ACL - Eliminating Parameter Aliasing with Dynamic Dispatch

Olga Antropova

TR #98-07
June 1998

Keywords: reference parameter aliasing, global variable aliasing, multi-body procedures, dynamic dispatch, static despatch, 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.3.4 [*Programming Languages*] Processors – interpreters; D.3.m [*Programming Languages*] Miscellaneous – dynamic dispatch, multiple dispatch, type systems; F.3.1 [*Logics and Meanings of Programs*] Specifying and verifying and reasoning about programs, logics of programs; F.3.3 [*Logics and Meanings of Programs*] Studies of Program Constructs – functional constructs, type structure.

An ACL implementation is available from
ftp://ftp.cs.iastate.edu/techreports/TR98-07/

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

Part I

ACL - Eliminating parameter aliasing with dynamic dispatch

Abstract. In this article we present a method of eliminating reference parameter aliases. The goal is to allow procedure calls with parameters being aliases, and at the same time guarantee that procedure bodies are alias-free. The method is to automatically dispatch to the correct procedure body based on the particular alias combination among actual parameters. Automating finding the alias combination makes writing verifiable programs verification simpler since code to find the combination is not explicitly present in client programs. The number of necessary procedure bodies is usually small which makes the approach practical. Efficiency of the dispatch is estimated to be no worse then in other languages.

1. Introduction

When a location can be referenced by several names those names are called *aliases*. The presence of aliases and mutation makes it more difficult to write correct programs and to reason about their correctness ([CM88], [HW73]). Some compiler optimizations become impossible in the presence of aliasing([ASU86], p 648), which results is slower executable code.

The most common and obvious source of aliases are alias declarations and pointer variables. Much research has been done to either completely eliminate or to restrict and control these aliases. In this article we concentrate on the other source of aliases, which has received much less attention, but which is common and not less important.

This other source of aliases is the parameters to the procedures and functions in languages, which support call-by-reference. Two kinds of aliasing can happen because of parameter passing. First, the same object may be passed twice as actual parameter; for example, if procedure p takes two reference parameters then the procedure call *p(x, x)* makes the corresponding formals aliased. Second, if a global variable is passed as an actual parameter by reference, then the global and formal become aliases. In Example 1.1 the names x and y in *p1*'s body are aliases, as the type "**&int**" indicates that the parameter is passed by reference.

Example 1.

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

The 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. Consider the example of matrix multiplication *proc mm(a[][]:&int, b[][]:&int, c[][]:&int)*, where the result of multiplication b by c is accumulated in matrix a. If *mm* directly works on a, b, and c (not on copies), then the results of following procedure calls with matrices x and *y: mm(x, x, x)*, *mm(x, x, y)*, *mm(x, y, x)* will all be incorrect.

When aliasing is possible the verification of a procedure's correctness is difficult. It involves separate proofs for all possible aliases 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++ has call by reference. Other object-oriented languages such as Smalltalk and Java, and even mostly-functional languages such as ML and Scheme, manipulate objects indirectly, through references. In such languages assignments as well as parameter passing may cause aliasing.

Aliasing also affects efficiency of programs. Most of the researchers, working on aliasing problems, concentrate on efficient alias analysis for enabling compiler optimizations [REFS?].

There are thus important reasons to investigate languages which prohibit aliasing.

1.1. Related work

Back in 1977 the programming language Euclid was developed [PHLML77], which "... demonstrated that it is possible to completely eliminate aliasing in a practical programming language" ([PHLML77], page 16). The approach the authors took to eliminate aliases resulting from reference parameters to procedures was to prohibit procedure calls when the actual parameters overlap. This includes structured data passed along with a component of it (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 if $i \neq j$. Euclid requires the compiler "to generate a legality assertion to guarantee their distinctness" ([PHLML77], page 14). For global variables, Euclid requires explicit importation of those that are going to be used by a procedure. Like parameters, imported globals should not overlap with the parameters. Pointer variables and pointer assignments are allowed in Euclid, but the pointers are considered to be indices into the "collection" of objects of the same type. Collections "are explicit program variables that act like the "implicit arrays" indexed by pointers" ([PHLML77], p. 14). The same restrictions apply to collections, elements of collections and pointers, when those are passed to the procedure or imported, as to arrays, array elements and subscripts.

Recent work in this direction by Utting extends the idea of collections to object-oriented systems [MU97]. All complex objects (possibly sharing memory locations) can be considered [treated/manipulated/structured] as a set of disjoint collections (local stores) of homogeniuos objects. Local stores are practically the arrays indexed by pointers to objects in those. The proof logic for the arrays can be applied directly for the local stored. For procedure calls, the requirements are similar to those in Euclid: actuals should be non-overlapping.

1.2. Problem with previous approaches

These previous approaches, in the way they treat parameter aliases, have 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 the client code must check for overlaps and call 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. Example 1.2 shows how the additional overlap checking code might look like; the procedures **p2_1**, **p2_2**, **p2_3**, and **p2_4** are variants of *p2* that handle different combination of aliases.

Example 2.
```
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])
```

Note that similar code might be needed in all places where the procedure *p2* is called. Note also that, in general (as shown in Example 1.2), different procedure is needed for each of possible alias combinations. The Example 1.2 shows a simple sequential search algorithm for deciding the alias combination. More efficient algorithms could be implemented, but this complicates client programmer's job even further. For bigger number of reference parameters writing the efficient alias analysis code is a non-trivial task.

The alternative of making copies before (and possibly after) a call is inefficient in many cases and still requires alias analysis.

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 correctness may depend on the order in which the values are copied back, and in many cases it may be impossible to reconcile the desired postconditions for different value-result parameters in the presence of aliasing. Because of this, treatments of call by result or call by value-result "usually" consider passing the same location to multiple result parameters to be "invalid" ([CM88], p. 57). Hence, such a language would have prohibitions on parameter aliasing that are similar to Euclid's.

The following sections describe the new method and the experimental implementation of it - the programming language ACL. We look at the programs in

ACL, discuss the results and implications of the method, and analyze the efficiency of the approach. The conclusion summarizes the results of the experiment, and talks about directions for future work.

2. Prohibiting aliases in procedures

This article presents a 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 pattern of aliasing that occurs dynamically.

2.1. Our approach: dispatch based on aliasing patterns

To prohibit parameter aliasing, our approach requires that the procedure implementation has 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. In the procedure body for a particular alias combination aliased locations should only be referenced through one of their alias names.

To avoid unnecessary alias combinations with global variables, we adopt Euclid's idea of explicit importation of global variables [PHLML77] in procedures. (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, it may depend on the values of expressions. 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 program verification becomes a simpler task compared to other languages (e.g., Euclid [PHLML77]). In Euclid, to make a program correct, the code, similar to the code in the procedure bodies for alias combinations, or alias analysis code should be written at all the points of the procedure calls repeatedly. Besides being a burden for the programmer, it also requires correctness proofs to be made for all of such additional pieces of code, whereas in our approach this additional code is part of a compiler and can be verified once and for all.

To experiment with the idea we implemented the small imperative programming language which we call ACL (Alias Controlling Language). The goals were

1) to find the difficulties of the proposed approach

2) to implement the algorithms for the procedure declaration type-checking and for the dynamic dispatch to the correct procedure body

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

The results of these experiments are described in the following sections. Briefly summarizing them we can say that the static dispatch can be used for most of the procedure calls. Furthermore, the complexity of the dynamic dispatch can be as low as $O(n \cdot \log n)$, where n is the number of the reference parameters. A number of the compiler optimization techniques can be applied to further reduce the dynamic dispatch time.

The requirement that the separate procedure body should be implemented for each possible alias combination makes the number of the procedure bodies exponential in terms of the number of the parameters. In the practical examples this number turns out to be not too large, in fact is seldom exceeds two.

2.2. ACL explained

The grammar of ACL is presented in an appendix A. The language is designed to be small, yet expressive enough to investigate the problem and our approach to solving it.

ACL has integer and boolean literals and variables. Arrays are implemented in order to have structured variables and investigate passing both variable and its element to a procedure. The language has both functions and procedures. Functions do not have side-effects (like all expressions) and return boolean or integer values. Procedures modify the store and do not return values. Both value and reference parameters are allowed to the functions and procedures.

Reference parameters are allowed to functions for the reasons of efficiency. All global variables are visible inside a function body. Parameter aliasing is not a problem in this case since all variables in a function body are read-only.

2.2.1. Procedures

Procedures are the key feature of ACL. A procedure has a formal parameter list, imported global variables list, a main procedure body (case without any aliases) and *alternatives*. The *alternative* is a procedure body which implements the same functionality as main procedure body but for a particular combination of aliases among parameters and imported variables. Each of the alternatives is preceded by a list of lists of aliases. An aliasing combination is a list of lists of aliased names since there may be several independent groups of aliases among parameters and global variables.

There are three factors in ACL that reduce total number of possible alias combinations:

1) value parameters cannot be aliases to any other parameters and to global variables;

2) parameters of different types cannot be aliases to each other, likewise a parameter of one type cannot be alias to global variable of another type;

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

An ACL program is a sequence of declarations followed by a command or a sequence of commands. Consider the following example.

Example 3.

```
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)
```

The program in Example 2.1 declares variable a and procedure *swap*. Recall that the notation *&* means that both parameters to *swap* are reference parameters. The body of the program is a procedure call after the keyword **in**. An alternative procedure body is preceded by a vertical bar | and combination of aliases. For the procedure call in Example 2.1, the second procedure body will be executed.

An array and its element in ACL are considered to be aliases. Example 2.2 implements a procedure which computes sum of the array a and stores the result in b.

Example 4.

```
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 alias b) {
var s:int in
    call sum(a, s, size);
    a[a:b:1] := s}
```

In Example 2.2 note that *size* is a value parameter and thus does not appear in alias lists. The import list is absent which means that no global variables are available in any of the procedure bodies. Parameter b is of the same type as elements of an array a. It is possible for the actual parameter initializing b to be an element of the array passed for a. This requires an alternative procedure body.

An important property of procedures is that in procedure bodies no two names are aliases. To guarantee this, the first name in the alternative's alias list is the only one that is used in the procedure body for all aliased variable names. Observe that all elements of the structured variable can be expressed through its name (e.g., an element of the array A is A[i] for some i), but a structured variable cannot be expressed through an element. It follows that a structured variable, if present in list of aliases, should be placed first in that alias list.

In Example 2.2 the variable b cannot be referenced using an identifier b in the second procedure body where b is an alias to a. Instead b should be expressed through a. Since in this case b is an element of array a, b is *a[i]* for some i. The subscript i is not known at procedure declaration time but is computable at run-time. ACL provides an expression of the form *x:y:n* which computes the n-th dimension's subscript of the array element y in an array x. So *a[a:b:1]* in an Example 2.2 denotes the same location as b.

Whole arrays are easier to deal with. Since the aliased arrays completely overlap, the name of the first array in the alias list can be used instead of the others.

To show how imported global variables would affect the implementation of a procedure we modify the procedure *sum* from the Example 2.2 to use global variable *size*.

Example 5.

```
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 alias b) {
var s:int in
    call sum1(a, s);
    a[a:b:1] := s}
| (b alias size) {
var s:int;
    call sum1(a, s);
    b := s}
in call sum1(x, size)
```

Note that in Example 2.3 the alias combinations *(a alias size)* and *(a alias b alias size)* are not possible since the global variable *size* cannot be an element of any other variable.

Note that in the presence of aliasing sometimes it is impossible to satisfy the procedure's specification post-condition. Consider an example of the procedure `min_max` which finds the minimum and maximum elements of a given array and stores the results in parameters `min` and `max`.

Example 6. *min_max (A[]:&int, min:&int, max:&int)* {

```
...}
| (min alias max)
{ error "precondition unsatisfied"}
...
```

The specification of such procedure should prohibit the *(min alias max)* case by specifying that in the pre-condition. In general the alias analysis code should be present in the client code to avoid violating the pre-condition. ACL forces the programmer to write an alternative procedure body for this case, but that only has the effect of making the procedure have an implementation that is more defensive than necessary.

ACL is typed language, and one of the goals of its type system is to make sure that each procedure body uses only the permitted names (and is thus alias free). The verification of this requirement is done by type-checking each alternative body in a type environment where the types of all aliased variables, except the first one in an alias list (e.g., a in Example 2.3), are changed to be of an alias type (e.g., *alias-of-a).* This makes it impossible to deference such a variable through any names other than the first in the alias list. The typing system does allow the aliased names to appear in index computing expressions (e.g., *a:b:1*). Since expressions do not have any side-effects, we can consider the procedure body, which has passed the type-checker, to be effectively alias-free.

