binary decision tree yields the given upper bound.

We note that the advantage of ACL is that programs are likely to be less error prone, since the responsibility for writing code to determine the aliasing combination is moved from the application program to the compiler.

We have chosen Euclid for comparison since it presents an extreme example of the separation of alias analysis code and procedures. However, even in languages that, unlike Euclid, allow reference parameters to overlap, a programmer will usually need to write code, either at the site of a procedure call or in the procedure itself, to ensure correctness when the aliasing combination is not statically known.

Of course, one would need to actually build a real ACL compiler to see how it compared with other languages in terms of efficiency. We believe that efficiency would depend crucially on how many calls could be statically dispatched. We are optimistic, however, because ACL should permit much better static analysis, as there is no aliasing in ACL code.

4 Conclusion and future work

It is worthwhile to emphasize again that avoiding aliasing is important not just for correctness, but also to enable better compiler optimizations. Our approach allows freedom from aliasing without making it much more difficult to use procedures and without sacrificing the efficiency of call-by-reference.

In essence, ACL makes the Euclid [27] approach to avoiding parameter aliasing practical by taking the responsibility for alias analysis away from procedure clients and giving it to the procedure's implementor. This makes the work of a procedure's clients easier. Yet, like Euclid, ACL retains both call-by-reference and the benefits of eliminating aliasing. Euclid also had pointers and pointer variables, but these could be treated in a similar way in ACL [27,30].

It may be that the most important benefit is the greatly increased opportunity for code optimization that the lack of aliasing allows. However, whether this is true in practice remains to be seen.

ACL warns the programmer when bodies for aliasing combinations are missing; this is a mechanical aid towards proofs of correctness for procedure implementations. Such warnings may be useful even for programmers who are not concerned with doing formal program verification.

As we argued in Section 3.3, the efficiency of programs written in ACL need be no slower in the worst case than equivalent Euclid programs.

ACL is a small experimental language which investigates the basic implications of the idea of dynamic dispatch and multi-body procedures. It would be interesting to study how the idea would apply to languages that operate on more complex objects, as occur in object-oriented languages. Recently several designs for object-oriented languages that deal with aliasing have appeared [9,14,15, 25,30]. Since these works concentrate on other kinds of aliasing, as opposed to parameter aliasing, it would be interesting to combine our ideas with theirs in a single language.

Acknowledgements

Thanks to Mark Utting for discussions about aliasing. Thanks to Mark and Jim Horning for comments on an earlier draft and for suggesting how to support the error pattern as described in Section 2.3. Thanks to Peter Müller, Don Pigozzi, and R. C. Sekar, for discussions about this work. Thanks also to Don and Clyde Ruby for comments on a draft.

Leavens's work was supported in part by NSF Grants CCR-9593168 and CCR-9803843.

