ACL — Eliminating Parameter Aliasing with Dynamic Dispatch

Olga Antropova and Gary T. Leavens

TR #98-08
July 1998

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

Olga Antropova and Gary T. Leavens*
Department of Computer Science, Iowa State University
226 Atanasoff Hall, Ames, IA, 50011-1040 USA
olga@cs.iastate.edu, leavens@cs.iastate.edu
phone: +1 515 294-1580 fax: +1 515 294-0258

July 24, 1998

Abstract

We have implemented a new method for eliminating reference parameter aliases. This method allows procedure calls with overlapping call-by-reference parameters, but at the same time guarantees that procedure bodies are alias-free. The method involves writing multiple bodies for a procedure: one per aliasing combination. Calls are automatically dispatched to the appropriate procedure body based on the particular alias combination among the actual parameters and imported global variables. This makes writing verifiable client code simpler, since such code does not need to determine the aliasing combination before the procedure is called. The efficiency of dispatch to these bodies is no worse than hand-coded determination of the aliasing combination would be in other languages.

In our experience, the number of necessary procedure bodies is usually small, which makes the approach practical. Forcing programmers to write one body for each aliasing combination also makes them consider each case of aliasing among the parameters and globals, making it more likely that the procedure is correctly implemented.

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

```
var x:int = 1;
proc p1 (y:&int)
{...
  y := 1; ...
  x := 2; ...}
in call p2(x)
```

Figure 1: Aliasing of a global variable.

1 Introduction

Two or more names that reference the same location are called *aliases*. Aliases and mutation make writing programs and reasoning about their correctness more difficult [Mor94] [HW73]. Some compiler optimizations become impossible in the presence of aliasing [AVAU86, p. 648], which results in slower executable code. For this reason, much research has concentrated on flow analysis to detect aliases.

In this paper we concentrate on aliases generated by reference parameters. The language we study has neither alias declarations nor pointer variables. 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 procedure `p` takes two reference parameters then the procedure call `p(x, x)` aliases the corresponding formals. Second, if a global variable is passed as an actual parameter by reference, then the global and formal become aliases. In Figure 1 the names `x` and `y` in `p1`'s body are aliases, as the type "`&int`" indicates that the parameter is passed by reference.

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.

Example 1.1 *Consider a matrix multiplication procedure with the following header.*

```
proc mm(a[][]:&int, b[][]:&int, c[][]:&int)
```

In this procedure, the result of multiplying `b` *by* `c` *is accumulated in matrix* `a`*. If* `mm` *directly works on* `a`*,* `b`*, and* `c` *(not on copies), then the results of the following procedure calls will all be incorrect.*

```
mm(x, x, x)
mm(x, x, y)
mm(x, y, x)
```

When aliasing is possible verification of a procedure's correctness is difficult. It involves separate proofs for all possible aliasing combinations among formal parameter names, and formal parameters and global variable names [GL80].

In many contemporary programming languages parameter aliases are common. For example C++ [Str97] has call-by-reference. Other object-oriented languages such as Smalltalk [GR83] and Java [AG98], and even mostly-functional languages such as ML [MTH90] and Scheme [RCA+86], manipulate objects indirectly, through references. In such languages assignment as well as parameter passing may cause aliasing.

1.1 Related work

Back in 1977 the programming language Euclid was developed. Euclid was designed to aid program verification, and prohibiting alias was important in making verification practical. Euclid "... demonstrated that it is possible to completely eliminate aliasing in a practical programming language" [PHL+77, p. 16]. The approach the authors took to eliminate aliases resulting from reference parameters was to prohibit procedure calls when the actual parameters overlapped. 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 would be that $i \neq j$. Often i and j are computed by expressions, and it is not possible to determine statically whether $i \neq j$. Euclid requires the compiler "to generate a legality assertion to guarantee their distinctness" [PHL+77, p. 14]. For global variables, Euclid requires explicit importation of those that are

used by a procedure. Like parameters, imported globals should not overlap with the parameters. Pointer variables and pointer assignments are allowed in Euclid, but pointers are considered to be indexes into the "collection" of objects of the same type. Collections "are explicit program variables that act like the 'implicit arrays' indexed by pointers" ([PHL+77], p. 14). Restrictions similar to those on arrays and their elements apply to collections and elements of collections, when those are passed to a procedure or imported.