2.3. Alternative procedure bodies: observing common patterns

The observant reader has probably noticed that often alternative procedure bodies implementations follow common patterns. We now present the bigger example of matrix multiplication and discuss the patterns occurring in ACL procedures.

Suppose that a procedure *mm* takes three matrices a, b and c, multiplies b by c and stores the result in a. Example 2.4 presents the definition of *mm* in ACL. The helping procedure, *copyMatrix,* is used by *mm*. For simplicity we assume that both dimension sizes for all matrices are equal.

Example 7.

```
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) {
            b[i][j] := a[i][j];
            j := j + 1};
        i := i + 1}}
| (a alias b ) {skip};

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(a, temp, size);
    call mm(a, temp, c, size)}
| (a alias c) {
array[size][size] temp:int in
    call copyMatrix(a, temp, 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(temp, a, size)}
```

One commonly occurring pattern, observed in alternative bodies of *mm* corresponding to alias combinations *(a alias b)* and *(a alias c)* in Example 2.4, is to copy the aliased variables into locally declared variables, and to call the procedure recursively with new actual parameters. Let us call this pattern *value-pattern* since it follows the *call-by-value* parameter passing mechanism.

The second pattern commonly occurs in the procedures where one of the aliased variable serves as an accumulator of the result of the computation (*matrix multiplication, factorial* and *search* are some examples). This pattern occurs in the last alternative body of an Example 2.4, and the Examples 2.2 and 2.3. 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. In the last procedure body of *mm* in Example 2.4 local variable *temp* is declared, the procedure *mm* is called recursively (*call mm (temp, a, a, size)),* and the result of computation is copied back into *a (call copyMatrix(temp, a, size)*). Let us call this pattern *result-pattern.* This pattern resembles the call-by-result mechanism.

The variation of the *result-pattern* would be a *value-result-pattern* when the initialization of variables is needed for the computation.

In common programming languages for the correctness in presence of aliasing *mm* could be implemented following *value* pattern for the second and third argument or the *result* pattern for the first argument. The important difference is that in these languages the copy of at least one parameter is always made, no matter if the actual parameters are aliases or not. In ACL the copies will be made only if the actual parameters are aliases. This makes the multi-body procedures in the presence of aliasing to be more efficient than their counterparts in other languages.

In the cases when aliasing is known to be harmless (as in *(b alias c)* case in *mm*) the main procedure body can be used for computation. To satisfy the alias-free procedure body requirement the main procedure body is copied to the alternative and the aliased names are substituted by the name first in the alias list. We call this pattern the *substitution-pattern.* The *substitution* pattern also results in more efficient code compared to the common languages since copies of the parameters are not made.

Another common pattern can be observed in *copyMatrix* procedure. In the case of two arrays being aliases, nothing has to be done. The same pattern appears in examples like *swap*, *comparison*, *search.* In all of those case the implementation takes advantage of the fact that the parameters are aliases and the resulting procedure call is very efficient.

The appearance of common patterns indicates that it may be possible to automate the alternative body generation. That is, let the programmer implement the main procedure body and the cases where knowledge of the aliasing combination may be used to advantage, and leave the rest for a tool, possibly with some annotations on what to be done.

3. Counting alias combinations and dynamic dispatch

ACL is implemented as an interpreter in Haskell[HJW92]. In this section we briefly describe the current algorithms and give their complexity. We also discuss the possible improvements and estimate the complexity of the improved algorithms.

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 the property of ACL, as this situation has to be dealt with in all other languages. Recall that in general different procedure bodies are needed for aliasing combinations in Euclid [PHLML77] (see Example 1.2). Also, the number of parameters is usually not too big in practical programming. The exponential number of aliasing cases is thus not a key measure for the practicality of our approach.

Exactly how many procedure bodies must the programmer write? Let us call the list of the reference parameters and the import variables V_{ref}. All possible combinations of aliases among the variables in V_{ref} can be constructed listing all possible partitions of the set of variable names in $V_{ref.}$ Consider the following example:

Example 8. *Let V_{ref} = [a,b,c,d], partitionNames is the function which lists all the possible partitions of a set of names. Then partitionNames [a, b, c, d] will give the following name partitions (In the following lists think of combination of the names (e.g., ab) as aliased variables, single names (e.g., a) as not aliased variables.):*

[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]

By removing the single names from each of the partitions above we get all combinations of aliases in V_{ref}:

[cd] [bc] [bd] [bcd] [ab] [ac] [ad] [ab, cd]

[acd] [abc] [ad, bc] [abd] [ac, bd] [abcd]

Example 3.1 shows that the number of all possible combinations of aliases among four reference parameters (of the same type) is fourteen. In general, number of all possible alias combinations is largest when all elements in V_{ref} are of the same type and there is no more than one imported variable (recall that impoted variables cannot be aliases). An asymptotic upper bound for number of the alternative procedure bodies (C_{pb}) is:

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

The following table shows the number of procedure bodies C_{pb} given by the function *partitionNames* for n reference parameters compared to $n!$.

n	1	2	3	4	5	6	7	8	9
$n!$	1	2	6	24	120	720	5040	40320	362880
C_{pb}	1	2	5	15	52	203	877	4140	21147

The numbers for C_{pb} shown in the table apply only if all parameters are of the same type and there is only one imported global variable of the same type. 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
3) imported atomic variables cannot be aliases to the structured (arrays in ACL) reference parameters.

Example 9. *A procedure proc3 with the following header:*

```
proc3(a:&int, b:&int, c:&int, d:&bool, e:&bool)
```

requires nine procedure bodies - one for the case of no aliases, and eight for the following alias combinations:

[], [abc, de], [abc], [de], [ab], [ac], [bc], [ab, de], [ac, de], [bc, de]

Example 10. *For the declarations*

```
var d:int;
var e:int;
proc proc4(a[]:&int, b:&int, c:&int) imports (d, e)
 { ...}
```

The number of required alternative procedure bodies is eighteen - one for each of the following alias combinations:

[], [abc], [ab], [ac], [bc], [bd], [be], [cd], [ce],[ac, bd], [ac, be], [ab, cd], [ab, ce], [bd, ce], [be, cd], [bcd], [bce], [ab, ac]

Note that in Examples 3.2 and 3.3 fewer than 52 procedure bodies are necessary even though there are five reference parameters and imported variables to both *proc3* and *proc4*. Seventeen procedure bodies does not sound like easy job for a programmer either, but keeping correctness in mind, this reflects the way how he or she must think of implementing the procedure in presence of aliasing (no matter what is the language).

The ACL type-checker constructs the list of all possible aliases combinations among reference parameters and verifies if the procedure body is present for each of those combinations. The efficiency of the involved algorithms was not a major goal since type-checking is static and does not affect the run-time efficiency of a program. Current algorithms work well for reasonable amount of reference parameters.

Clearly, for a large number of reference parameters writing all required alternative bodies become impractical, but because procedures tend to have few parameters expected number of cases is not too big. In our experiments with the example programs for common kinds of the procedures, such as *sort*, *search*, *factorial*, *sum*, *copy*, *swap*, *comparisons* and like, the number of procedure bodies in most of the cases is just two.

Furthermore, often too many parameters indicate that the procedure is designed to do many different things, which is not a good software engineering practice. Excessive usage of the global variables is considered to be a bad programming practice too [WS73]. Multi-body procedures will indirectly push the programmer to think more carefully about the design in terms of what the procedure should accomplish (one task per procedure should be a goal), how many parameters are really needed, how many of them should be passed by reference (remember that the value parameters cannot be aliases). Adding the **const** modifier to the language would further reduce the number of necessary cases. Finally, if the derivation of the alternative procedure bodies can be partly automated, the programmer's job becomes much easier.

3.2. Dispatch algorithms

3.2.1. Static dispatch

ACL is specialized for eliminating parameter aliases. Since other kinds of aliases are also banned, the combination of aliases among actual parameters in the procedure call can be determined statically in many cases. As an example consider the procedure declared as follows

proc proc5 (x[]:&int, y[]:&int, z:&int, t[]:&int, f:int) imports (b) {...}

The procedure call *call proc5(a, a, a[5], b, b[3])* corresponds to the alias combination *(x alias y alias z, t alias b)*. In such definite cases of aliasing dispatch to the corresponding procedure body can be done statically. So no time is lost on the additional computations of aliasing combination or finding the corresponding procedure body at run-time.

The static dispatch is not possible in the case when the actual parameters are the array elements with variable subscripts. For example, in the procedure declaration like *proc proc6(x: & int, y: & int, z: & int)* {...} and the call like *call proc6(A[i], A[j], A[i])* one cannot determine the exact alias combination statically. However one can construct a partial alias list. At the run-time this partial alias list will be completed using the algorithms described below. Furthermore, flow analysis could be employed to make the necessary information available at the point of a procedure call at run-time.

3.2.2. Dynamic dispatch

When static alias analysis is not possible the concrete combination of aliases among actual parameters can be determined at run-time comparing addresses of those. Current version of ACL interpreter implements a simple selection algorithm (analogous to selection sort). In terms of the comparisons the complexity of the algorithm is $O(n^2)$ in the worst case (ironically it is the no-aliases case). The average time complexity is also $O(n^2)$, as for the selection sort. The best case has complexity of $O(n)$ (the case when all the variables are aliases). Here n is the number of reference parameters.

The average time complexity could be improved to $O(n \cdot \log n)$ time when instead of the selection algorithm a binary tree construction is used to analyze the aliasing combination. The tree is constructed in $O(n \cdot \log n)$ time in an average case. The best case, with the complexity $O(n)$, happens if the variables are all aliases. The worst case $O(n^2)$ happens when the list of actuals parameter names from which the binary tree is constructed is in such order, that their addresses are sorted.

After the alias combination for a particular procedure call is computed the corresponding procedure body should be found dynamically. Now we analyze the performance of this search.

An upper bound on the number of the procedure bodies is $n!$, which in turn has an upper bound of n^n (by Stirling's approximation) [CLR90]. The sequential search for the procedures with large number of reference parameters is inefficient. The binary search algorithm will take $\mathrm{O}(\log\ (\mathrm{n}^n)) = \mathrm{O}(n{\cdot}\log n)$ time. (Sorting of the procedure bodies by the alias lists will be done statically.)

Usually the number of parameters to a procedure is not too large. In standard C++ libraries *assert*, *ctype*, *math*, and *stdlib* almost all the functions have one parameter. In the *string* library majority of functions have 2 reference parameters [HR94]. For a small number of reference parameters the difference in the performance of the $\mathrm{O}(n^2)$ and $\mathrm{O}(n{\cdot}\log n)$ algorithms is negligible. The bigger time saving may be achieved using some flow alias analysis techniques in combination with statically known information about aliases.

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 though, that the efficiency of the program written in ACL will be no worse than the efficiency of a similar program in languages such as Euclid [PHLML77].

Though in Euclid the dispatch to the procedures is static, 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 the program at the point of each procedure call. The best efficiency could be achieved in Euclid if this code is written using decision trees (nested if statements). In this case the run-time efficiency of such code is $\mathrm{O}(\log \mathrm{n}^n) = \mathrm{O}(n{\cdot}\log n)$, where n is number of reference parameters. But this is a complexity of ACL's dynamic dispatch. Recall that the decision making code in Euclid is the responsibility of the client code's programmer (whereas in ACL the analysis is done by the interpreter). For large number of the parameters writing the balanced decision tree is a very difficult task, so in practical programming the use of simpler algorithm (like in Example 1.2, where complexity is $\mathrm{O}(\mathrm{n}^n)$) is likely. So in practice the efficiency of ACL's dispatch is likely to exceed Euclid's. Furthermore, since the analysis is moved from the application program to the interpreter (or the compiler) a vast number of optimizations is possible.

We have chosen the Euclid language for comparison with our approach since it presents the extreme example of the separation of alias analysis code and the procedures. Recall that Euclid disallows aliased parameters. In the other languages, which do not have this constraint, the code similar to the alias analysis should be present for correctness either in the client code or in the procedure itself, or the *copy-call-copy-back* pattern should be used in the procedure implementation as a precaution to avoid aliases. So the efficiency of those languages is comparable to the efficiency of ACL. And, of course, common languages do not guarantee freedom of aliasing.

4. Conclusion and future work

It is worthwhile to emphasize again that the aliasing problem is important and deserves more attention of researchers. The knowledge that code is alias free is beneficial to both compiler optimizations and arguments about correctness. Summarizing the results of our experiment with the proposed approach we can say that ACL achieves the goal of eliminating parameter aliases.