References

1. R. Sethi A. V. Aho and J. D. Ullman. *Compilers. Principles, Techniques and Tools.* Addison-Wesley, 1986.
2. American National Standards Institute. *Reference Manual for the Ada Programming Language*, February 1983. ANSI/MIL-STD 1815A. Also published by Springer-Verlag as LNCS 155.
3. Olga Antropova. Acl — eliminating parameter aliasing with dynamic dispatch. Technical Report 98-07, Department of Computer Science, Iowa State University, Ames, Iowa, 50011, July 1998. Available by anonymous ftp from ftp.cs.iastate.edu, and by e-mail from almanac@cs.iastate.edu.
4. Ken Arnold and James Gosling. *The Java Programming Language.* The Java Series. Addison-Wesley, Reading, MA, second edition, 1998.
5. John Barnes, Ben Brosgol, et al. Ada 95 rationale. the language. the standard libraries. Technical report, Itemetrics Inc., 733 Concord Av, Cambridge, MA 02138, January 1995.
6. Daniel G. Bobrow, Kenneth Kahn, George Kiczales, Larry Masinter, Mark Stefik, and Frank Zdybel. Commonloops: Merging lisp and object-oriented programming. *ACM SIGPLAN Notices*, 21(11):17–29, November 1986. OOPSLA '86 Conference Proceedings, Norman Meyrowitz (editor), September 1986, Portland, Oregon.
7. Giuseppe Castagna. *Object-Oriented Programming: A Unified Foundation.* Progress in Theoretical Computer Science. Birkhauser, Boston, 1997.
8. Craig Chambers. Object-oriented multi-methods in Cecil. In Ole Lehrmann Madsen, editor, *ECOOP '92, European Conference on Object-Oriented Programming, Utrecht, The Netherlands*, volume 615 of *Lecture Notes in Computer Science*, pages 33–56. Springer-Verlag, New York, N.Y., 1992.
9. David G. Clarke, John M. Potter, and James Noble. Ownership types for flexible alias protection. In *OOPSLA '98 Conference Proceedings*, volume 33(10) of *ACM SIGPLAN Notices*, pages 48–64. ACM, October 1998.
10. Daniel P. Friedman, Mitchell Wand, and Christopher T. Haynes. *Essentials of Programming Languages.* McGraw-Hill Book Co., New York, N.Y., 1992.
11. Adele Goldberg and David Robson. *Smalltalk-80, The Language and its Implementation.* Addison-Wesley Publishing Co., Reading, Mass., 1983.
12. D. Gries and G. Levin. Assignment and the procedure call proof rules. *TOPLAS*, 2(4):564–579, 1980.
13. C. A. R. Hoare and N. Wirth. An axiomatic definition of the programming language pascal. *Acta Informatica*, 2(4):335–355, 1973.

14. John Hogg. Islands: Aliasing protection in object-oriented languages. In *Proceedings of the OOPSLA '91 Conference on Object-oriented Programming Systems, Languages and Applications*, pages 271–285, November 1991. Published as ACM SIGPLAN Notices, volume 26, number 11.
15. John Hogg, Doug Lea, Alan Wills, Dennis deChampeaux, and Richard Holt. The Geneva Convention on the treatment of object aliasing. *OOPS Messenger*, 3(2):11–16, April 1992.
16. Paul Hudak, Simon Peyton Jones, Philip Wadler, et al. Report on the programming language Haskell: A non-strict, purely functional language, version 1.2. *ACM SIGPLAN Notices*, 27(5), May 1992.
17. International Organization for Standardization. *Ada 95 Reference Manual. The Language. The Standard Libraries*, January 1995. ANSI/ISO/IEC-8652:1995.
18. Butler W. Lampson, James L. Horning, Ralph L. London, James G. Mitchell, and Gerald J. Popek. Report on the programming language euclid. Technical Report CSL-81-12, Xerox Palo Alto Research Centers, October 1981. Also *SIGPLAN Notices*, 12(2), February, 1977.
19. Henry Ledgard. *The American Pascal Standard: with Annotations*. Springer-Verlag, New York, N.Y., 1984. Contains ANSI/IEEE 770X3.97-1983.
20. 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):1–26, 1978.
21. Robin Milner, Mads Tofte, and Robert Harper. *The Definition of Standard ML*. The MIT Press, Cambridge, Mass., 1990.
22. David A. Moon. Object-oriented programming with *Flavors*. *ACM SIGPLAN Notices*, 21(11):1–8, November 1986. OOPSLA '86 Conference Proceedings, Norman Meyrowitz (editor), September 1986, Portland, Oregon.
23. Caroll Morgan. Procedures, parameters and abstraction: separate concerns. In Morgan and Vickers [24], pages 47–58.
24. Carroll Morgan and Trevor Vickers, editors. *On the refinement calculus*. Formal approaches of computing and information technology series. Springer-Verlag, New York, N.Y., 1994.
25. James Noble, Jan Vitek, and John Potter. Flexible alias protection. In Eric Jul, editor, *ECOOP '98 – Object-Oriented Programming, 12th European Conference, Brussels, Belgium*, volume 1445 of *Lecture Notes in Computer Science*, pages 158–185. Springer-Verlag, July 1998.
26. Andreas Paepcke. *Object-Oriented Programming: The CLOS Perspective*. The MIT Press, 1993.
27. G. J. Popek, J. J. Horning, B. W. Lampson, J. G. Mitchell, and R. L. London. Notes on the design of euclid. *ACM SIGPLAN Notices*, 12(3):11–18, March 1977. Proceedings of an ACM Conference on Language Design for Reliable Software, Raliegh, North Carolina, March, 1977.
28. Jonathan Rees, William Clinger, H. Abelson, N. I. Adams IV, D. H. Bartley, G. Brooks, R. K. Dybvig, D. P. Friedman, R. Halstead, C. Hanson, C. T. Haynes, E. Kohlbecker, D. Oxley, K. M. Pitman, G. J. Rozas, G. J. Sussman, and M. Wand. Revised^3 report on the algorithmic language scheme. *ACM SIGPLAN Notices*, 21(12):37–79, December 1986.
29. Bjarne Stroustrup. *The C++ Programming Language: Third Edition*. Addison-Wesley Publishing Co., Reading, Mass., 1997.
30. Mark Utting. Reasoning about aliasing. In *Proceedings of the Fourth Australasian Refinement Workshop (ARW-95)*, pages 195–211. School of Computer Science and