Recent work on eliminating aliasing in object-oriented languages by Utting extends Euclid's idea of collections [Utt95]. 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. The proof rules for arrays can thus be applied. For procedure calls, the requirements are similar to those in Euclid: actuals should be non-overlapping.

1.2 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. It is not uncommon in programming to make a procedure call such as `p2(a[i], a[j], a[k])`. If the programming language prohibits procedure calls with overlapping actuals, then client code must check for overlaps and call a different procedure (with fewer arguments) if there is an overlap. Note that in some cases it is not possible to decide statically if the parameters overlap. Suppose that variables `i`, `j`, and `k` depend on the user input, or are the results of complex computations. In such cases whether some of them are equal can only be known at run-time. Figure 2 shows how the additional alias analysis code might look; the procedures `p2_1`, `p2_2`, `p2_3`, and `p2_4` are variants of `p2` that handle different combinations of aliases.

Unless some of these combinations can be statically ruled-out, similar code is needed in all places where `p2` is called. Note also that, in general (as shown in Figure 2), a different procedure is needed for each possible alias combination.

A language with call-by-value-result may seem to be a solution to the aliasing problem. One problem is that the efficiency of call-by-reference is lost. A more important problem, however, is that in the presence of aliasing it may be impossible to reconcile the desired postconditions for different value-result parameters in the presence of aliasing. Because of this, proof

```
if (i == j) {
   p2_1(a[i], a[k])
} else if (i == k) {
   p2_2(a[i], a[j])
} else if (j == k) {
   p2_3(a[i], a[j])
} else if (i == j && j == k) {
   p2_4(a[i])
} else {
   p2(a[i], a[j], a[k])
}
```

Figure 2: Hand-coded analysis of aliasing combinations.

rules for languages with call-by-value-result (and call-by-result) "usually" consider passing the same location to multiple result parameters to be "invalid" [Mor94, p. 57]. Hence, to reason about such a language one would need prohibitions on parameter aliasing that are similar to Euclid's.

The following sections describe the new approach to the problem of eliminating aliasing due to reference parameters, and our experimental implementation of it—the programming language ACL. We look at programs in ACL, discuss the results and implications of the approach, and analyze the efficiency of the approach. The conclusion summarizes the results of the experiment, and discuss directions for future work.

2 Prohibiting aliases in procedures

This article presents a new way to avoid the aliasing caused by parameter passing. It is different from those described above in that it automates calling the appropriate procedure based on the aliasing combination that occurs dynamically, using a variant of multimethod dispatch [BKK+86] [Cas97] [Moo86] as found, for example, in CLOS [Pae93] and Cecil [Cha92].

2.1 Our approach: dispatch based on aliasing patterns

To prohibit parameter aliasing, our approach expects a procedure implementation to have multiple bodies—one for each possible combination of aliases among the parameters and global variables. Each procedure body implements the same behavioral specification for one of the alias combinations.[1] In the procedure body for a particular alias combination, aliased locations can only be referenced through one of the aliased names.

To avoid unnecessary alias combinations with global variables, we adopt Euclid's idea of explicitly importing global variables in procedures [PHL+77]. (Functions and procedures 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.

An important implication of the proposed approach is that both static analysis and program verification become simpler, since aliasing is eliminated. Furthermore, writing client code is easier than in Euclid, since alias analysis code need not be written repeatedly by hand, at each call site. In our approach this additional code is part of a compiler and can be verified once and for all.

To experiment with this idea we implemented a small imperative programming language called ACL (Alias Controlling Language). The experiment's goals were:

1. to implement algorithms for type checking procedure declarations that ensure that enough bodies are written by the ACL programmer, to ensure that every possible aliasing combination has a body,

2. to implement algorithms for dynamic dispatch to the correct procedure body, and

3. to investigate the feasibility of implementing procedures with multiple bodies.

[1] In cases where the precondition prohibits a particular aliasing combination, an error can be signaled.

The results of these experiments are described in the following sections. In summary, static dispatch can be used for most procedure calls, and the complexity of dynamic dispatch is $O(n \cdot \log n)$, where n is the number of the reference parameters. The number of possible alias combinations, and hence of the alternative procedure bodies is exponential in the number of parameters. However, in practical examples this number turns out to be not too large, and in fact it seldom exceeds two.

2.2 ACL explained

The grammar of ACL is presented in Appendix A. The language is designed to be small, yet expressive enough to investigate the problem of eliminating reference parameter aliases and our approach to solving it.