The ACL requirement to implement the procedure for all aliases combination among the parameters has an important advantage of making verification simpler (as discussed in Section 2.1). From the point of view of practical programming it simplifies writing client code programs.

Writing multiple bodies in procedure implementation turns out to be not too big a burden for a programmer since in practical examples this number is usually small, most often just two.

As we argued in Section 3.3, with dynamic dispatch for procedures, the efficiency of programs written in ACL will be no worse than the efficiency of similar programs written in other programming languages (even those that use static dispatch). Considering the possibility of compiler optimizations the efficiency of ACL may exceed the efficiency of these other languages.

Note that approach presented in this article does not require the complete absence of aliases becides the parameter aliases. For example, reference variables as found in C++ could be handled. Though, in the presence of other kinds of aliasing, procedure bodies cannot be guaranteed to be alias-free, since there will be a possibility to introduce aliases in these other ways.

ACL is a small experimental language which investigates the basic implications of the idea of dynamic dispatch and multi-body procedures. It will be interesting to study how the idea would apply to languages that operate more complex objects. One of the directions for future research would be to incorporate ACL's mechanisms into existing languages like Euclid [PHLML77].

Another direction of the future work is to investigate the possibility of applying these ideas to object-oriented languages. Recently several works appeared in the area of creating the object-oriented languages that deal with aliasing [MU97], [HLWCH92], [JH91]. Since those works concentrate on the aliasing of other kinds than the parameter aliasing, it seems interesting to research if it is possible to connect their and our approaches in a single language.

An interesting direction for the research is to combine our ideas of eliminating parameter aliases with the call-by-value-result mechanism. This might allow an automation of the alternative procedure body generation and even the dynamic alternative procedure body generation, which, considering the number of possible aliasing combinations, is very attractive.

References

[PHLML77] G.J. Popek, J. J. Horning, B. W. Lampson, J. G. Mitchel, and R. L. London. *Notes on the Design of Euclid.*, SIGPLAN Notices, 12(3):11-18, March 1977. Proceedings of an ACM Conference on Language Design for Reliable Software.

[GL80] D. Gries and G. Levin. *Assignment and the Procedure Call Proof Rules.* ACM TOPLAS, 2(4): 564-579, 1980.

[MU97] M. Utting. *Reasoning about Aliasing.* Formal Aspects of Computing, 3:1-15, 1997.

[CM88] C. Morgan. *Procedures, Parameters and Abstractions: Separate Concepts.*,in C. Morgan and T. Vickers (Eds.). *On the Refinement Calculus.*, Springler-Verlag., 1992, p57.

[HW73] C. A. R. Hoare and N. Wirth. *An Axiomatic Definition of Programming Language Pascal.*, Acta Informatica, 2(4): 335-355, 1973

[WS73] W. Wulf and M. Shaw. *Global Variable Considered Harmful*, SIGPLAN, 8 (2):28-34, Feb. 1973

[HLWCH92] J. Hogg, D. Lea, A. Wills, D. deChampeaux and R. Holt. *The Geneva Convention on the Treatment of the Object Aliasing.*, OOPS Messenger, 3(2):11-16, Apr. 1992.

[JH91] J. Hogg. *Islands: Aliasing Protection in Object-Oriented Languages.*, in Proceedings of OOPSLA '91, pp. 271-285, Nov. 1991.

[ASU86] A. V. Aho, R. Sethi and J. D. Ullman. *Compilers. Principles, Techniques and Tools.*, Addison-Wesley Publishing Company, 1986.

[FWH92] D. P. Friedman, M. Wand and C. T. Hayness. *Essentials of Programming Languages.*, MIT Press, 1992

[HR94] M. R. Headington and D.D. Riley. *Data Abstraction and Structures Using C++.*, HEATH, 1994.

[CLR90] T. H. Cormen, C. E. Leiserson and R. L. Rivest. *Introduction to Algorithms.*, MIT Press, 1990.

[HJW92] P. Hudak, S. R. Jones, P. Wadler and others. *Report on the Programming Language Haskell: A Non-strict, Purely Functional Language, version 1.2.*, SIGPLAN, 27(5), May 1992.

Appendix A: The Abstract Syntax Grammar for ACL

The abstract grammar is presented in Haskell[HJW92] notation.

```
type C = Command
type E = Expression
type N = Numeral
type P = Program
type D = Declaration
type B = Boolean
type PB = ProcBody
type IL = ImportList
type L = Location

data Name = Ident String

data Program = Prog C
data ProcBody = Body C | Alt AliasList C
data Declaration = Var Name PrimitiveAttrib E
      | Arr Name [Size] PrimitiveAttrib E
      | D `DSemi` D
      | Fun Name Formals PrimitiveAttrib E
      | Proc Name Formals IL PB [PB]

type AliasList = [[Name]]
type ImportList = [Name]

data Command = Assign L E
      | If E C C
      | While E C
      | Skip
      | C `Semi` C
      | Call Name Actuals
      | D `In` C
```

```
type Size = E
type Formals = [(Name, PrimitiveAttrib)]
type Actuals = [E]

data Expression = Num N
      | Boolval B
      | Array [E]
      | E ‘Plus‘ E | E ‘Mult‘ E | E ‘Minus‘ E | E ‘Div‘ E
      | Not E | E ‘Or‘ E | E ‘And‘ E | E ‘Equals‘ E | E ‘Lt‘ E
      | IfE E E E
      | App Name Actuals
      | Deref L
      | Let D E
      | Varref Name
      | ArrRef Name [E]
      | Index Name Name E

data Location = ID Name | IDElem Name [E]
type Numeral = Integer
type Env = [(Name, Integer)]
type Boolean = Bool
```

Part II

Soundness of ACL's Type System

1. Type Soundness

In this section we prove the soundness of the ACL type system

We use the following notation $\|$phrase$\|$ denote the meaning of the phrase, π is a static type assignment for the identifiers, $\pi \vdash \mathrm{U}{:}\theta$ means that the phrase U is typed in a type environment π and the type of U in π is θ. Env_π is a set of run-time environments that are consistent with π.

We will use the Haskell list notation to represent the types of ordered sequences. [T] denotes the set of ordered sequences of the items of type T.

The type-assignment π is modeled as a finite set of pairs $(\mathrm{I}_j{:}t_j)$, where $\mathrm{I}_j \in$ Identifier, t_j is the type of I_j and $\mathrm{I}_j \neq \mathrm{I}_k$ for $j \neq k$. *Identifier* is the set of the identifiers. The run-time environment is modeled as the set of the pairs $(\mathrm{I}_j = v_j)$, where $\mathrm{I}_j \in$ Identifier, v_j is the value to which I_j is bound in the environment *env* and $\mathrm{I}_j \neq \mathrm{I}_k$ for $j \neq k$. The value to which the identifier is bound in the environment is a *DenotableValue*. DenotableValue is a disjoint union:

DenotableValue = Location
| [Integer] $\times$ [Location]
| (Env $\times$ (Store $\rightarrow$ [ExpressibleValue] $\rightarrow$ ExpressibleValue))
| (Env $\times$ (Store $\rightarrow$ [ExpressibleValue] $\rightarrow$ Store)), where

Store is a sequence of *locations*; *location* is a cell capable of holding a *Storable-Value* (explained later) and denoted by the positive integer which is the sequential number of the location in a Store.

When we write $v \in$ Location we mean that v is in the Location part of the denotable values. $v \in$ Location denotes a variable, $v \in$ [Integer]$\times$[Location] denotes an array, for the last two subsets v denotes a function or a procedure correspondingly.

The *ExpressibleValue* is the result of the expression evaluation. Expressible-Value is a disjoint union:

ExpressibleValue = Integer
| Boolean

| [Integer] × [Integer]
| [Integer] × [Boolean]
| Location
| [Integer] × [Location]

These represent respectively integers, booleans, multi-dimensional arrays of integers, multi-dimensional arrays of booleans, locations or the denotable values of arrays. The last two subsets of the ExpressibleValue are used to accommodate the call-by-reference parameter passing mechanism.

Definition: An environment *env is consistent with type assignment* π,written env $\in Env_\pi$, when for all I $\in$ Identifier, if (I:θ) $\in \pi$ then (I $= v$) $\in$ *env* and $v \in \|\theta\|$.

In the type system we use the union-minus operation $\bar{\cup}$ for overriding one finite function with another (as in block structure). It is defined as follows

Definition: For the type-assignments π_1 and π_2

$$\pi_1 \bar{\cup} \pi_2 = \pi_2 \cup (\pi_1 - \{(I:v) \ | \ (I:w) \in \pi_2 \ and \ (I:v) \in \pi_1\})$$

.

The union-minus operation for the environments is defined similarly.

Lemma 1.1. For $\mathrm{env}_1 \in \mathrm{Env}_{\pi_1}$ and $\mathrm{env}_2 \in \mathrm{Env}_{\pi_2}$

$$\mathrm{env}_1 \bar{\cup} \mathrm{env}_2 \in \mathrm{Env}_{\pi_1 \bar{\cup} \pi_2}.$$

Definition: The typing system of a language is *sound* when for all well-formed syntax phrases U in the language, if the type of the phrase U is θ then the meaning of U, is in the meaning of θ. That is

$$\|\pi \vdash \mathrm{U}:\theta\| \in \ \mathrm{Env}_\pi \rightarrow \|\theta\|$$

A syntax phrase is the syntax tree in ACL.

Definition: A *well-formed syntax tree* is the one for which the type attributes can be attached to all of its nonterminals.

The *store* is modeled as the sequence of the locations each of which can hold the *StorableValue.* The StorableValue is a disjoint union:

$$\text{StorableValue} = \text{Integer} \mid \text{Boolean}$$

Whenever a store is needed for the computation of the meaning of the phrase we compute the meaning of the phrase in the store, which is consistent with the environment and type-assignment.

Definition: The store st is *consistent with the type-assignment* π and the environment env (written $st \in \mathrm{Store}_{\pi,\ env}$) if for all (I:θ) $\in \pi$ the following holds:

if $\theta = \tau$-*loc* and (I $=$ **l**) $\in env$, then $lookupStore(\mathbf{l},\ st) \in \|\tau\|$,

if θ $=$ n-τ-*loc-arr* $\in$ π and (I $=$ (dims,**locs**)) $\in$ env then for all l $\in$ **locs** $lookupStore(\mathrm{l},\ st) \in \|\tau\|$,

if $\theta = \pi_1$-θ-*list*-τ-*fun* and (I $= (env_1,$ f)) $\in env$, then st $\in \mathrm{Store}_{\pi_1,\ env_1}$,

if $\theta = \pi_1$-θ-*list-proc* and (I $= (env_1,$ p)) $\in env$, then st $\in \mathrm{Store}_{\pi_1,\ env_1}$.

We also say that $\perp$ is consistent with π and env written st $\in (\mathrm{Store}_{\pi,env})_\perp$

Lemma 1.2. For $\mathrm{env}_1 \in \mathrm{Env}_{\pi_1}$, $\mathrm{env}_2 \in \mathrm{Env}_{\pi_2}$ and a store st such that $st \in \mathrm{Store}_{\pi_1,\ env_1}$and $st \in \mathrm{Store}_{\pi_2,\ env_2}$

$$st \in \mathrm{Store}_{\pi_1 \dot\cup \pi_1,\ env_1 \dot\cup env_2}.$$

Lemma 1.3. For a type assignment π, an environment $env \in \mathrm{Env}_\pi$ and store $st \in \mathrm{Store}_{\pi,\ env}$, for any $\pi_1 \subseteq \pi$ and $env_1 \subseteq env$ such that for all (I:θ) $\in \pi_1$ (I $=$ v) $\in env_1$:

$$st \in \mathrm{Store}_{\pi_1,\ env_1}.$$

The type system also uses the disjoint-union operation which is defined as follows:

Definition: For the type-assignments π_1 and π_2

$$\begin{aligned}\pi_1 \dot\cup \pi_2 \;\; = \;\; & \pi_1 \cup \pi_2 \text{ if } \{\mathrm{I} \mid (\mathrm{I}{:}\theta_1) \in \pi_1\} \cap \{\mathrm{I} \mid (\mathrm{I}{:}\theta_2) \in \pi_2\} = \emptyset; \\ & \text{else it is undefined.}\end{aligned}$$

In the proofs of soundness we will show that whenever the environment and the store are changed as the result of the computation of the meaning of the phrase, the consistency between environment and type-assignment and between store, environment and type-assignment is preserved.

