Alias-free parameters in C using multibodies

Medhat G. Assaad

TR #01-05

July 2001

Keywords: pointer parameter aliasing, global variable aliasing, multi-body procedures, dynamic dispatch, static dispatch, ACL language, C/ACL, alias-free programs, compiler optimizations, GCC.

2000 CR Categories: D.3.1 [*Programming Languages*] Formal Definitions and Theory — sematics; D.3.3 [*Programming Languages*] Language Constructs and Features — control structures, procedures, functions, and subroutines; D.3.4 [*Programming Languages*] Processors — compilers, optimization; D.3.m [*Programming Languages*] Miscellaneous — dynamic dispatch, multiple dispatch; F.3.1 [*Logics and Meanings of Programs*] Specifying and Verifying and Reasoning about Programs — logics of programs.

Department of Computer Science

226 Atanasoff Hall

Iowa State University

Ames, Iowa 50011-1040, USA

TABLE OF CONTENTS

CHAPTER 1 INTRODUCTION

Two names are *aliases* if they point to the same storage cell in memory, which means that this storage cell can be modified through more than one path; so if a variable x is not modified in a code segment, still, the memory location associated with x can be modified if another variable y that is an alias to x is modified in the same code segment. The presence of aliases adds to the complexity of a program, making it more difficult to reason about and verify. Moreover, the presence of aliases limits the amount of optimization that can be done by a compiler, unless it does high-quality alias analysis [11, p. 293], which is usually very costly.

This paper discusses some of the different approaches that have been used to deal with aliasing problems, and it studies the effectiveness of the ACL approach [10]. The ACL approach limits aliasing based on dispatch to different bodies of the same function according to the aliasing pattern at the call site. In this paper we use the same approach and apply it to the C programming language, and then measure the performance of the resulting code.

The rest of this introduction is organized as follows. Section 1.1 discusses problems caused from aliasing. Section 1.2 gives an overview of some of the approaches that are used to deal with aliasing problems, and shows the reasons why we found that ACL has something new to offer. Section 1.3 shows the ACL approach. Finally we give an outline of the rest of the paper.

1.1 Problems caused from aliasing

Aliasing can occur for many reasons, one of which is passing pointers. Many approaches (which will be discussed later) have been suggested for dealing with aliasing. In this paper, we apply the ACL approach to the C programming language, to measure the performance that can be achieved by eliminating aliasing between procedure arguments, and between arguments

and global variables, which enables the compiler to do more aggressive optimization.

Aliasing from parameters can happen in two ways. First, the same pointer can be passed into two parameters, in this case, this same location will have two different names, aliases, inside the called procedure. The other way is when a pointer to a global is passed as one of the arguments. If this global is used inside the called procedure, then again, it will have two names inside that procedure. These pointers can result from the use of call-by-reference, indirect storage models, or, as in C, from explicit constructs in the language.

Aliases can cause, in general, two kinds of problems. The first problem is that in the presence of aliases, it is much harder to reason about the correctness of the code. For example, in Figure 1.1, the `get_sum()` procedure is supposed to calculate the sum of the elements of array `a`, and return this sum in the location pointed to by the parameter `sum`. While this code may look like it will always return the required result, in the presence of aliasing, this may not always be true. It may return an incorrect result if `*sum` is itself one of the locations in the array `a`; for example if invoked with `get_sum(b, 10, &b[3])`. In C, `*sum` refers to the memory location pointed to by `sum`, while `&x` returns the address of `x`.

```
void get_sum (int a[], int num, int *sum){
   int i;

   *sum=0;
   for (i=0; i < num; i++)
      *sum += a[i];
}
```

Figure 1.1 An example of correctness problems caused by aliases

The second kind of problem that may be caused by aliases may not affect the correctness of code but will limit the ability of the compiler to achieve good optimization. In the code given in Figure 1.2, while the equality check between `c` and `*a` may seem to be always true and redundant, it cannot be removed by an optimizing compiler, since `*a` and `g` may be aliases, and `maybe_modify_g()` may potentially alter the value of `g`. Thus the redundancy of the check is

not provable unless the compiler can figure out the aliasing relationship between the argument to `foo()` and all the globals that are used in `maybe_modify_g()`. With separate compilation, or if `foo()` is a function in a library, the compiler will have no way to analyze all the locations from which `foo()` is called. Moreover, if `foo()` is called, say, 1000 times, in only one of which `*a` and `g` are aliases, the compiler will not be able to eliminate this seemingly redundant check to insure the correctness of this one case, thus missing an opportunity to increase code efficiency.

```
int g = 5;

void foo (int *a){
   int c;

   c = *a;
   maybe_modify_g ();
   if (c == *a){
     /* do something here */
   }
}
```

Figure 1.2 An example of optimization problems caused by aliases

1.2 Literature review

Other research has tried different approaches to solving aliasing problems. These approaches usually follow one of two patterns, either restricting the aliasing possible, or giving some hint to the compiler about the aliasing that pattern that will occur at run time. Euclid [14] and RESOLVE [4, 12, 15, 6] restrict the possible aliasing patterns. Some other compilers [16, 7, 8] and the latest C standard [9] use hints to instruct the compiler about aliasing, so that this information can be used for optimization. We discuss these below.