ACL has integer and boolean literals and variables. Arrays are implemented in order to investigate passing structured data and their elements to procedures. 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.

ACL has both value parameters, and reference parameters. Reference parameters are signaled by an ampersand (`&`) before the formal's type.

Both kinds of parameters are allowed for functions as well as procedures. Since 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.

2.2.1 Procedures

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. The main procedure body and the alternative bodies each implement the same behavioral specification, but each only handles one combination of aliases among the parameters and imported variables.

There are three factors in ACL that reduce the number of possible aliasing combinations:

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

Figure 3: The swap procedure in ACL.

1. value parameters cannot be aliases to other parameters or to global variables;

2. parameters of different types cannot be aliased to each other in ACL[2], likewise a parameter of one type cannot be aliased to a global variable of another type;

3. since ACL uses the "direct model" [FWH92] of arrays (as in Pascal) and has no reference or pointer variables, imported global variables cannot be aliases to each other, and an atomic imported global cannot be an alias to an element of an array.

An ACL program is a sequence of declarations followed by the keyword `in` and a command or sequence of commands. For example, the program in Figure 3 declares a global variable `a` and a procedure `swap`. The body of the program calls `swap`, and the call executes the alternative body, since both actuals are the same.

An array element can be an alias to a formal parameter. For example, consider Figure 4, which implements 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`. This requires an alternative procedure body.

[2]However, this would not be true in a language that permitted subtyping.

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

Figure 4: Summing an array in ACL.

Within a procedure 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.

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 variable `j` allows the alias `b` to be named as `a[j]` within the alternative's body. The subscript `j` is declared by the programmer in the alias list[3]. ACL allocates an integer variable for such declared subscripts, and initializes them to the correct subscript at run-time. For example, in Figure 4 when `b` is an alias for an element of `a`, the declared `j` is initialized so that `a[j]` denotes the same location as `b`.

Observe that all elements of a structured variable can be expressed using its name and a declared index; for example, in Figure 4, the element `b` of array `a` can be expressed as `a[`j`]` for some j. However, since the index j is unknown, 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. Hence in such cases an ACL programmer must list an array (along with the declarations of unknown indexes) first in

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

```
var size:int = 10;
array[10] x:int = [1, 2, 3, 4, 5, 6, 7, 8, 9, 10];
proc sum1(a[]:&int, b:&int) imports (size) {
   var i:int = 1 in
   b := 0;
   while (i < size + 1) {
      b := b + a[i];
      i := i + 1}}
| (a[j:int] alias b) {
   var s:int in
   call sum1(a, s);
   a[j] := s }
| (b alias size) {
   var s:int;
   call sum1(a, s);
   b := s }
in call sum1(x, size)
```

Figure 5: Summing an array with a global variable `size`.

list of aliases.

To show how imported global variables affect the implementation of a procedure, we present in Figure 5 a variant of the procedure `sum` that uses a global variable `size`. The alias combinations `(a[j:int] alias size)` and `(a[j:int] alias b alias size)` are not possible since the global variable `size` cannot be an element of any array.

Some alias combinations make it impossible to satisfy a procedure's specified post-condition. Consider a procedure `min_max` that finds the minimum and maximum elements of a given array and stores the results in reference parameters `min` and `max`. The specification of such procedure should prohibit the `(min alias max)` combination by requiring the corresponding actuals to be disjoint in its precondition. In general alias analysis code should be present where the procedure is called in the client code to avoid violating such a precondition. ACL forces the programmer to write an alternative procedure body for this aliasing combination, which has the effect of making

the implementation more defensive than necessary. (We consider an idea for getting around this restriction in Section 2.4 below.)

2.3 Patterns and Non-Patterns

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

The matrix multiplication procedure, `mm`, takes three matrices by reference, `a`, `b`, and `c`, multiplies `b` by `c`, and stores the result in `a`. Figure 6 presents the definition of `mm` in ACL. For simplicity we assume that both dimension sizes for all matrices are equal. 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.