1.1. Type attributes and their meanings

The type attributes of the syntactic phrases in ACL are as follows:

$\tau = int \mid bool$

$\delta = \tau \;\mid\; \tau$-*loc* $\mid$ n-θ-*arr*

$\theta = \delta \mid \delta$-*exp* $\mid$ *comm* $\mid$ π-*dec* $\mid$ θ-*list* $\mid$ π-θ-τ-*fun* $\mid$ π-θ-*proc* $\mid$ η-δ-*alias* $\mid$ δ-*ref*

$\pi = \{(\mathrm{j} : \theta_j)\}_{j \in J}$, J is a finite set, J $\subseteq$ Identifier,
where $\eta \in$ Identifier; n = int-exp-list.

The meaning of these type attributes is as follows. Note that symbol $\perp$ means undefined value (or bottom), $\|Type\|_\perp = \{\perp\} \cup \|Type\|$.

$\|int\| =$ Integer
$\|bool\| =$ Boolean
$\|\tau\text{-}loc\| =$ Location, where Location $= \{\mathrm{l} \mid \mathrm{l} \in \text{Integer}, \mathrm{l} \geq 0\}$.
$\|n\text{-}\theta\text{-}arr\| = [\text{Integer}] \times \|\theta\text{-}list\|$
$\|\delta\text{-}exp\| = \text{Store} \to \|\delta\|_\perp$
$\|\delta\text{-}ref\| = \|\delta\|$
$\|\theta\text{-}list\| = [\|\theta\|]$
$\|\pi\text{-}\theta\text{-}list\text{-}\tau\text{-}fun\| = (\text{env: } \|\pi\| \times$
$\bigcup_{\pi \in TypeAssignment} \bigcup_{env \in Env_\pi} (\text{Store}_{\pi,env} \cap \text{Store}_{\pi_1,env_1}) \to$
$\bigcup_{\theta^c\text{-}list \text{ corresponds to } \theta\text{-}list} \|\theta^c\text{-list}\| \to \|\tau\|_\perp)$,

where *env* and π are the declaration time environment and type-assignment, and env_1 and π_1 are the run-time environment and type-assignment.

$\|\pi\text{-}\theta\text{-}list\text{-}proc\| = (\text{env: } \|\pi\| \times$
$\bigcup_{\pi \in TypeAssignment} \bigcup_{env \in Env_\pi} (\text{Store}_{\pi,env} \cap \text{Store}_{\pi_1,env_1}) \to$
$\bigcup_{\theta^c\text{-}list \text{ corresponds to } \theta\text{-}list} \|\theta^c\text{-list}\| \to$
$(\text{Store}_{\pi,env} \cap \text{Store}_{\pi_1,env_1})_\perp)$,

where *env* and π are the declaration time environment and type-assignment, and env_1 and π_1 are the run-time environment and type-assignment.

$\|\eta\text{-}\delta\text{-}alias\| = \|\delta\|$
$\|comm\| = \bigcup_{\pi \in TypeAssignment} \bigcup_{env \in Env_\pi} (\text{Store}_{\pi,env})_\perp \to (\text{Store}_{\pi,env})_\perp$

$\|\{(\mathrm{j} : \theta_j) \mid \mathrm{j} \in \mathrm{J}, \mathrm{J} \text{ is a finite set, } \mathrm{J} \subseteq \text{Identifier}\}\| = \left\{(\mathrm{j} = \|\theta_j\|)_{\mathrm{j} \in \mathrm{J}}\right\}_\perp$
$\|\pi\text{-}dec\| = \bigcup_{env \in Env_\pi} \text{Store}_{\pi,env} \to$
$(\text{env}_1\text{: } \|\pi_1\| \times (\text{Store}_{\pi,env} \cap \text{Store}_{\pi_1,env_1}))_\perp$

Note that in the function and procedure types θ-*list* is the list of types of the formal parameters. In the second argument of the meaning of those types θ^c-*list* is the list of types of the actual parameters. It is any list which corresponds to the θ_1-*list*. The correspondence of the lists θ_1 and θ_2 is defined as follows:

Definition: For all k such that $1 \leq k \leq length\,(\theta_1)$:
list θ_1 and θ_2 correspond if $(length\,(\theta_1) = length\,(\theta_2))$ and
for $\mathrm{T}_k \in \theta_1$ and $\mathrm{J}_k \in \theta_2$:

$((\mathrm{T}_k = \tau\text{-}loc)$ and $((\mathrm{J}_k = \tau\text{-}loc\text{-}\mathrm{exp})$ or $(\mathrm{J}_k = \tau\text{-}exp)))$
or $((\mathrm{T}_k = \tau\text{-}loc\text{-}ref)$ and $(\mathrm{T}_k = \tau\text{-}loc\text{-}exp))$
or $(((\mathrm{T}_k = n\text{-}\tau\text{-}loc\text{-}arr)$ or $(\mathrm{T}_k = n\text{-}\tau\text{-}loc\text{-}arr\text{-}ref))$ and $(\mathrm{J}_k = n\text{-}\tau\text{-}loc\text{-}arr\text{-}exp))$

1.2. Soundness proofs

Theorem: The typing rules of ACL are sound. That is, for the syntactic categories Identifier, Location, Expression, Command and Declaration the following holds:

For all well formed syntax phrases I $\in$ Identifier and env $\in \mathrm{Env}_\pi$

$$\|\pi \vdash \ \mathrm{I} : \theta\| \ \mathrm{env} \in \|\theta\|\,.$$

For all well formed syntax phrases L $\in$ Location, env $\in \mathrm{Env}_\pi$ and st $\in \mathrm{Store}_{\pi,\, env}$

$$\|\pi \vdash \ \mathrm{L}{:}\tau\text{-}loc\text{-}exp\| \ \mathrm{env\ st} \in \|\tau\text{-}\mathrm{loc}\|_\perp\,.$$

For all well formed syntax phrases E $\in$ Expression, env $\in \mathrm{Env}_\pi$ and st $\in \mathrm{Store}_{\pi,\, env}$

$$\|\pi \vdash \mathrm{E}{:}\tau\text{-}exp\| \ \mathrm{env\ st} \in \|\tau\|_\perp\,.$$

For all well formed syntax phrases C $\in$ Command, env $\in \mathrm{Env}_\pi$ and st $\in \mathrm{Store}_{\pi,\, env}$

$$\|\pi \vdash \mathrm{C}{:}comm\| \ \mathrm{env\ st} \in \Big(\mathrm{Store}_{\pi,\, env}\Big)_\perp\,.$$

For all well formed syntax phrases D $\in$ Declaration, env $\in \mathrm{Env}_\pi$ and st $\in \mathrm{Store}_{\pi,\, env}$

$$\|\pi \vdash \ \mathrm{D}{:}\pi_1\text{-}dec\| \ \mathrm{env\ st} \in \Big(\mathrm{env}_1{:}\ \ \|\pi_1\| \times \Big(\mathrm{Store}_{\pi,\, env} \cap \ \mathrm{Store}_{\pi_1,\, env_1}\Big)\Big)_\perp\,.$$

Proof: The proof is by the structural induction on the typing rules for all the abstract syntax phrases of ACL. Since the syntactic categories Declaration, Expression, Command and Location are mutually recursive the inductive proof is simultaneous for all the categories. That is, when proving soundness of the typing rule for the phrase from one syntactic category we assume that the inductive hypothesis hold for the subphrases even if those belong to the other syntactic categories.

In the following let π be a type-assignment and let env $\in \mathrm{Env}_\pi$ and st $\in \mathrm{Store}_{\pi,\, env}$ be given.

1.2.1. Identifier

In Semantics *Ident str* is evaluated in env

$\|\pi \vdash \textit{Ident str} : \theta\|$ env $= \textit{lookup}(\textit{str}, \text{env})$,

where $\textit{lookup} :: (\textit{Identifier}, \textit{Envirionment})_\perp \to \textit{DenotableValue}_\perp$

$\textit{lookup}(\textit{str}, \text{env})$ is defined (not $\perp$) and is in $\|\theta\|$ by the definition of $\pi-$ env consistency.

Here and in further in the proofs we use Haskell notation for function types: ($\textit{lookup} :: (\textit{Identifier}, \textit{Envirionment})_\perp \to \textit{DenotableValue}_\perp$),where symbol '*::*' separates the name of the function (here *lookup*) from the types of its parameters and the return type; all types following '*::*' except the last one are parameter types, the last one is the return type of the function; all parameter types and the return type are separated by arrows '$\to$'.

1.2.2. Location

Recall that we need to prove that for all well formed syntax phrases L $\in$ Location, env $\in \text{Env}_\pi$ and st $\in$ Store $_{\pi,\ env}$

$$\|\pi \vdash \text{ L:}\tau\textit{-loc-exp}\| \text{ env st} \in \|\tau\text{-loc}\|_\perp .$$

Variable The phrase L is *ID name,* where *name* $\in$ Identifier.

By the inductive hypothesis $\|\pi \vdash \textit{name} : \tau\textit{-loc}\|$ env $\in \|\tau\textit{-loc}\|$. In Semantics module:

$\|\pi \vdash \textit{ID name} : \tau\textit{-loc-exp}\|$ env st $= \|\pi \vdash \textit{name} : \tau\textit{-loc}\|$ env.

Array element The phrase L is *IDElem name exps,* which denotes an array element, where *name* is the array name and *exps* is the list of the subscripts of the element. By the inductive hypothesis we have:

$\|\pi \vdash \textit{name} : n\text{-}\tau\textit{-loc-arr}\|$ env $\in$ [Integer] $\times [\|\tau\textit{-loc}\|] =$ [Integer] $\times$ [Location] ,

$\|\pi \vdash \textit{exps} : \textit{int-exp-lsit}\|$ env $\in [\|\textit{int-exp}\|] =$ Store $\to$ $[\text{Integer}]_\perp$.

In the semantics we have two cases depending on the validity of the subscripts at the run-time. Function $\textit{validate} :: [\textit{Integer}]_\perp \to [\textit{Integer}]_\perp \to \textit{Boolean}_\perp$ checks if the subscripts are valid. Let v = *validate* **dims vals**, where

(**dims, locs**) $= \|\pi \vdash \textit{name} : n\text{-}\tau\textit{-loc-arr}\|$ env st $\in$ [Integer] $\times$ [Location] ,

vals $= \|\pi \vdash \textit{exps} : \textit{int-exp-list}\|$ env st $\in [\text{Integer}]_\perp$

In case v = False in semantics:

$\|\pi \vdash \textit{IDElem name subs} : \tau\textit{-loc-exp}\|$ env st $= \perp \in \text{Location}_\perp$.

In case v = True the meaning of *IDElem name exps* is computed as follows:

$\|\pi \vdash$ *IDElem name subs* : τ*-loc-exp*$\|$ env st = *nthElem* **locs pos** , where

nthElem :: $[Location]_\perp \to Integer_\perp \to Location_\perp$,

locs $\in$ [Location] by the inductive hypothesis,

pos = (*computeLoc* **dims vals**) $\in$ Integer, and is an offset of the element in the list of locations, where *computeLoc* :: $[Integer]_\perp \to [Integer]_\perp \to Integer_\perp$, **dims** $\in [Integer]$ and **vals** $\in [Integer]_\perp$ by inductive hypothesis.

Thus *nthElem* **locs pos** $\in$ $\text{Location}_\perp$.

So in both cases:

$\|\pi \vdash$ *IDElem name subs* : τ*-loc-exp*$\|$ env st $\in$ $\text{Location}_\perp$.

1.2.3. Expression

We need to prove that for all well formed syntax phrases E $\in$ Expression, env $\in$ Env_π and st $\in$ Store $_{\pi,\ env}$

$$\|\pi \vdash \text{E}{:}\tau\text{-}exp\| \text{ env st} \in \|\tau\|_\perp .$$

Literals In semantics: $\|\pi \vdash$ *Num n* : *int-exp*$\|$ env st = *n*, and by the syntax n $\in$ Integer.

Case *Bool b* of boolean literal is analogous.

Array literal In semantics module

$\|\pi \vdash$ *Array e* : *n-*τ*-arr-exp*$\|$ env st = (**sizes**, $\|\pi \vdash$ **lse** : τ*-exp-list*$\|$ env st), where

sizes = *getSizse* (*Array e*), where *getSizes* :: $Expression_\perp \to [Integer]_\perp$,

lse = *flatten* (*Array e*), where *flatten* :: $Expression_\perp \to [Expression]_\perp$,

$\|\pi \vdash$ **lse** :τ*-exp-list*$\|$ env st $\in [\|\tau\|]_\perp$ by inductive hypothesis.

So, by the inductive hypothesis and by the types of the functions used in computation of the meaning:

$\|\pi \vdash$ *Array e* : *n-*τ*-arr-exp*$\|$ env st $\in$ ([Integer] $\times$ $[\|\tau\|])_\perp$.