The main goals behind the design of Euclid was to help in program verification. Thus they had to deal with problems that are caused from aliasing. Euclid has pointer variables, and passing by-reference, which both can cause aliasing between parameters. Pointers in Euclid are

indexes into collections of objects. Collections are explicitly declared variables that are similar the implicit arrays accessed by pointers in other languages, like C [14]. Thus eliminating aliasing in Euclid is all about eliminating aliasing between for arrays. But Euclid prohibits any aliasing between the actual parameters passed to a function by generating checks in the code that will signal run-time errors if any overlap is detected between the arguments.

In general, to ensure that there is no aliasing between the actual parameters, the developer will have to add alias analysis code at the call site. Figure 1.3 shows such code for a function, `p`, that takes three reference parameters, when called with `a[i]`, `a[j]`, and `a[k]` as the actual parameters. In that code `p_abc`, `p_ab`, `p_ac`, and `p_bc` are different versions of `p` that handle different aliasing patterns but achieve the same postconditions. The problem of this approach is that to ensure correctness, the programmer must add these checks at each call site. This is what ACL automates.

```
if ( i == j && j == k) {
  call p_abc(a[i])
} else if ( i == j ) {
  call p_ab(a[i], a[k])
} else if ( i == k ) {
  call p_ac(a[i], a[j])
} else if ( j == k ) {
  call p_bc(a[i], a[j])
} else {
  call p(a[i], a[j], a[k])
}
```

Figure 1.3 Hand-coded alias analysis at the call-site [10]

RESOLVE on the other hand is concerned with code reuse and modularity. It uses *swapping* [12, p. 26] for data movement. Swapping means that assignments in RESOLVE are bidirectional, i.e. when x is assigned to y the value of x as copied to y and the value of y is copied to x. This swapping is also used when passing arguments to functions. By using this swapping operation, it is guaranteed that there is only one copy of each pointer during execution, thus there will be no aliasing. But this also means that the parameters that can

be passed to a function are restricted, so for example, "... repeated arguments to calls, use of global variables as arguments to calls, aliased pointers and array references ... are outlawed or simply cannot arise" [3, p. 50]. Thus RESOLVE suffers from the same problems as Euclid.

Some compilers, like IBM's XL compiler [16, 8, 7], use special hints given by the developer to decide the aliasing pattern between parameters. The XL compiler for example, uses a pragma that indicates which parameters are guaranteed not to be aliases in a piece of code. This pragma can be used at the function definition site to instruct the compiler about the possible aliasing patterns at call sites. This pragma is used for optimization, but it does not pose any restrictions on the parameters used at call sites. But, even if, unlike Euclid, no run-time errors will be generated if these pragmas do not hold at each call site, the behavior of the code at these calls is undefined. Thus to ensure correctness the programmer will have to use code like that in Figure 1.3, and so encounters the same problems as the Euclid programmes.

In the C standard, a new type qualifier, '`restrict`' [9, Section 6.7.3.1], can be used with pointers to instruct the compiler about the aliasing patterns that that pointer can be involved in. Figure 1.4 shows an example using `restrict`. In that example the compiler will assume that at each invocation of `f`, `x` and `y` will not be aliases, and it may do optimization based on that assumption. But the developer must insure that none of the call sites breaks the assumed pattern, otherwise the behavior is undefined. So, as with IBM's XL compiler, the extra checks are needed at each call site.

In summary, all of these approaches, either explicitly or implicitly, limit the options avail-

```
void f(int * restrict x, int * restrict y, int size)
{
  int i;

  for (i=0; i<size; i++)
    x[i] = y[i];
}
```

Figure 1.4 The use of the `restrict` type qualifier in C

able for the developer at the call site. And, they may necessitate adding extra code at each call site to ensure correctness of the resulting code. Hence all could benefit from the ACL idea.

1.3 The ACL approach

Alias Controlling Language, or ACL for short, takes a language design approach to eliminating aliasing [10, p. 4]. In this approach, a separate body is provided for each aliasing pattern that can occur between the parameters and between the parameters and the globals, thus each procedure may have multiple bodies, each of which is guaranteed not to have any aliasing between the parameters. At the call site, a call will be dispatched (statically if possible, or otherwise dynamically) to the body that matches the aliasing pattern among the arguments, if that body does not exist an error will occur. Thus, ACL follows a Euclid-style solution, but it moves the burden of alias analysis from the call site to the function-definition site.