Engineering, The University of New South Wales, April 1995. Available from `http://www.cs.waikato.ac.nz/~marku`
31. N. Wirth. The programming language pascal. *Acta Informatica*, 1(1):35–63, 1971.
32. D. B. Wortman and J. R. Curdy. Early experience with euclid. In *Fifth International Conference on Software Engineering.* IEEE, 1980.

A Appendix: Concrete Syntax of ACL

The following is the concrete syntax for ACL. Names in *italic* font are nonterminals Keywords and terminal symbols are in a `typewriter` font. We use curly brackets ({}) as meta-symbols for grouping, and $^+$ and * for zero, or one or more of the preceding group. Phrases inside the square brackets ([...]) are optional. Comments in ACL programs and in the grammar extend from a dash (--) to the end of a line.

```
Prog ::=                                                  -- Program
      Comm

Decl ::=                                                  -- Declaration
      var Name : BasicType [ = Exp ]
    | array Name : BasicType { [ Exp ] }+ [ = Exp ]
    | fun Name Formals : BasicType { Exp }
    | proc Name Formals [ imports (ImpList) ] ProcBody { '|' Alt }*
    | Decl { ; Decl }+

ProcBody ::=                                              -- Procedure-Body
      { Comm }

Alt ::=                                                   -- Alternative-Body
    | ( AliasList ) { Comm }

AliasList ::=                                             -- Alias-List
      OVL { alias OVL }+ { , OVL { alias OVL }+ }*

OVL ::=                                                   -- Overlapping-Location
      Name { [ Name : BasicType ] }*

ImpList ::=                                               -- Import-List
      [ const ] Name { , [ const ] Name }*

Comm ::=                                                  -- Command
      Loc := Exp
    | if Exp { Comm } else { Comm }
    | while Exp { Comm }
    | skip
    | Comm { ; Comm }+
    | call Name Actuals                                     -- procedure call
    | Decl [ ; ] Comm

Formals ::=                                               -- Formal-Parameters
      ( [ Formal { , Formal }* ] )
```

```
Formal ::=                                   -- Formal-Parameter
      Name : FormalType

Exp ::=                                      -- Expression
      NumLit
    | BoolLit
    | [ Exp { , Exp }* ]                       -- literal array
    | Exp + Exp | Exp * Exp | Exp - Exp | Exp / Exp
    | ! Exp | Exp '||' Exp | Exp && Exp | Exp == Exp
    | Exp < Exp | Exp > Exp | Exp <= Exp | Exp >= Exp
    | if Exp { Exp } else { Exp }
    | Name Actuals                             -- function call
    | Loc                                      -- dereferencing location
    | let Decl in Exp
    | Name { [ Exp ] }+                        -- identifier expression

Actuals ::=                                  -- Actual-Parameters
      ( [ Exp { , Exp }* ] )

Loc ::=                                      -- Location
      Name | Name { [ Exp ] }+

NumLit ::=                                   -- Numeral-Literal
      Integer

BoolLit ::=                                  -- Boolean-Literal
      true | false

FormalType ::=                               -- Formal-Type
      [ const [ & ] ] BasicType { [] }*

BasicType ::=                                -- Basic-Type
      int | bool
```