Binary arithmetic operations The meaning of the phrase *Plus e1 e2* is computed using the function *plus*

(*plus* :: $Integer_\perp \to Integer_\perp \to Integer_\perp$) as follows:

$\|\pi \vdash$ *Plus e1 e2* : *int-exp*$\|$ env st =

plus ($\|\pi \vdash$ *e1* : *int-exp*$\|$ env st, $\|\pi \vdash$ *e2* : *int-exp*$\|$ env st).

The inductive hypothesis is that:
$\|\pi \vdash e1 : int\text{-}exp\|$ env st $\in$ Integer$_\perp$ and
$\|\pi \vdash e2 : int\text{-}exp\|$ env st $\in$ Integer$_\perp$.
Thus $\|\pi \vdash Plus\ e1\ e2 : int\text{-}exp\|$ env st $\in$ Integer$_\perp$.

The cases of *Mult e1 e2, Sub e1 e2, Div e1 e2* are analogous. The case of division is a little different because the result of division by zero is $\perp \in$ Integer$_\perp$. But the *divide* function gives the result $\perp$ in this case, so
$\|\pi \vdash$ Div e1 e2: $int\text{-}exp\|$ env st $\in$ Integer$_\perp$.

Logics operations We show the proof for the phrase *Not e.* The proofs for the cases of *e1 Or e2, e1 And e2, e1 Lt e2* are similar to *Not e* and to the proofs for the binary arithmetic operations.

The meaning of *Not e* is computed using the function *not* (*not* :: $Boolean_\perp \rightarrow Boolean_\perp$) :
$\|\pi \vdash Not\ e : bool\text{-}exp\|$ env st = *not* ($\|\pi \vdash e : bool\text{-}exp\|$ env st). The conclusion follows from the inductive hypothesis: $\|\pi \vdash e : bool\text{-}exp\|$ env st $\in$ Boolean$_\perp$ and the type of function *not*.

Equality test For the syntactic phrase *Equals e1 e2* typing requires that both *e1* and *e2* be of the type *τ-exp,* where $\tau \in \{int, bool\}$, and τ is the same for both *e1* and *e2*. By the inductive hypothesis
$\|\pi \vdash e1 : \tau\text{-}exp\|$ env st $\in \|\tau\|_\perp$, and $\|\pi \vdash e2 : \tau\text{-}exp\|$ env st $\in \|\tau\|_\perp$.

In the semantics we compute the meanings of *e1* and *e2.* Let them be M1 and M2 correspondingly.

Case M1, M2 $\in$ Integer$_\perp$:
$\|\pi \vdash Equals\ e1\ e2 : bool\text{-}exp\|$ env st = *equalint* M1 M2, where
$equalint :: Integer_\perp \rightarrow Integer_\perp \rightarrow Boolean_\perp$.

Case M1, M2 $\in$ Boolean$_\perp$:
$\|\pi \vdash Equals\ e1\ e2 : bool\text{-}exp\|$ env st = *equalbool* M1 M2, where
$equalbool :: Boolean_\perp \rightarrow Boolean_\perp \rightarrow Boolean_\perp$.

The result follows from the types of the functions *equalint, equalbool* and the inductive hypothesis.

Dereferencing location The prase is *Deref loc*, where *loc* $\in$ Location. By the inductive hypothesis
$\|\pi \vdash loc : \tau\text{-}loc\text{-}exp\|$ env st $\in$ Location$_\perp$. In the semantics module:
$\|\pi \vdash Deref\ loc : \tau\text{-}exp\|$ env st = *lookup* ($\|\pi \vdash loc : \tau\text{-}loc\text{-}exp\|$ env st, st)

Since store st $\in \text{Store}_{\pi,env}$, the location ($\|\pi \vdash loc : \tau\text{-}loc\text{-}exp\|$ env st) denotes the storage cell containing τ. It follows that *lookup* ($\|\pi \vdash loc : \tau\text{-}loc\text{-}exp\|$ env st, st)$\in \|\tau\|$.

Local declarations We need to show that $\|\pi \vdash \textit{Let decls e} : \delta\text{-}exp\|$ env st $\in \|\delta\|_\perp$. In semantics module the meaning of *Let decls e* is computed as follows:

$\|\pi \vdash \textit{Let decls e} : \delta\text{-}exp\|$ env st $= \|\pi\bar{\cup}\pi_1 \vdash e : \delta\text{-}exp\|$ (env $\bar{\cup}\mathbf{env}_1$) $\mathbf{st}_1$, where $(\mathbf{env}_1, \mathbf{st}_1) = \|\pi \vdash decls : \pi_1\text{-}dec\|$ env st.

By the inductive hypothesis for the declarations *decls:*

$(\mathbf{env}_1, \mathbf{st}_1) \in \left(\mathbf{env}_1\!: \ \|\pi_1\| \times \left(\text{Store}_{\pi,\, env} \cap \ \text{Store}_{\pi_1,\, env_1}\right)\right)_\perp$. This means that $\mathbf{env}_1 \in \text{Env}_{\pi_1}$, and $\mathbf{st}_1 \in \text{Store}_{\pi,\, env}$ and $\mathbf{st}_1 \in \text{Store}_{\pi_1,\, env_1}$. By the lemmas 1.1 and 1.2 it follows that (env $\bar{\cup}\mathbf{env}_1$) $\in \text{Env}_{\pi\bar{\cup}\pi_1}$ and $\mathbf{st}_1 \in \text{Store}_{\pi\ \bar{\cup}\ \pi_1,\ env\ \bar{\cup}\ env_1}$. Then by the inductive hypothesis for the subexpression e:

$\|\pi\bar{\cup}\pi_1 \vdash e : \delta\text{-}exp\|$ (env $\bar{\cup}\mathbf{env}_1$) $\mathbf{st}_1 \in \|\delta\|_\perp$.

Identifier expressions We have two identifier expressions: *Varref name* and *ArrRef name subs.* For *Varref name* in semantics $\|\pi \vdash \textit{Varref name} : \delta\text{-}exp\|$ env st $= \|\pi \vdash name : \delta\|$ env, and this is in $\|\delta\|_\perp$by the inductive hypothesis for the identifier.

For the *ArrRef name subs* the proof is similar to the proofs for the *Varref name* and array element location (see 1.2.2 part for *IDElem name exps*).

Indexof expression In the phrase *Index n1 n2 e, n1* is the name of the array, *n2* is a name of the variable which is an alias to one of the array *n1* locations, *e* is an integer expression denoting the dimension for which we want to know the index of *n2* in *n1.*

We must show that $\|\pi \vdash \textit{Index n1 n2 e} : int\text{-}exp\|$ env st $\in \text{Integer}_\perp$. The proof relies on the inductive hypothesis for *n1*, *n2* and *e.* These are that:

$\|\pi \vdash n1 : n\text{-}\tau\text{-}loc\text{-}arr\|$ env $= (\mathbf{dims}, \mathbf{locs}) \in [\text{Integer}] \times [\text{Location}]$,

$\|\pi \vdash n2 : \tau\text{-}loc\text{-}n1\text{-}alias\|$ env $= \mathbf{l} \in \text{Location}_\perp$, and

$\|\pi \vdash e : int\text{-}exp\|$ env st $= \mathbf{d} \in \text{Integer}_\perp$.

The meaning of *Index n1 n2 e* is computed as follows:

$\|\pi \vdash \textit{Index n1 n2 e} : int\text{-}exp\|$ env st $=$ *computeIndex* **offset d dims**,where

offset $=$ *getOffset* **locs l**,

$getOffset :: [Location]_\perp \to Location_\perp \to Integer_\perp;$

$computeIndex :: Location_\perp \to Integer_\perp \to [Integer]_\perp \to Integer_\perp.$

So, by the inductive hypothesis and the types of the functions *computeIndex* and *getOffset:*

$\|\pi \vdash \textit{Index n1 n2 e} : \textit{int-exp}\|$ env st $\in$ Integer$_\perp$.

Function application In the phrase *App name acts, name* is the function name and *acts* is the list of actual parameters. We show that

$\|\pi \vdash \textit{App name acts} : \tau\textit{-exp}\|$ env st $\in \|\tau\|_\perp$.

In semantics we compute the meaning of the subphrases and we know their types by the inductive hypothesis. Let

(env$_1$,**f**) $= \|\pi \vdash \textit{name} : \pi_1\text{-}\theta_1\textit{-list-}\tau\textit{-fun}\|$ env $\in \|\pi_1\text{-}\theta_1\textit{-list-}\tau\textit{-fun}\| =$ (env$_1$: $\|\pi_1\| \times$

$\bigcup_{\pi \in TypeAssignment} \bigcup_{env \in Env_\pi} (\text{Store}_{\pi,env} \cap \text{Store}_{\pi_1,env_1}) \rightarrow$

$\bigcup_{\theta_1^c\textit{-list} \text{ corresponds to } \theta_1\textit{-list}} \|\theta_1^c\text{-list}\| \rightarrow \|\tau\|_\perp)$

vals $= \|\pi \vdash \textit{acts} : \theta_2\textit{-list}\|$ env st $\in \|\theta_2\textit{-list}\|$.

By the typing requirement we also know that θ_2*-list* corresponds to θ_1*-list*.

The meaning of function application is computed as follows:

$\|\pi \vdash \textit{App name acts} : \tau\textit{-exp}\|$ env st $=$ **f** st **vals.**

Since st $\in$ Store$_{\pi,env}$ and (*name*, (env$_1$, **f**)) $\in$ env, by definition of consistency st $\in$ Store$_{\pi_1,env_1}$ also. The result follows from the type of function **f**.

1.2.4. Commands

We have to prove that for all well formed syntax phrases C $\in$ Command, env $\in$ Env$_\pi$ and st $\in$ Store $_{\pi,\ env}$

$$\|\pi \vdash \text{C}:\textit{comm}\| \text{ env st} \in \left(\text{Store}_{\pi,\ env}\right)_\perp.$$

Skip In the semantics $\|\pi \vdash \textit{Skip} : \textit{comm}\|$ env st = st. It is given that st $\in \left(\text{Store}_{\pi,\ env}\right)_\perp$.

Assignment In the semantics module the meaning of the phrase *Assign loc exp* is computed as follows:

$\|\pi \vdash \textit{Assign loc exp} : \textit{comm}\|$ env st $=$

updateStore ($\|\pi \vdash \textit{loc} : \tau\textit{-loc-exp}\|$ env st) ($\|\pi \vdash \textit{exp} : \tau - \textit{exp}\|$ env st) st.

By the inductive hypothesis:

$\|\pi \vdash \textit{loc} : \tau\textit{-loc-exp}\|$ env st $\in$ Location$_\perp$,

$\|\pi \vdash \textit{exp} : \tau - \textit{exp}\|$ env st $\in \|\tau\|_\perp$,

and the type of *updateStore* is
$Location_{\perp} \rightarrow StorableValue_{\perp} \rightarrow Store_{\perp} \rightarrow Store_{\perp}$.

Because of the typing requirement that τ is the same for both *location* and *expression* (e.g., loc:*int-loc* and exp:*int-exp*) this update of the store preserves the consistency between the environment, type-assignment π and the store. That is
$\|\pi \vdash \mathit{Assign\ loc\ exp} : \mathit{comm}\|$ env st $\in (\mathrm{Store}_{\pi,\, env})_{\perp}$.

Conditional evaluation In the semantics module the conditional statement phrase *If exp c1 c2* is evaluated as follows:
$\|\pi \vdash \mathit{If\ exp\ c1\ c2} : \mathit{comm}\|$ env st $=$ *if* ($\|\pi \vdash \mathit{exp} : \mathit{bool\text{-}exp}\|$ env st)
then ($\|\pi \vdash \mathit{c1} : \mathit{comm}\|$ env st) *else* ($\|\pi \vdash \mathit{c2} : \mathit{comm}\|$ env st) .

Let $\|\pi \vdash \mathit{exp} : \mathit{bool\text{-}exp}\|$ env st $= \mathbf{b} \in \mathrm{Boolean}_{\perp}$ by the inductive hypothesis.

Case $\mathbf{b} = $ True
$\|\pi \vdash \mathit{If\ exp\ c1\ c2} : \mathit{comm}\|$ env st $= \|\pi \vdash \mathit{c1} : \mathit{comm}\|$ env st $\in (\mathrm{Store}_{\pi,\, env})_{\perp}$ by the inductive hypothesis.