Figure 1.5 shows a procedure written in C/ACL, i.e. C with ACL extensions. In this code we notice two things that give ACL its distinct flavor. First, a procedure must specify explicitly all the globals it uses with an `imports()` clause. Then, the "`| alias (x, a)`" starts the second body for `swap_with_a()`, this body, which in this case does nothing, will be used when `swap_with_a()` is called with an alias to the global variable `a`. In C/ACL only the first variable in the alias list that can be used inside the corresponding body. So, in that example, only `x` can be used in the second body.

```
int a = 1;

void swap_with_a (int *x) imports (a){
   int tmp = *x;
   *x = a;
   a = tmp;
} | alias (x, a){
   /* do nothing */
}
```

Figure 1.5 C/ACL code with multiple bodies

1.3.1 The 'imports' clause

The imports clause is used to declare all the globals that are used in the function, which can be used to check that all the possible bodies are implemented, or give warnings for unimplemented ones. To make the transition to ACL easier and to maintain backwards compatibility with C, the imports clause acts like a flag for ACL functions. So, even if a function does not use any globals, it must have an imports clause – an empty one in this case – to be treated as an ACL-function. A function that does not have an imports clause is treated as a normal C function, and is not allowed to have multiple bodies.

Globals listed in the imports clause can be listed as identifiers, in which case they will have the same type inside the function as outside, or, they can be given a more restrictive type – in terms of `const`ness – which can be used by the compiler when optimizing that body. Figure 1.6 shows an example in which `size` is a global integer, inside the function `array_sum` it will be treated as a constant.

1.3.2 Multi-bodies

Each ACL function can have more than one body implementing it. Inside the first body all the parameters and imported globals can be used. Subsequent bodies are started with an alias list which has the following syntax: '`| alias (` *alias_list* `;` *alias_list* `; ...)`' where each *alias_list* is a comma-separated list of identifiers. An *alias_list* can include any parameter that has a pointer type or any of the imported globals. The semantics is that inside that body, all the stated identifiers in each *alias_list* will be aliases to each other. Inside that body, only identifiers from the parameters and imported globals that are not included in alias lists and the first identifier in each *alias_list* can be used. For example, Figure 1.5 shows a function with two bodies.

1.4 Hypothesis

In order to measure the merits of ACL for optimization we set up an experiment in which we can measure the speedup that can be gained from using it in a piece of code. This practical

```
int size = 5;

int array_sum (int a[]) imports (const int size){
   int sum=0;
   int i;

   for (i=0; i<size; i++){
     sum += a[i];
   }

   return sum;
}
```

Figure 1.6 Globals with restricted types in the `imports` clause

experiment should also be helpful in uncovering the issues that may hamper the adoption of ACL by developers.

Our main hypothesis is that by applying ACL to a program we can get n enhancement in the run-time execution speed of the program. We hypothesize that this will occur for the following reasons:

1.4.1 Specially written bodies

Since the developer can supply a separate body for each aliasing pattern, and since the developer knows the semantics of this function for each aliasing pattern, the developer can write each body in an efficient way to make use of that pattern. An example is the second body in Fig. 1.5 in which, knowing that `x` and `a` are aliases, we had only to give an empty body. Our hypothesis is that each body is no less efficient than the original function.

1.4.2 Alias-free arguments

In a function that uses ACL features, the arguments to each body will be guaranteed not to be aliased with other arguments or globals. Our hypothesis is that this extra piece of information enables a compiler to do a better job when optimizing these bodies, thus generating more efficient code for each of them, compared to a standard compiler.

1.4.3 Dynamic dispatch

We hypothesize that the dynamic dispatch code that matches an aliasing pattern to the corresponding body will be the only overhead from using ACL.

1.4.4 Static dispatch

Our final hypothesis is that static dispatch will be possible from the call site in many cases. By using static dispatch, thus avoiding the overhead from dynamic dispatch, and with optimization for the first two reasons, we hypothesize that the execution speedup from ACL will outgrow its overhead.

1.5 Paper organization

The rest of this paper is organized as follows. Chapter 2 presents our experiments. Chapter 3 gives our experimental results, with our interpretation and conclusion. Chapter 4 discusses the issues and the ideas that we had when carrying out this experiment. And finally, Chapter 5 summarizes our results and discusses some future work.

CHAPTER 2 EXPERIMENTAL DESIGN

2.1 Setup

For testing our hypothesis we apply ACL to an existing compiler and measure the performance difference between compiling code with and without ACL features. For this result to be meaningful the compiler that is to be changed had to had the following features:

- It had to be open source, so we can change it.
- It had to be doing aggressive optimization, and use aliasing information in doing these optimizations so that our hypothesis can be effective.
- The compiled language had to support either passing by reference or passing pointers so that situations in which there is aliasing between the arguments and the globals may occur.
- The compiled language should preferably be a widely used programming language so that the practical benefits of ACL can be seen which may help in its adoption in real life software projects.

For all these reasons, our choice was GCC (the GNU Compiler Collection). Our goal was to apply ACL to the C front end of this compiler suite.

The development machine was an AMD-K2 380 MHz with 192MB of RAM, running RedHat Linux 6.2, and the version of GCC modified was GCC 2.95.2 with Pentium specific optimizations.