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

```
proc mm(a[][]:&int, b[][]:&int, c[][]:&int, size:int) {
  var i:int = 1; var j:int; var k:int in
  while (i < size + 1) {
    j := 1;
    while (j < size + 1) {
      k := 1; a[i][j] = 0;
      while (k < size + 1) {
        a[i][j] := a[i][j] + b[i][k] * c[k][j];
        k := k + 1};
      j := j + 1};
    i := i + 1}}
| (b alias c) {
  var i:int = 1; var j:int; var k:int in
  while (i < size + 1) {
    j := 1;
    while (j < size + 1) {
      k := 1; a[i][j] = 0;
      while (k < size + 1) {
        a[i][j] := a[i][j] + b[i][k] * b[k][j];
        k := k + 1};
      j := j + 1};
    i := i + 1}}
| (a alias b) {
  array[size][size] temp:int in
  call copyMatrix(temp, a, size);
  call mm(a, temp, c, size)}
| (a alias c) {
  array[size][size] temp:int in
  call copyMatrix(temp, a, size);
  call mm(a, b, temp, size)}
| (a alias b alias c) {
  array[size][size] temp:int in
  call mm(temp, a, a, size);
  call copyMatrix(a, temp, size) }
```

Figure 6: Matrix multiplication in ACL.

```
proc copyMatrix(a[][]:&int, b[][]:&int, size:int) {
   var i:int = 1; var j:int in
   while (i < size + 1) {
     j := 1;
     while (j < size + 1) {
       a[i][j] := b[i][j];
       j := j + 1};
     i := i + 1}}
| (a alias b) {skip};
```

Figure 7: Copying a matrix in ACL.

or more reference parameters of the same type are acting as result parameters.

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 a `comparison` or `search` procedure, the result is given immediately for some aliasing combinations.

2.4 Relaxing the "one body per aliasing combination" Requirement

The patterns described above may allow automated generation of some alternative bodies. That is, a language like ACL could let the programmer implement the main procedure body and the cases where knowledge of the aliasing combination may be used to advantage, and have the programmer give annotations on the formals to indicate what patterns should be followed for the other bodies. For example, Euclid and C++ both have a way to declare that certain reference parameters are read-only. Such an annotation would let a compiler automatically apply the substitution pattern when just

read-only parameters are aliased. We leave exploration of the general form of such extensions to ACL as future work.

However, one simple way to support the error pattern would be to treat omitted alternative bodies as instructions to automatically generate a body that signals an error when called. Then when a call to the procedure dispatched to such an omitted body, an error would be signaled. (When static dispatch to such a case is possible, a compiler could statically issue an error message.) This has the advantage, compared to ACL, that it does not require programmers to write alternative bodies for aliasing combinations that are ruled out by preconditions. The disadvantage of this approach, however, is that it would be impossible to determine whether an alternative had been intentionally or accidentally omitted.

3 Implementation Overview

ACL has been implemented as an interpreter written in Haskell [HJW+92]. The implementation is available from the following URL.

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

In this section we briefly describe some of its key algorithms. We also discuss the possible improvements.

3.1 Constructing and counting aliases combinations

It is intuitive that the number of procedure bodies to cover all aliasing combinations should be exponential. However the exponential number of alias combinations is not fundamentally a property of ACL, but rather of logics for reasoning about programs in the presence of aliases. Hence the same number of alias combinations must be dealt with in reasoning about programs written in any language. Recall that, in general, different procedure bodies are needed for different aliasing combinations in Euclid programs (see Figure 2). The exponential number of aliasing combinations is thus not a key measure for the practicality of our approach.

But how many procedure bodies must the programmer write? All possible combinations of aliases among variables of the same type can be constructed by listing all possible partitions and then deleting partitions with only one element.

Example 3.1 *The partitions of the set* $\{a, b, c, d\}$ *are as follows. (In the lists below, think of combinations of names, like ab, as being aliased variables, and single names, like b, as not being aliased.)*

[a, b, c, d] [a, b, cd] [a, bc, d] [a, bd, c] [a, bcd]
[ab, c, d] [ac, b, d] [ad, b, c] [ab, cd] [acd, b]
[abc, d] [ad, bc] [abd, c] [ac, bd] [abcd]

Removing the single names from each of the partitions above gives all aliasing combinations:

[] [cd] [bc] [bd] [bcd] [ab] [ac] [ad] [ab, cd]
[acd] [abc] [ad, bc] [abd] [ac, bd] [abcd]

This example shows that the number of all possible combinations of aliases among four reference parameters (of the same type) plus a no-alias case is fifteen. An upper bound for number of the alternative procedure bodies for n reference parameters, $C_{pb}(n)$ is:

$$C_{pb}(n) = O(n!).$$

However, this is a gross overestimate, as the following table shows.

n	1	2	3	4	5	6	7	8	9
$C_{pb}(n)$	1	2	5	15	52	203	877	4140	21147

The numbers shown in the table apply only if all parameters are of the same type and there are no imported global variables. Recall that the following factors will reduce the count of necessary procedure bodies:

1. variables of different types cannot be aliases,
2. there are no aliases among imported globals, and
3. imported atomic variables cannot be aliases to array elements passed by reference.