Case $\mathbf{b} = $ False
$\|\pi \vdash \mathit{If\ exp\ c1\ c2} : \mathit{comm}\|$ env st $= \|\pi \vdash \mathit{c2} : \mathit{comm}\|$ env st $\in (\mathrm{Store}_{\pi,\, env})_{\perp}$ by the inductive hypothesis.

In case of $\mathbf{b} = \perp$ $\|\pi \vdash \mathit{If\ exp\ c1\ c2} : \mathit{comm}\|$ env st $= \perp$

Thus in all cases $\|\pi \vdash \mathit{If\ exp\ c1\ c2} : \mathit{comm}\|$ env st $\in (\mathrm{Store}_{\pi,\, env})_{\perp}$.

Loop The meaning of the phrase *While exp c* relies on the meaning of the subphrases which by the inductive hypothesis are:
$\|\pi \vdash \mathit{exp} : \mathit{bool\text{-}exp}\|$ env st $\in \mathrm{Boolean}_{\perp}$,
$\|\pi \vdash c : \mathit{comm}\|$ env st $\in (\mathrm{Store}_{\pi,\, env})_{\perp}$.

Now $\|\pi \vdash \mathit{While\ exp\ c} : \mathit{comm}\|$ env $= \mathbf{w}$, where
$\mathbf{w}$(store) $=$ *if* ($\|\pi \vdash \mathit{exp} : \mathit{bool\text{-}exp}\|$ env store) *then* $\mathbf{w}$($\|\pi \vdash c : \mathit{comm}\|$ env store) *else* store. That is if we take w_k as the k-th approximation to this fixpoint, $\mathbf{w}(\mathrm{st}) = \mathrm{st}'$ if there exists $\mathrm{k} \geq 0$ such that $\mathbf{w}_{\mathrm{k}}(\mathrm{st}) = \mathrm{st}'$. For all $\mathrm{k} \geq 0$, $\mathbf{w}_{\mathrm{k}} \in (\mathrm{Store}_{\pi,\, env})_{\perp} \rightarrow (\mathrm{Store}_{\pi,\, env})_{\perp} = \|\mathit{comm}\|$,hence $\mathrm{w} \in \|\mathit{comm}\|$ and the resulting store is consistent with π and env.

Sequence of the commands The semantics of the phrase *c1 Semi c2* is as follows:
$\|\pi \vdash \mathit{c1\ Semi\ c2} : \mathit{comm}\|$ env st $=$
$\|\pi \vdash \mathit{c2} : \mathit{comm}\|$ env ($\|\pi \vdash \mathit{c1} : \mathit{comm}\|$ env st).

By the inductive hypothesis

$\|\pi \vdash c1 : comm\|$ env st $= \mathbf{st'} \in (\text{Store}_{\pi,\, env})_\perp$.

Since the store $\mathbf{st'}$ is consistent with the environment env and the type-assignment π, by the inductive hypothesis

$\|\pi \vdash c2 : comm\|$ env $\mathbf{st'} \in (\text{Store}_{\pi,\, env})_\perp$.

Local declarations We need to show that $\|\pi \vdash decls\ In\ c : comm\|$ env st $\in (\text{Store}_{\pi,\, env})_\perp$. In semantics module the meaning of *decls In c* is computed as follows:

$\|\pi \vdash decls\ In\ c : comm\|$ env st $= \|\pi \bar{\cup} \pi_1 \vdash c : comm\|$ (env $\bar{\cup}\mathbf{env}_1$) $\mathbf{st}_1$, where $(\mathbf{env}_1,\ \mathbf{st}_1) = \|\pi \vdash decls : \pi_1\text{-}dec\|$ env st.

By the inductive hypothesis

$(\mathbf{env}_1,\ \mathbf{st}_1) \in \left(\mathbf{env}_1\text{:}\ \|\pi_1\| \times \left(\text{Store}_{\pi,\, env} \cap \text{Store}_{\pi_1,\, env_1}\right)\right)_\perp$. This means that $\mathbf{env}_1 \in \text{Env}_{\pi_1}$, and $\mathbf{st}_1 \in \text{Store}_{\pi,\, env}$ and $\mathbf{st}_1 \in \text{Store}_{\pi_1,\, env_1}$. By the lemmas 1.1 and 1.2 it follows that (env $\bar{\cup}\mathbf{env}_1) \in \text{Env}_{\pi\bar{\cup}\pi_1}$ and $\mathbf{st}_1 \in \text{Store}_{\pi\, \bar{\cup}\, \pi_1,\, env\, \bar{\cup}\, env_1}$.

Let $\mathbf{st}_2 = \|\pi \bar{\cup} \pi_1 \vdash c : comm\|$ (env $\bar{\cup}\mathbf{env}_1$) $\mathbf{st}_1$.

By the inductive hypothesis $\mathbf{st}_2 \in \text{Store}_{\pi\, \bar{\cup}\, \pi_1,\, env\, \bar{\cup}\, env_1}$.

Now we argue that $\mathbf{st}_2 \in (\text{Store}_{\pi,\, env})_\perp$.

Let us partition the type-assignment π and the environment env as follows: $\pi = \pi' \cup \pi''$ and env $=$ env$'\cup$ env$''$, where

for all $(\text{I}{:}\theta_1) \in \pi'$ and $(\text{I} = v_1) \in$ env$'$ $(\text{I}{:}\theta_2) \notin \pi_1$ and $(\text{I} = v_2) \notin \text{env}_1$ and

for all $(\text{I}{:}\theta_1) \in \pi''$ and $(\text{I} = v_1) \in$ env$''$ $(\text{I}{:}\theta_2) \in \pi_1$ and $(\text{I} = v_2) \in \text{env}_1$.

In other words π' and env$'$ are the parts of π and env which are not redefined by the local declarations *decls,* thus $\pi' \subseteq \pi\bar{\cup}\pi_1$ and env$' \subseteq$ env $\bar{\cup}$ $\mathbf{env}_1$. π'' and env$''$ are redefined by the local declarations and $\pi'' \cap (\pi\bar{\cup}\pi_1) = \emptyset$, and env$'' \cap$ (env$\bar{\cup}\mathbf{env}_1) = \emptyset$.

By the lemma 1.3 $\mathbf{st}_2 \in \text{Store}_{\pi',\, env'}$.

Also by the lemma 1.3 the starting store st $\in \text{Store}_{\pi'',\, env''}$. (Note that the allocated storage is never deallocated). Since in declarations if storage needs to be allocated always the fresh cells are allocated, by the definition of store, environment, type-assignment consistency $\mathbf{st}_1 \in \text{Store}_{\pi'',\, env''}$. The command c is evaluated in the environment env $\bar{\cup}$ $\mathbf{env}_1$ where all the identifiers that are in env$''$ are bound to the new denotable values, and in case of locations those are guaranteed to be distinct from the denotable values the same identifiers have in env$''$. It follows that the command c cannot change the part of the store $\mathbf{st}_1$ which is bound to the identifiers in env$''$,except by calling a procedure in env$\bar{\cup}$env$_1$, but by consistency such procedure preserves the type consistency of the store. Thus the resulting store $\mathbf{st}_2 \in \mathrm{Store}_{\pi'',\, env''}$.

Given the results that $\mathbf{st}_2 \in \mathrm{Store}_{\pi',\, env'}$ and $\mathbf{st}_2 \in \mathrm{Store}_{\pi'',\, env''}$, and since all the identifiers in π' and env$'$ are distinct from the identifiers in π'' and in env$''$, we can conclude that

$\mathbf{st}_2 \in \mathrm{Store}_{\pi' \cup \pi'',\, env' \cup env''}$, that is $\mathbf{st}_2 \in \mathrm{Store}_{\pi,\, env}$.

Procedure call In the phrase *Call name acts, name* is the procedure name and *acts* is the list of actual parameters. We must show that

$\|\pi \vdash \mathit{Call\ name\ acts} : \mathit{comm}\|$ env st $\in \mathrm{Store}_{\pi,\, env}$.

In semantics we compute the meaning of the subphrases and we know their types by the inductive hypothesis. Let

$(\mathrm{env}_1, \mathbf{f}) = \|\pi \vdash \mathit{name} : \pi_1\text{-}\theta_1\text{-}\mathit{list\text{-}proc}\|$ env $\in \|\pi_1\text{-}\theta_1\text{-}\mathit{list\text{-}proc}\| = (\mathrm{env}_1 : \|\pi_1\| \times$

$\bigcup_{\pi \in TypeAssignment} \bigcup_{env \in Env_\pi} (\mathrm{Store}_{\pi,env} \cap \mathrm{Store}_{\pi_1,env_1}) \to$

$\bigcup_{\theta_1^c\text{-}list \text{ corresponds to } \theta_1\text{-}list} \|\theta_1^c\text{-list}\| \to$

$(\mathrm{Store}_{\pi,env} \cap \mathrm{Store}_{\pi_1,env_1})_\perp)$

$\mathbf{vals} = \|\pi \vdash \mathit{acts} : \theta_2\text{-}\mathit{list}\|$ env st $\in \|\theta_2\text{-}\mathit{list}\|$.

By the typing requirement we also know that θ_2-*list* corresponds to θ_1-*list*.

The meaning of the procedure call is computed as follows:

$\|\pi \vdash \mathit{Call\ name\ acts} : \mathit{comm}\|$ env st $= \mathbf{f}$ st **vals.**

Since st $\in \mathrm{Store}_{\pi,env}$ and (name, $(\mathrm{env}_1, \mathbf{f})) \in$ env, by definition of consistency, st $\in \mathrm{Store}_{\pi_1,env_1}$ also. Since $\mathbf{f}$ preserves both $\mathrm{Store}_{\pi_1,env_1}$ and $\mathrm{Store}_{\pi,env}$ by its type, it follows that the result is in $\mathrm{Store}_{\pi,env\perp}$.

1.2.5. Declarations

We need to prove that for all well formed syntax phrases D $\in$ Declaration, env $\in \mathrm{Env}_\pi$ and st $\in \mathrm{Store}_{\pi,\, env}$:

$$\|\pi \vdash \mathrm{D}:\pi_1\text{-}\mathit{dec}\| \text{ env st} \in \Big(\mathrm{env}_1\text{: } \|\pi_1\| \times \Big(\mathrm{Store}_{\pi,\, env} \cap \mathrm{Store}_{\pi_1,\, env_1}\Big)\Big)_\perp.$$

Variable declaration The meaning of the variable declaration is computed as follows:

$$\Bigg\|\pi \vdash \mathit{Var\ name\ attr\ exp} : \underbrace{\{(\mathit{name}, \tau\text{-}\mathit{loc})\}}_{\pi_1}\text{-}\mathit{dec}\Bigg\| \text{ env st} = (\mathbf{env}_1, \mathbf{st}_1) \text{ where}$$

$([\mathbf{loc}], \mathbf{st}_1) = allocateStore\ [\|\pi \vdash exp : \tau\text{-}exp\|\ \text{env st}]\ \text{st}$,
$\mathbf{env}_1 = [(name, \mathbf{loc})] \in \|\pi_1\|$,
The type of the function *allocateStore:*
$allocateStore :: [StorableValue]_\perp \to Store_\perp \to ([Location] \times Store)_\perp$.
The inductive hypothesis for the expression is as follows:
$\|\pi \vdash exp : \tau\text{-}exp\|\ \text{env st} \in \|\tau\|_\perp$.

The environment $\mathbf{env}_1$ is consistent with the type-assignment π_1 since the identifier *name* is bound the value $\text{v} = \mathbf{loc}$ in $\mathbf{env}_1$ and $\|\text{v}\| \in \|\tau\text{-}loc\|$.

The consistency of the store $\mathbf{st}_1$with the environment $\mathbf{env}_1$ and type-assignment π_1 relies on the correctness of the function *allocateStore* that takes the homogeneous list of storable values (Integers or Booleans) and the store, allocates the list of fresh cell and stores the given storable values in them sequentially. The list of locations where the corresponding storable values are stored and the changed store are returned by the function.

By the typing requirement $attr = \tau\text{-}loc$ and expression *exp* has type $\tau\text{-}exp$. In the type assignment π_1 the identifier *name* is bound to the type $\tau\text{-}loc$. The function *allocateStore* stores the meaning of the expression *exp* (which is in $\|\tau\|_\perp$) and returns the location **loc** where the value is stored and the changed store $\mathbf{st}_1$. The identifier *name* is bound to the location **loc** in a mini-environment $\mathbf{env}_1$. This way the identifier *name* in $\mathbf{env}_1$ denotes the location **loc** of type $\tau - loc$ (as the type-assignment π_1 says) in the store $\mathbf{st}_1$, and the value of type τ is stored at that location. This means that the store $\mathbf{st}_1$ is consistent with the type-assignment π_1 and the mini-environment $\mathbf{env}_1$.