2.2 Overview of GCC

Before presenting the changes that had to be made to GCC to support ACL, it is helpful to have an overview of how GCC is structured. It has to be noted here, that code efficiency was paramount in GCC, and this was sometimes at the expense of clarity, so globals, for example, are used extensively and across files; this makes it, sometimes, hard to understand the internals of GCC in a modular way, rather, one has to have a grasp of the whole picture.

The only concession GCC makes to understandability, and to achieve a level of information hiding, most structures in GCC are accessed through accessor macros. Thus to change the way a data structure is defined, one only has to change the definition of these macros – provided that the data structures are not accessed directly.

2.2.1 The C grammar file

GCC uses bottom-up (LALR) parsing, and it generates its parser from a grammar file using bison[1]. Both C and ObjectiveC share the same input grammar file, GCC/gcc/c-parse.in[2], and the C specific parts are surrounded by `ifc` and `end ifc` pairs, while the ObjectiveC specific parts are surrounded by `ifobjc` and `end ifobjc` pairs. At the time when GCC itself is compiled, that input grammar file is split into two files, one of which, GCC/gcc/c-parse.y, is the one used to generate the C parser.

To avoid confusion in this paper, when we refer to the grammar we will use the generated C grammar file, but it must be clear that all changes were actually made in the input file, and they were always surrounded by the `ifc` and `end ifc` pairs so that they don't affect the generated ObjectiveC grammar file.

2.2.2 Trees in GCC

GCC consists of many front ends and many back ends, so it can compile different source languages[3], and target different platforms. Information is passed from front ends to back ends

[1] Bison is a Yacc-like parser generator.
[2] GCC is the root of the gcc source tree.
[3] As of version 2.95.2 GCC can compile C, C++, Objective C, Fortran 77, Chill, and Java.

using two trees, one of which, called "`tree`" is mainly used in the front end, and the other called "`rtx`" and is mainly used in the back end.

The `tree` nodes, defined in GCC/gcc/tree.h, hold information about symbol table entries. This information is somewhat common to all programming languages compiled by GCC, with some room for language specific information. In the case of C, this language specific information is defined in GCC/gcc/c-tree.h.

A `tree` node is a pointer to a union of structures, each of which is a structure that represent a different kind of entity; so for example there is a `tree_decl` structure that represents a declaration (either variable, parameter, label, typedef or function declaration), `tree_identifier` which represents an identifier, `tree_list` which represents a list of other `tree` nodes, and `tree_type` which represents types. All `tree` nodes share as their first element a `tree_common` structure which holds data common to all of them, including a tag that differentiates between different union members.

Access to different structure members is done through accessor macros defined with the different tree structures. These macros, in addition to being used to access the structure members, can be used to type check `tree` nodes (this can be done with special flags in the makefiles.)

The `rtx` nodes, defined in GCC/gcc/rtl.h, represent source code statements when converted to a platform independent *Register Transfer Language*, which will be optimized and then converted to platform specific assembly language by the back end.

2.2.3 Identifiers & Symbol lookup in C

During the lexical analysis phase, defined in GCC/gcc/c-lex.c, each identifier is returned as a `tree_identifier` node. Different identifiers sharing the same name will return the same identifier node; i.e., there exists only one `tree_identifier` node for each unique name throughout the compilation process. The macro `IDENTIFIER_POINTER` returns a pointer to the string representing the name of an identifier.

For symbol lookup purposes, the C front end adds, among other things, two members to the identifier node, which can be accessed with the macros `IDENTIFIER_GLOBAL_VALUE` and

IDENTIFIER_LOCAL_VALUE. The first points to a tree_decl corresponding to the global declaration of that identifier (if any), and the second points to the local declaration visible in the current binding level. These two values help make symbol lookup – which is a frequent operation when parsing – very efficient; all that has to be done, is to use the identifier node returned from the lexer and check one of these two values depending on the context.

Symbol lookup in the C front end is done through the two functions lookup_name(name) and lookup_name_current_level(name) defined in GCC/gcc/c-decl.c, where the first does the expected symbol lookup and the second only searches in the current binding level at the time of lookup.

2.2.4 Chains and lists

Throughout the compilation process, lists are used extensively. Each tree node has a chain member that can be used to chain that node to other nodes. To enable tree nodes to share in more than one list, there is also a special tree structure called tree_list. These tree_list nodes can be chained together to form a list, with each node pointing to two tree elements. The contents of a tree_list node are accessed using the macros TREE_PURPOSE and TREE_VALUE. The semantics of TREE_PURPOSE and TREE_VALUE depend on the location these lists are used.

2.2.5 Binding levels in C

A binding level is associated with each segment of code that can have its own declarations. The binding level is used to bind a name to its value in that code segment. In C, binding levels can nest, with declarations in inner binding levels *shadowing* declarations with the same name in outer binding levels. In GCC a binding level is represented by a binding_level structure defined in GCC/gcc/c-decl.c, shown in appendix A.2, with the current binding level pointed to by the current_binding_level global variable.