Example 3.2 *Consider a global boolean variable and procedure* `proc3` *that imports it.*

```
var d:bool;
proc3(a:&int, b:&bool, c:&int) imports (d)
```

This procedure requires four bodies — one for the case of no aliases, and three for the following alias combinations.

[ac], [bd], [ac, bd]

However, aliasing between array elements and atomic variables increases the number of necessary procedure bodies.

Example 3.3 *Consider a procedure with the following header.*

```
proc proc4(a[]:&int, b[]:&int, c:&int, d:&int)
```

Besides the alias combination `(a[k:int] alias c alias d)` *there also is a combination* `(a[i:int] alias c, a[j:int] alias d)`*. The number of required procedure bodies is eighteen—one for each of the following alias combinations. (The notation "[a[i]cd]" is shorthand for the aliasing combination* `(a[i:int] alias c alias d)`*.)*

[], [cd], [ab], [ab, cd], [ab, a[i]c], [ab, a[i]d], [ab, a[i]c, a[j]d], [ab, a[i]cd], [a[i]c, [b[j]d]], [a[i]d, [b[j]c]], [a[i]c], [a[i]d], [a[i]c, a[j]d], [a[i]cd], [b[j]c], [b[j]d], [b[i]c, b[j]d], [b[i]cd]

Clearly, for a large number of reference parameters writing all required alternative bodies becomes impractical, but procedures tend to have few parameters of the same type[4], so the expected number of cases is not too big. In our experiments with example programs the number of procedure bodies in most examples is just two.

3.2 Dispatch algorithms

3.2.1 Static dispatch

ACL allows 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 can often be determined statically. As an example, consider two global arrays and a procedure with the following header.

```
array a[5]:  int;
```

[4]In a language with a richer type structure than ACL, the number of parameters of the same type would often be smaller.

```
array b[5]:  bool;
proc proc5 (x[]:&int, y[]:&int, z:&int, t[]:&bool, f:int)
     imports (b)
```

The procedure call

```
call proc5(a, a, a[5], b, a[3])
```

corresponds to the alias combination

```
(x alias y, x[i:int] alias z, t alias b).
```

In such definite cases of aliasing, dispatch to the corresponding procedure body can be done 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 list could be completed using the algorithms described below.

3.2.2 Dynamic dispatch

When static alias analysis is not possible, the concrete combination of aliases among actual parameters will be determined at run-time. The current version of the ACL interpreter uses a simple selection algorithm to compute the run-time aliasing combination for a particular procedure call. Sequential search is then used to find the procedure body that corresponds to the computed aliasing combination. The performance of these algorithms is reasonable when the number of reference parameters and imported global variables is small.

Better performance could be achieved with 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[5].

For the expected, small number of reference parameters, the difference between ACL's current dispatch algorithms and the pattern-matching algorithm is negligible. More important time saving would be achieved by adding flow analysis to take advantage of static dispatching opportunities.

[5] 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.

3.3 Efficiency of ACL dispatch compared to other languages

The necessity of dynamic dispatch could be considered a disadvantage of the multi-body procedures approach. We claim, however, that the efficiency of an ACL program need be no worse when written in another language when dealing with aliases. For example, ACL programs should be no slower than Euclid programs [PHL+77].

This claim is true despite Euclid's static dispatch. To see this, 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). When the proof that there is no aliasing among the reference parameters and imported globals can be carried out statically, ACL would often be able to perform a static dispatch also. If this proof is too involved for the Euclid compiler to discover, then it would insert an assertion to check for aliasing at run-time; the running time for this assertion would be similar to the time needed for dynamic dispatch in ACL. In other cases, the Euclid programmer must write alias analysis code by hand, and can do no better than an ACL compiler could do. Thus, given an equally sophisticated ACL compiler, the efficiency of ACL's dispatch is likely to be no slower than Euclid's, and less error prone, since the responsibility for such dispatch 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. Yet such languages, because they offer no guarantees about disjointness of variables, make static analysis much more difficult than it is in Euclid or ACL.

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 approach 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 the benefits of eliminating aliasing and has call-by-reference. The most important benefit is, again, the greatly increased opportunity for code optimization, due to lack of aliasing.