Since the declaration of a variable allocates the fresh cell on the store and the already allocated storage cells are not changed, the resulting store is consistent with π and env.

Thus $\text{st}_1 \in \left(\text{Store}_{\pi,\, env} \cap \text{Store}_{\pi_1,\, env_1}\right)$.

Array declaration Syntax for the array declaration is *Arr name ss attr exp,* where *ss* is the list of dimension sizes and *exp* is initializing literal array expression. By the inductive hypothesis:

$\|\pi \vdash ss : int\text{-}exp\text{-}lsit\|\ \text{env} \in \text{Store} \to [\|int\|]$
$\|\pi \vdash exp : \text{n-}\tau\text{-arr-exp}\|\ \text{env} \in \text{Store} \to [\text{Integer}] \times [\|\tau\|]$.
By the typing requirement $attr = \tau\text{-}loc.$

In semantics module the meaning of the *Arr name ss attr exp* is computed as follows:

$$\left\|\pi \vdash \textit{Arr name ss attr exp} : \underbrace{\{(\textit{name}, \text{n-}\tau\text{-}\textit{loc}\text{-arr})\}}_{\pi_1} \textit{-dec}\right\| \text{env st} = (\mathbf{env}_1, \mathbf{st}_1)$$

where

$(\text{ds}, \mathbf{vals}) = \|\pi \vdash \ \textit{exp} : \textit{n-}\tau\textit{-arr-exp}\|$ env st $\in ([\text{Integer}] \times [\|\tau\|])_\perp$ by the inductive hypothesis,

$(\mathbf{locs}, \mathbf{st}_1) = $ *allocateStore* **vals** st,

$\mathbf{dims} = \|\pi \vdash \textit{ss} : \textit{int-exp-list}\|$ env st $\in ([\text{Integer}])_\perp$ by the inductive hypothesis,

$\mathbf{env}_1 = [(\textit{name}, \ (\mathbf{dims}, \mathbf{locs}))] \in \{(\text{Identifier}, \ \|\textit{n-}\tau\textit{-loc-arr}\|)\}$.

By the same arguments as for the variable declaration the environment $\mathbf{env}_1$ is consistent with type-assignment π_1 and the store $\mathbf{st}_1 \in \text{Store}_{\pi,env} \cap \text{Store}_{\pi_1,env_1}$.

Sequence of declarations The syntactic phrase is *d1 DSemi d2.*

In semantics: $\|\pi_0 \vdash \textit{d1 DSemi d2}: (\pi_1 \dot{\cup} \pi_2)\textit{-dec}\ \|$ env st $= (\mathbf{env}_1 \cup \mathbf{env}_2, \mathbf{st}_2)$, where

$(\mathbf{env}_1, \mathbf{st}_1) = \|\pi_0 \vdash \textit{d1}{:}\pi_1\textit{-dec}\ \|$ env st $\in$

$(\mathbf{env}_1\text{:} \ \|\pi_1\| \times (\text{Store}_{\pi_0,env} \cap \text{Store}_{\pi_1,env_1}))_\perp$ by the inductive hypothesis,

$(\mathbf{env}_2, \mathbf{st}_2) = \|\pi_0 \dot{\cup} \pi_1 \vdash \textit{d2} : \pi_2\textit{-dec}\ \|$ $(\text{env} \cup \mathbf{env}_1)$ $\mathbf{st}_1 \in$

$\left(\mathbf{env}_2\text{:} \ \|\pi_2\| \times \left(\text{Store}_{\pi_0 \dot{\cup} \pi_1, env \cup env_1} \cap \text{Store}_{\pi_2,env_2}\right)\right)$ by the inductive hypothesis.

Since $\mathbf{env}_1 \in \text{Env}_{\pi_1}$, $\mathbf{env}_2 \in \text{Env}_{\pi_2}$, and since the identifiers in π_1 and $\mathbf{env}_1$ are distinct from those in π_2 and $\mathbf{env}_2$ (which is guaranteed by the $\dot{\cup}$ operation) it follows (by the definition of the environment - type-assignment consistency) that $\mathbf{env}_1 \cup \mathbf{env}_2 \in \text{Env}_{\pi_1 \dot{\cup} \pi_2}$.

Since $\mathbf{st}_2 \in \text{Store}_{\pi_0 \dot{\cup} \pi_1, env \cup env_1} \cap \text{Store}_{\pi_2,env_2}$ by definition $\mathbf{st}_2 \in \text{Store}_{\pi_0 \dot{\cup} \pi_1, env \cup env_1}$ and $\mathbf{st}_2 \in \text{Store}_{\pi_2,env_2}$.

By the lemma 1.3 from the facts that $\mathbf{st}_2 \in \text{Store}_{\pi_0 \dot{\cup} \pi_1, env \cup env_1}$, env $\in \text{Env}_\pi$ and $\mathbf{env}_1 \in \text{Env}_{\pi_1}$ it follows that $\mathbf{st}_2 \in \text{Store}_{\pi_0,env}$ and $\mathbf{st}_2 \in \text{Store}_{\pi_1,env_1}$.

Since the identifiers in π_1 and $\mathbf{env}_1$ are distinct from the identifiers in π_2 and $\mathbf{env}_2$

$\mathbf{st}_2 \in \text{Store}_{\pi_1 \dot{\cup} \pi_2, env_1 \cup env_2.}$ So we can conclude that

$\|\pi_0 \vdash \textit{d1 DSemi d2} : \ (\pi_1 \dot{\cup} \pi_2)\textit{-dec}\ \|$ env st $\in$

$\left(\mathbf{env}_1 \cup \mathbf{env}_2\text{:} \ \|(\pi_1 \dot{\cup} \pi_2)\| \times \left(\text{Store}_{\pi_0,env} \cap \text{Store}_{\pi_1 \dot{\cup} \pi_2, env_1 \cup env_2.}\right)\right)_\perp$.

Function declaration The syntactic phrase is *Fun name fs attr exp,* where *fs* is formal parameters list (which is of a type type-assignment), *exp* is the function body, and *attr* is the type of *exp* by the typing requirement. We should show that:

$$\left\| \pi \vdash \textit{Fun name fs attr exp}\colon \underbrace{\{(\textit{name},\pi\text{-}\theta\text{-}\textit{list}\text{-}\tau\text{-}\textit{fun})\}}_{\pi_1}\textit{-dec} \right\| \text{env st} \in \|\pi_1\textit{-dec}\|$$

$=$

$(\{\{\textit{name}\} \times \|\pi\text{-}\theta\text{-}\textit{list}\text{-}\tau\text{-}\textit{fun}\|\} \times (\text{Store}_{\pi,env} \cap \text{Store}_{\pi_1,env_1}))_\perp =$

$(\{\{\textit{name}\} \times (\text{env: } \|\pi\| \times$

$(\bigcup_{\pi_2 \in TypeAssignment} \bigcup_{env_2 \in Env_{\pi_2}} (\text{Store}_{\pi_2,env_2} \cap \text{Store}_{\pi,env}) \to$

$\bigcup_{\theta^c\textit{-list} \text{ corresponds to } \theta\textit{-list}} \|\theta^c\text{-list}\| \to \|\tau\|_\perp))\} \times$

$(Store_{\pi,env} \cap Store_{\pi_1,env_1}))_\perp$

In semantics

$\|\pi \vdash$ *Fun name fs attr exp*: $\{(\textit{name}, \theta\text{-}\textit{list}\text{-}\tau\text{-}\textit{fun})\}$-dec $\|$ env st $= (\mathbf{env}_1, \text{st})$,

where $\mathbf{env}_1 = \{(\text{name}, (\text{env}, \mathbf{fun}))\}$,

$\mathbf{fun} = (\backslash\text{st}' \to (\backslash\text{acts} \to \|\pi \bar{\cup} \textit{fs} \bar{\cup} \{(\textit{name}, \theta\text{-}\textit{list}\text{-}\tau\text{-}\textit{fun})\} \vdash \textit{exp}{:}\tau\text{-}\textit{exp}\|$

$(\mathbf{env}'' \bar{\cup} \{(\textit{name}, (\text{env}, \mathbf{fun}))\})\ \mathbf{st}''))$, where

$(\mathbf{env}'', \mathbf{st}'') = \textit{mapActuals fs}$ acts env st$'$.

Let $\pi_2 \in$ TypeAssignment and $\text{env}_2 \in \text{Env}_{\pi_2}$ be given (consider them to be function application time type-assignment and environment).

We need to show that if $\text{st}' \in (\text{Store}_{\pi_2,env_2} \cap \text{Store}_{\pi,env})$, then:

$\mathbf{env}' = \mathbf{env}'' \bar{\cup} \{(\textit{name}, (\text{env}, \mathbf{fun}))\} \in \text{Env}_{\pi\ \bar{\cup}\ fs\ \bar{\cup}\ \{(\text{name},\ \pi\text{-}\theta\text{-}list\text{-}\tau\text{-}fun)\}}$ and

$\mathbf{st}'' \in \text{Store}_{\pi \bar{\cup} fs\ \bar{\cup} \{(name,\ \pi\text{-}\theta\text{-}list\text{-}\tau\text{-}fun)\},\ \mathbf{env}''\ \bar{\cup}\ \{(\text{name},\ (\text{env},\ \mathbf{fun}))\}}$.

Than by the inductive hypothesis for *exp*

$\|\pi \bar{\cup} \textit{fs} \bar{\cup} \{(\textit{name}, \pi\text{-}\theta\text{-}\textit{list}\text{-}\tau\text{-}\textit{fun})\} \vdash \textit{exp} : \tau\text{-}\textit{exp}\|\ \mathbf{env}'\ \mathbf{st}'' \in \|\tau\|_\perp$.

It thus remains to show the following two lemmas.

Lemma 1.2.5.1. (env, **fun**) $\in \|\pi\text{-}\theta\text{-}\textit{list}\text{-}\tau\text{-}\textit{fun}\|$.

Proof: Let $\pi_2 \in$ TypeAssignment and $\text{env}_2 \in \text{Env}_{\pi_2}$ be given. We assume that the store $\text{st}' \in (\text{Store}_{\pi_2,\ env_2} \cap \text{Store}_{\pi,\ env})$, and that the list of actual parameters $\textit{acts} \in \bigcup_{\theta^c\textit{-list} \text{ corresponds to } \theta\textit{-list}} \|\theta^c\textit{-list}\|$.

Since the function that we are constructing is recursive the proof is by fix-point induction.

Let function generator for **fun**, G be

$\text{G} = (\backslash\text{F} \to (\backslash\text{st}' \to (\backslash\text{acts} \to \|\pi \bar{\cup} \textit{fs} \bar{\cup} \{(\textit{name}, \pi\text{-}\theta\text{-}\textit{list}\text{-}\tau\text{-}\textit{fun})\} \vdash \textit{exp}{:}\tau\text{-}\textit{exp}\|$

$(\mathbf{env}'' \bar{\cup} \{(\text{name}, (\text{env}, \text{F}))\})\ \mathbf{st}'')))$, where

$(\mathbf{env}'', \mathbf{st}'') = \textit{mapActuals fs acts}$ env st$'$.

We will show (by induction) that for all functions F_i, that are approximations of the function **fun** with the recursive bindings unfolded i times ($i \geq 0$)

(env, G $(\text{F}_i)) \in \|\pi\text{-}\theta\text{-}\textit{list}\text{-}\tau\text{-}\textit{fun}\|$.

Base case: Let $\text{F}_0 = (\backslash\text{st}' \to \backslash\text{acts} \to \perp)$. By the types of st$'$, acts and F_0

$\{(\text{name}, (\text{env}, \text{F}_0))\} \in \|(\textit{name}, \pi\text{-}\theta\text{-}\textit{list}\text{-}\tau\text{-}\textit{fun})\|$

By the lemma 1.2.5.2 $\mathbf{env}'' \in \text{Env}_{\pi\ \cup\ fs}$ and $\mathbf{st}'' \in \text{Store}_{\ \pi\cup\ fs,\ \mathbf{env}''}$.

By lemma 1.1 $\mathbf{env}''\cup\{(\text{name}, (\text{env}, \text{F}_0))\} \in \text{Env}_{\pi\ \cup\ fs\ \cup\ \{(\textit{name},\ \pi\text{-}\theta\text{-}\textit{list}\text{-}\tau\text{-}\textit{fun})\}}$.