2.2.6 The compilation process

Instead of building the syntax tree for the whole compilation unit – a preprocessed source file – GCC compiles one function at time all the way to assembly language. During compilation,

GCC maintains symbol table information in `tree` nodes. Information related to global symbols is persistent throughout the compilation of the compilation unit. When parsing the body of a function, each statement is converted to `rtx` nodes, which represent the syntax tree of that statement. At the end of the function, the syntax tree of the whole function is optimized and converted into assembly language.

In the case of inlined functions, or when doing aggressive optimization, the optimized syntax tree of the function is stored in its symbol table node to be used later.

2.2.7 Memory allocation in GCC

To streamline memory allocation in GCC, special functions for memory allocation are used. When memory is allocated for an object – a `tree` node or an `rtx` node – that object is put on an 'object stack'. Depending on the type of the object, and the phase of compilation, different objects will be put on different stacks, with each stack having its own life span. For example, all identifiers will be on the permanent stack, and will persist from the point of their creation up to the end of the compilation. Globals will also reside on the permanent stack, while local variables and function parameters will be on the function stack which is flushed at the end of each function. There is other stacks that are emptied at the end of an expression, these usually hold the syntax tree generated when evaluating subexpressions. Object stacks themselves are defined in GCC/libiberty/obstack.c, while the code that selects which type of object resides on which stack, and when to free the objects on each stack is defined in GCC/gcc/tree.c.

2.3 Code changes

To support ACL functions, we had to first implement the code for the imports clause which will make only imported globals visible inside the function, and we had to implement multi-bodies with the code to select the body that matches the aliasing pattern for each call. We added a structure called `acl_function` to GCC/gcc/c-decl.c, shown in appendix A.4. This structure has the main information needed to emit the code needed for the multiple bodies; it holds the decision tree that is used for the dynamic dispatch code, and a list of the globals used in that function. The global variable `current_acl_function` points to the current ACL

function being compiled, which is analogous to the `current_function_decl` global variable used originally in GCC.

To increase code efficiency, these structures are reused when not needed, instead of allocating new structures each time. The code for reusing these structures was modeled after similar code in the C front end, and is shown in appendix A.5.

2.3.1 The 'imports' clause

When implementing the imports clause, we had to implement two main pieces of functionality. The first was making only the imported globals available inside the function body, which means name lookup must be modified. The second was allowing the type of a global to be restricted inside that body. All globals had to return to their original types after the end of the ACL function, and this must still work correctly with nested functions as they are supported in GCC.

2.3.1.1 Supporting more restrictive types

Since each variable declaration – `VAR_DECL` – structure points to a type structure, all that has to be done to support the more restrictive types was changing the pointer inside the variable declaration to the new type before parsing the function, and changing it back to the original type at the very end of that function. This was done with a chain of `acl_old_decl` structures stored in the `current_acl_function`. When parsing the imports clause, the type of the global is changed to the new type, and the original type is stored in an `acl_old_decl` structure. When parsing the body of the function, any of the globals that are declared in the imports clause will have the type pointed to by its `DECL` structure, which is the type declared in the imports clause. At the end of the function, the chain of `acl_old_decl` structures will be traversed, and the type of each imported global will be changed to the one stored in the `acl_old_decl` structure, which is the type of that global before starting that ACL function.

Figure 2.1 shows a part of the `acl_old_decl` chain, with one global variable which originally had `TYPE A`, but was declared with `TYPE B` in the imports clause.