ACL automates checking that all aliasing combinations have been considered for correctness. This will be useful even for programmers who are not concerned with doing formal program verification.

Writing multiple bodies for a procedure is not too big a burden for programmers, since in practical examples this number is usually small, most often just two.

However, the requirement that one body must be written for each aliaing combination could be relaxed, as described in Section 2.4, while still eliminating aliasing. Along these lines, we believe that a fruitful direction for the research is to add to ACL annotations on formal parameters. Such annotations would allow the automatic generation of some alternative bodies.

As we argued in Section 3.3, the efficiency of programs written in ACL will be no worse than the efficiency of similar programs written in other programming languages. Considering the possibility of compiler optimizations the efficiency of ACL may exceed the efficiency of these other languages.

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 [Hog91] [HLW+92] [Utt95]. Since these works concentrate on other kinds of aliasing, as opposed to parameter aliasing, it would be interesting to combine our ideas with their approaches 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 2.4. 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.

References

[AG98] Ken Arnold and James Gosling. *The Java Programming Language*. The Java Series. Addison-Wesley, Reading, MA, second edition, 1998.

[AVAU86] R. Sethi A. V. Aho and J. D. Ullman. *Compilers. Principles, Techniques and Tools*. Addison-Wesley, 1986.

[BKK+86] 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.

[Cas97] Giuseppe Castagna. *Object-Oriented Programming: A Unified Foundation*. Progress in Theoretical Computer Science. Birkhauser, Boston, 1997.

[Cha92] 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.

[FWH92] Daniel P. Friedman, Mitchell Wand, and Christopher T. Haynes. *Essentials of Programming Languages*. McGraw-Hill Book Co., New York, N.Y., 1992.

[GL80] D. Gries and G. Levin. Assignment and the procedure call proof rules. *TOPLAS*, 2(4):564–579, 1980.

[GR83] Adele Goldberg and David Robson. *Smalltalk-80, The Language and its Implementation*. Addison-Wesley Publishing Co., Reading, Mass., 1983.

[HJW+92] 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.

[HLW+92] 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.

[Hog91] 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.

[HW73] C. A. R. Hoare and N. Wirth. An axiomatic definition of the programming language pascal. *Acta Informatica*, 2(4):335–355, 1973.

[Moo86] 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.

[Mor94] Caroll Morgan. Procedures, parameters and abstraction: separate concerns. In Morgan and Vickers [MV94], pages 47–58.

[MTH90] Robin Milner, Mads Tofte, and Robert Harper. *The Definition of Standard ML*. The MIT Press, Cambridge, Mass., 1990.

[MV94] 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.

[Pae93] Andreas Paepcke. *Object-Oriented Programming: The CLOS Perspective*. The MIT Press, 1993.

[PHL+77] 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.

[RCA+86] 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.

[Str97] Bjarne Stroustrup. *The C++ Programming Language: Third Edition.* Addison-Wesley Publishing Co., Reading, Mass., 1997.

[Utt95] 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`

A Appendix: Concrete Syntax of ACL

The following is the concrete syntax for ACL. Names in *italic* font denote syntax domains. 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 the grammar extend from a dash (--) to the end of a line.

Prog ::= -- *Program*
    Com

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

ProcBody ::= -- *Procedure-Body*
    Com
  | `(` *AliasList* `)` *Com*

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

OVL ::= -- *Overlapping-Location*
    Name
  | *Name* { `[` *Name* `:` *Type* `]` }$^+$

ImpList ::= -- *Import-List*
    Name {`,` *Name*}*

Com ::= -- *Command*
    Loc `:=` *Exp*
  | `if` (*Exp*) { *Com* } `else` { *Com* }
  | `while` `(` *Exp* `)` { *Com* }

```
    | skip
    | Com { ; Com }+
    | call Name Actuals
    | Decl in Com

Formals ::=                                          -- Formal-Parameters
      ( )
    | ( Name{[]}*:[ & ]type { , Name{[]}*:[ & ]type }* )

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

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
    | if ( Exp ) { Exp } else { Exp }
    | Name Actuals                                   -- Function call
    | Loc                                            -- Dereferencing location
    | let Decl in Exp
    | Name                                           -- Identifier expression
    | Name { [ Exp ] }+                              -- Identifier expression

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

NumLit ::=                                           -- Numeral-Literal
      Integer

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

type ::=                                             -- Primitive type
      int | bool
```