Since $\mathbf{st}'' \in \text{Store}_{\ \pi\ \cup\ fs,\ \mathbf{env}''}$, and $\mathbf{st}'' \in \text{Store}_{\{(\textit{name},\ \theta\text{-}\textit{list}\text{-}\tau\text{-}\textit{fun})\},\ \{(\text{ name, (env, F}_0))\}}$ trivially by the definition of store, environment and type-assignment consistency, then by lemma 1.2

$\mathbf{st}'' \in \text{Store}_{\pi\cup fs\ \cup\{(\textit{name},\ \pi\text{-}\theta\text{-}\textit{list}\text{-}\tau\text{-}\textit{fun})\},\ \mathbf{env}''\ \cup\ \{(\text{name, (env, F}_0))\}}$.

It follows by the inductive hypothesis by expressions that

$\{(\text{name}, (\text{env}, \text{F}_0))\} \in \|(\textit{name}, \pi\text{-}\theta\text{-}\textit{list}\text{-}\tau\text{-}\textit{fun})\|$.

Inductive hypothesis: Let $\text{F}_{i+1} = \text{G}(\text{F}_i)$. We assume that

$(\text{env}, \text{G}(\text{F}_i)) \in \|(\textit{name}, \pi\text{-}\theta\text{-}\textit{list}\text{-}\tau\text{-}\textit{fun})\|$.

Inductive case: Show that

$(\text{env}, \text{G}(\text{F}_{i+1})) \in \|(\textit{name}, \pi\text{-}\theta\text{-}\textit{list}\text{-}\tau\text{-}\textit{fun})\|$.

By definition:

$\text{G}(\text{F}_{i+1}) = (\backslash\text{st}' \to (\backslash\text{acts} \to \|\pi\cup fs\cup\{(\textit{name}, \pi\text{-}\theta\text{-}\textit{list}\text{-}\tau\text{-}\textit{fun})\} \vdash \textit{exp}{:}\tau\text{-}\textit{exp}\|$
$(\mathbf{env}''\cup\{(\text{name}, \text{G}(\text{F}_i))\})\ \mathbf{st}''))),$ where
$(\mathbf{env}'', \mathbf{st}'') = \textit{mapActuals fs acts}\ \text{env st}'$.

By the inductive hypothesis

$\{(\text{name}, (\text{env}, \text{G}(\text{F}_i)))\} \in \|\{(\textit{name}, \theta\text{-}\textit{list}\text{-}\tau\text{-}\textit{fun})\}\|$.

Then the result follows by the same argument as for the base case $\square$.

Lemma 1.2.5.2. For an environment $\textit{env} \in \text{Env}_\pi$, store $\textit{st} \in \text{Store}_{\textit{env},\ \pi}$ and lists of formal parameters *fs* and the actual parameters *acts* that correspond let

$(\mathbf{env}'', \mathbf{st}'') = \textit{mapActuals fs acts env st},$ then

$$\begin{aligned} \mathbf{env}'' &\in \text{Env}_{\pi\ \cup fs}, \text{ and} \\ \mathbf{st}'' &\in \text{Store}_{\pi\ \cup\ fs,\ \mathbf{env}''}. \end{aligned}$$

Proof: The proof is by induction on the formal *fs* and actual *acts* parameter lists. (Note that in the implementation of typing and semantics we are using the list for the type-assignment and the run-time environment to model the sets. So we are going to use the notation [] instead of more common {} for a set. The order of elements in formal parameters list and actual parameters list matters only in the sense that i-th element of formals correspond to i-th element of actuals.).

Base case: Both *fs* and *acts* are empty lists:

$$\textit{mapActuals [] [] env st} = (\textit{env, st}).$$

Since *env* and *st* are consistent with each other and π, it follows that they are consistent with $\pi \bar{\cup}$ [] .

Inductive hypothesis: We assume that for all *env* and *st* consistent with each other and with some type environment π :

$$mapActuals\ [\ f_1, ..., f_{n-1}]\ \ [a_1,\ ...,\ a_{n-1}]\ env\ st\ =\ (env_1,\ store_1),$$

such that env_1 and $store_1$ are consistent with each other and a type assignment $\pi \bar{\cup}\ [f_1, ..., f_{n-1}]$.

Inductive step: We need to prove that *mapActuals* $[f_1, ..., f_n]\ [a_1,...,a_n]$ *env st* $= (env_3,\ store_3)$, where $env_3 \in \mathrm{Env}_{\pi\ \bar{\cup}\ [f_1,...,f_n]}$ and $store_3 \in \mathrm{Store}_{\ \pi\ \bar{\cup}\ [f_1,...,f_n], env_3}$.

In this case:

$$\begin{aligned} & mapActuals\ \ [f_1, ..., f_n]\ \ [a_1,...,a_n]\ \ env\ st \\ = \ \ & mapActuals\ \ [f_1, ..., f_{n-1}]\ \ [a_1,\ ...,\ a_{n-1}]\ env_2\ store_2, \end{aligned}$$

where $env_2 \in \mathrm{Env}_{\pi \bar{\cup}\{f_n\}}$ and $store_2 \in \mathrm{Store}_{\ \pi\ \bar{\cup}\ \{f_n\},\ env_2}$ are computed by the function *mapActuals* as follows:

Case1: If f_n is a reference parameter: $f_n = (name,\ \tau\text{-}loc\text{-}ref)$ and the actual parameter $a_n = \mathrm{v}$, then the environment env_2 is computed as

$$env_2\ =\ env\bar{\cup}(name,\ \mathrm{v}).$$

Since by the definition of the parameter lists correspondence $\mathrm{v} \in \|\tau\text{-}loc\|$ and since $\|\tau\text{-}loc\text{-}ref\| = \|\tau\text{-}loc\|$, it follows that $env_2 \in \mathrm{Env}_{\pi \bar{\cup} f_n}$ by the definition of the environment - type-assignment consistency. The store is not changed

$$store_2\ =\ store.$$

$store_2 \in \mathrm{Store}_{\pi\ \bar{\cup}\ \{f_n\}, env_2}$ since name in env_2 is bound to already existing location in the store and since (by formal-to-actual correspondence) type of this location is $\tau\text{-}loc$.

Case 2: If f_n is an array-reference parameter: $f_n = (name,\ \text{n-}\tau\text{-}loc\text{-}arr\text{-}ref)$ and the actual $a_n = \mathrm{v}$. Then

$$env_2\ =\ env\bar{\cup}(name,\ \mathrm{v}),\ \text{ and}$$

$$store_2\ =\ store.$$

The proof of consistency is analogous to one for location by reference parameter.

Case 3: If f_n = (*name*, τ-*loc*) then by the parameter lists correspondence the actual a_n = v , where v $\in \|\tau\|$ or a_n = l, where l $\in$ Location and *lookupStore*(l, *store*) $\in \|\tau\|$.

In case a_n = v let (l$'$, store$'$) = *allocateStore*([v] , *store*). Then

$$env_2 = \ env \bar{\cup} (name,\ \mathrm{l}'), \text{ and}$$

$$store_2 = \mathrm{store}'.$$

In case a_n = l, let (l$'$, store$'$) = *allocateStore* ([*lookupStore*(l, *store*)] ,*store*). Then

$$env_2 = env \bar{\cup} (name,\ \mathrm{l}'), \text{ and}$$

$$store_2 = \mathrm{store}'.$$

$env_2 \in \mathrm{Env}_{\pi\ \bar{\cup}\ \{f_n\}}$ since $env \in \mathrm{Env}_\pi$ and since value bound to *name* in env_2 is in the meaning of type which *name* has in $\pi\bar{\cup}\ \{f_n\}$. In case f_n = (*name*, τ-*loc*) the fresh location is allocated on store. By the formal-to-actual correspondence the value which is stored there is in $\|\tau\|$,thus $\mathrm{store}_2 \in$ Store$_{\ \pi\ \bar{\cup}\ \{f_n\}, env_2}$.

Case 4: If f_n = (*name*, n-τ-*loc-arr*) then by the formal-to-actual correspondence the actual a_n = (dims, locs) $\in$ [Integer] $\times$ [Location] and for all l $\in$ locs *lookupStore*(l, *store*) $\in \|\tau\|$. Let vals $\in [\|\tau\|]$ be such that vals_i = *lookupStore*(locs_i, *store*), where 1 $\leq$ i $\leq$ *length*(locs) and (locs$'$, store$'$) = *allocateStore* (vals, *store*), then

$$env_2 = env\ \bar{\cup}\ (name,\ (\mathrm{dims},\ \mathrm{locs}')), \text{ and}$$

$$store_2 = \mathrm{store}'.$$

Now $env_2 \in \mathrm{Env}_{\pi\bar{\cup}f_n}$ since $env \in \mathrm{Env}_\pi$ and since value bound to *name* in env_2 is in the meaning of type which *name* has in $\pi\bar{\cup}\ \{f_n\}$. In case of f_n = (*name*, n-τ-*loc-arr*) the fresh locations are allocated on store. By the formal-to-actual correspondence the values which are stored in those are all in $\|\tau\|$,thus $store_2 \in$ Store$_{\pi\ \bar{\cup}\ \{f_n\}, env_2}$.

So, in all four cases of formal f_n and actual a_n (which are all possible cases when the lists of formal and actual parameters correspond), the environment $env_2 \in \mathrm{Env}_{\pi \bar{\cup} \{f_n\}}$, and $store_2 \in \mathrm{Store}_{\pi \bar{\cup} \{f_n\},\, env_2}$.

By the inductive hypothesis:

$$mapActuals\ [f_1, ..., f_{n-1}]\ \ [a_1, ..., a_{n-1}]\ env_2\ store_2 = (env_3,\ store_3), \text{ where}$$

$\mathrm{env}_3 \in \mathrm{Env}_{\pi \bar{\cup} \{f_n\} \bar{\cup} [f_1,...,f_{n-1}]}$ and $\mathrm{store}_3 \in \mathrm{Store}_{\pi \bar{\cup} \{f_n\} \bar{\cup} [f_1,...,f_{n-1}],\, env_3}$. Since all the identifiers in the list of the formal parameters are distinct by the typing requirement it follows that

$env_3 \in \mathrm{Env}_{\pi\ \bar{\cup}\ [f_1,...,f_n]}$ and $store_3 \in \mathrm{Store}_{\ \pi\ \bar{\cup}\ [f_1,...,f_n], env_3}$ □. (Lemma 1.2.5.2)

Procedure declaration The syntax for the procedure declaration is: *Proc name fs il pb pbs,* where *fs* is the list of formal parameters, *il* is the list of global variables that can be used by the procedure, *pb* is the procedure body (which is a command), *pbs* is the list of alternative procedure bodies for all possible alias combinations among the actual parameters and imported global variables.

Analogous to the function declaration:

$\left\| \pi \vdash Proc\ name\ fs\ il\ pb\ alts\text{:}\ \underbrace{\{(\mathrm{name}, \pi\text{-}\theta\text{-}list\text{-}proc)\}}_{\pi_1}\text{-dec} \right\|$ env st $=$ ($\mathbf{env}_1$, st), where

$\mathbf{env}_1 = [name,\ (\mathrm{env},\ \mathbf{proc}))]$, and **proc** is a function:

$\mathbf{proc} = (\backslash \mathrm{st}' \rightarrow (\backslash \mathrm{acts} \rightarrow \|\pi_{il} \bar{\cup} fs \bar{\cup} \{(name,\ \pi\text{-}\theta\text{-}list\text{-}proc)\} \vdash \mathbf{body}\text{:}comm\|$ $\mathbf{env}'\ \mathbf{st}\ '')),$ where

$(\mathbf{env}'', \mathbf{st}'') = mapActuals\ fs\ acts\ \mathrm{env}\ \mathrm{st}',\ \mathbf{env}' = \mathbf{env}'' \bar{\cup} \{(name,\ (\mathrm{env},\ \mathbf{proc}))\}$ and

$\mathbf{body} = findAlternative\ fs\ il\ pb\ pbs\ \mathbf{env}''$.

findAlternative is the function which finds the procedure body that needs to be executed depending on the aliases among the formals and the import list variables.

In the above formulas, π_{il} is the subset of the type-assignment π which includes only the identifiers from the import list *il.* Note that for $env \in \mathrm{Env}_\pi$ if $\pi_{il} \subseteq \pi \Rightarrow env \in \mathrm{Env}_{\pi_{il}}$ (i.e. env is consistent with π_{il}) by the definition of consistency. Analogously if store *st* is consistent with the environment *env* and the type-assignment $\pi \Rightarrow$ for $\pi_{il} \in \pi$ *st* is consistent with *env* and π_{il} by the definition of the store, environment and type-assignment consistency.

The rest of the argument is analogous to the argument as for the function declarations □.