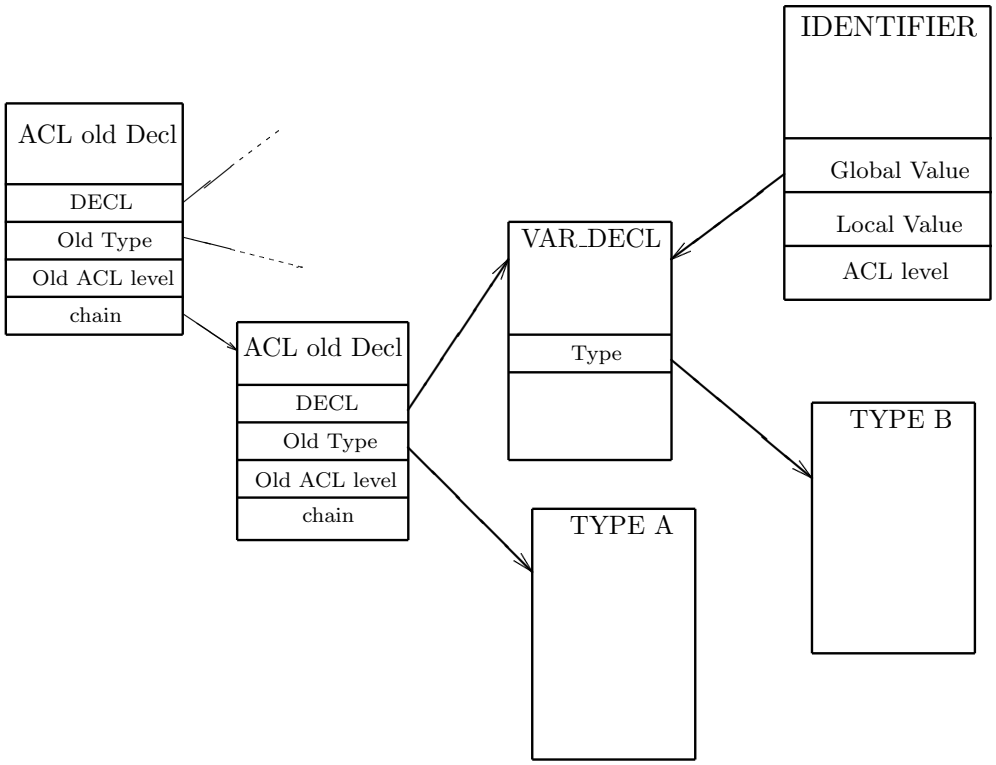

Figure 2.1 The globals after an imports clause

When creating the `acl_old_decl` chain, the chain elements are sorted in alphabetical order, using each global's name; this will be important when emitting the code for the multi-bodies, as will be explained later.

2.3.1.2 Name lookup with imported globals

Since GCC supports nested functions, name lookup had to be changed to accommodate ACL functions and imported globals. The main concern here was to keep name lookup as efficient as it originally was. We added a variable `acl_level` to the binding level structure, which stores the nesting level of ACL functions. So, in a nested function chain, the acl level for all bindings outside of ACL functions will be zero, and starting from the binding level for the first ACL function will be one, and it will be incremented each time an ACL function is met in

that nesting chain. We also made each binding level point to its corresponding `FUNCTION_DECL`. A variable `acl_level` was also added to the `IDENTIFIER` structure as shown in appendix A.3.

Initially the `acl_level` for all identifiers – and thus for their corresponding variable declarations – is zero. An imported global will take the `acl_level` of the binding level corresponding to the function in which it is imported.

At name lookup time, first, the original lookup is done return a `DECL`; this `DECL` is checked to make sure that either its `acl_level` matches that of the current binding level – which means it was imported in the inner-most ACL function – or, that from the point of lookup up to the point in which that variable was defined (given by the macro `DECL_CONTEXT`) we don't encounter an ACL function.

If one of the previous two conditions is met, the declaration from the original name lookup is returned, otherwise NULL is returned which signals name lookup failure. Name lookup is done through the function '`lookup_name`' defined in GCC/gcc/c-decl.c, with another function '`lookup_name_in_acl_imports`' used for name lookup when parsing the imports clause, which goes one binding level up before doing regular name lookup, so that parameters from the current function are not considered when doing name lookup for imported globals.

2.3.2 Multi-bodies

For a multi-body C/ACL function, each body will be translated into a regular C function at compilation time, and another function will be generated for the dynamic dispatch. So, for example, the resulting assembly code for a 3-body ACL function will be 4 assembly functions, one corresponding to each body and one containing the dynamic dispatch code which matches the aliasing pattern between the arguments and the globals.

2.3.2.1 Body naming

Since each single ACL function will map to multiple assembly functions, a consistent naming scheme must be used to name these assembly functions. As an example, let us consider the function '`int foo (int *x, int *y, int *z) imports (m, l)`', and let us suppose that this function has three bodies, with the two additional bodies have alias lists '`alias (x, y)`'

and '`alias (y, z, l; x, m)`'. When compiled, this function will be translated into four functions. The one doing the dynamic dispatch will be called '`foo`' as it is the entry point to that function. The first body, in which there is no aliasing between the parameters and the imported globals, will be called `_foo$acl` where the name `foo` is prefixed by an under-bar and `$acl` is appended to it. For the second body, where the first parameter is aliased to the second parameter, the name will be '`_foo$acl_ab`' where the `_ab` part specifies the alias list with lower-case characters corresponding to parameters, where '`a`' corresponds to the first parameter, '`b`' corresponds to the second and so on. The name of the third body will be '`_foo$acl_aB_bcA`', where `_bcA` corresponds to the first alias list, and `_aB` corresponds to the second one. In such lists upper-case characters correspond to imported globals, where '`A`' corresponds to the first imported global when sorted alphabetically, and '`B`' corresponds to the second one and so on. To insure unique names, the parts corresponding to alias lists are sorted alphabetically, i.e. a body with an alias list '`alias (x, m; y, z, l)`' will generate the same name as the one given above for the alias list `alias (y, z, l; x, m)`. The code for crafting these names is in the function '`process_alias_lists`', shown in appendix A.14, in GCC/gcc/c-decl.c. That function uses the function '`acl_get_id_symbol`', shown in appendix A.16, which returns the character corresponding to an identifier.

2.3.2.2 Generating functions from bodies

When parsing an ACL function, each body will be translated into a function. For each body, an `acl_body` structure, shown in appendix A.6, will be created. When processing the alias list of each body, the function generated from that body will be given a name as described above. At the same time, all the identifiers that are not first in an *alias_list* will be rendered invisible inside that function. Parameters will be removed from the list used when creating that function. And, globals will be restored to their original state, including their original `acl_level`, which will make them invisible inside that function. At the end of the body, the state of these globals will be set to their state just after the imports clause, thus making them visible to subsequent bodies. Hiding these variables is done by the function `acl_remove_id` shown in appendix A.16. And, restoring the globals is done by the function `acl_finish_body`

shown in appendix A.12.

When generating these functions, we had to instruct the compiler that there is no aliasing between the arguments and the globals. This way the compiler can use this piece of information to perform some optimizations which otherwise would not be possible. We use the '`flag_argument_noalias`' flag, originally in GCC, for that purpose. We set that flag before optimizing ACL bodies, and restore it to its original value afterwards.

2.3.3 The Dynamic Dispatch code

2.3.3.1 Creating the decision tree

At the same time as the alias lists of each body are processed, the part of the decision tree leading to that body is created. The branch of the tree corresponding to the first body, in which there is no aliasing, will contain comparisons between pairs of each of the parameters that have a pointer type and the imported globals. As an example, after starting the first body of a function `bar` with a prototype '`void bar(int *x, int *y, int *z) imports(m)`' the decision tree will look like the one at the top left corner of Figure 2.2 (the arrow pointing to the right is the 'not alias' path). In that figure, as well as in the code for the decision tree, each variable is represented by the character used for it in the function name.

For each subsequent body, a branch will be constructed. The branches for the bodies need not contain all the comparisons, since by knowing at some point that two variables are aliases we only need to compare one of them to the other variables. Continuing on the previous example, if the second body has the alias list '`| alias (x, y, z)`' then the first comparison between `a` and `b` must follow the 'alias' arrow (to the left). After that comparison we do not need to compare `b` to anything else. This will be done in the code by setting the 'ignore' flag in the decision tree node structure for all nodes containing `b` for that body. By ignoring these redundant checks we increase the efficiency of the code generated from that decision tree. Figure 2.2 shows the tree after starting that body, as well as after starting two other bodies with alias lists '`| alias (y, m)`' and '`| alias (x, z; y, m)`' respectively.

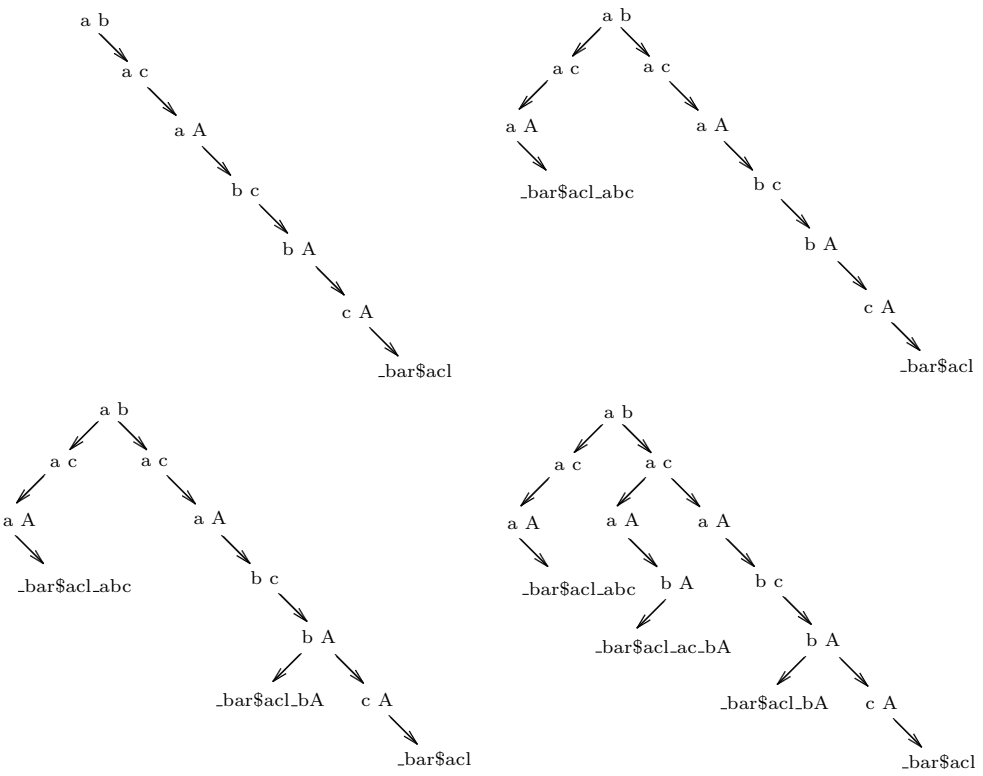

Figure 2.2 Stages in building an example decision tree

2.3.3.2 Converting the decision tree into code

After converting all the bodies into functions, the dynamic dispatch code is generated from the decision tree. At each node of the tree, the two variables in that node will be compared. In the case of parameters, the variable itself will be compared for equality as it is a pointer. In the case of global variables, the address of that global variable will be compared.

To increase the run-time efficiency of the generated code, the checks from all the connected nodes with only one arrow going out of them will be grouped into one condition as shown in Figure 2.3. Thus the decision tree from the previous example will be converted to code similar to that in Figure 2.4. In this code, nodes with both 'alias' and 'not-alias' arrows will

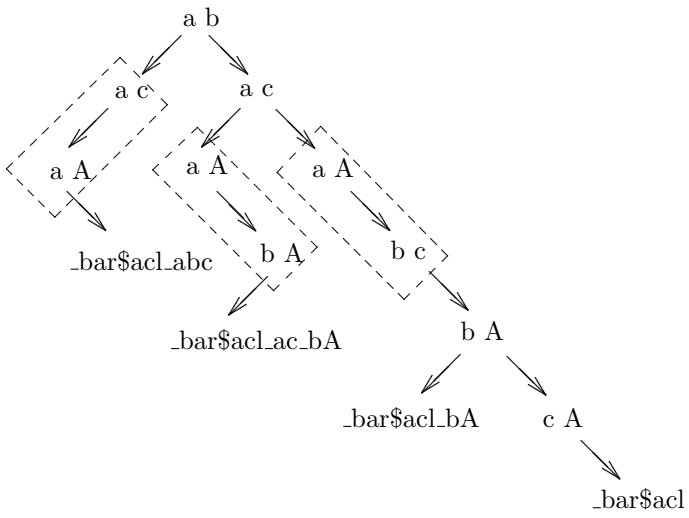

Figure 2.3 Comparisons grouped in the decision tree

be translated into an `if ...else` statement. Nodes with only one arrow will be translated to `if` statements without an `else` part, which can fall through to the error reporting code.

2.3.4 Static dispatch

Originally, we planned to do the static dispatch at the call site. We had to analyze the arguments at each call site and, based on the aliasing pattern, call the appropriate body. But when we profiled one of the benchmarks with only the dynamic dispatch code, and with aggressive optimization, we found that GCC inlines the call to the ACL body from the dynamic dispatch code in the original call site. Given that what GCC was doing was enough for the purposes of our experiment, we relied on GCC to do the static dispatch instead of implementing it ourselves.

```
if ( x != y ){
  if ( x != z ){
    if ( x != &m && y != z ){
      if ( y != &m ){
        if ( z != &m ){
          _bar$acl (x, y, z);
          return;
        }
      } else {
        _bar$acl_bA (x, y, z);
        return;
      }
    }
  } else {
    _bar$acl_ac_bA (x, y);
    return;
  }
} else {
  _bar$acl_abc (x);
  return;
}

/* Signal error or abort */
```

Figure 2.4 Code generated from the decision tree

CHAPTER 3 RESULTS

To test our compiler we used benchmarks from the SPEC CPU2000 benchmark suite. We chose to use that suite because it will ensure that we are thoroughly testing the compiler with real programs and big input data sets. Moreover, by changing real programs, this can unearth situations in which using ACL is easy and straight forward, and other situations in which it will not give the desired improvement.

When running the SPEC benchmarks, by default each benchmark is run three times. At the end of these runs, the median of the three results is reported as the result of the benchmark. The reported result is the ratio between a reference time for the benchmark and the measured time. This means that faster executions yield higher results.

It has to be noted here that originally the SPEC suite is designed to compare machines or optimization techniques. This means that initially, it is not intended for the source of the benchmarks to be changed. In our case, we are not using it to compare our machine to other machines, but rather to compare the effect of using ACL in the code. So, it should be clear that the results we are getting can not serve for comparison with other machines.

3.1 Testing methodology

For each of the benchmarks that we used in our testing, we first profiled the unmodified source. Using the profile data, we located the candidate functions to be converted to ACL. These functions had to be taking a large amount of the execution time, and of course, they had to be passing pointers or using globals.

After converting the chosen functions to C/ACL, we ran the benchmark suite twice, the first time using the original source code, and then using the modified source code. In both cases we used the same optimization flags.

To see the maximum effect that we can get from using ACL, we used the most aggressive optimization possible in GCC. Finally, we tested one of the benchmarks with less aggressive optimization to measure the effect of ACL when static dispatch is not used.

To ensure consistency in the results, all tests were run from the console with no other users logged in, and most unnecessary daemons were stopped.

3.2 Test results

We ran the test on six benchmarks: parser, mcf, gcc, vortex, gap, and twolf. Parser is an English language parser, it takes as input a sequence of sentences and returns as output the analysis of the input sentences. MCF is a combinatorial optimization program, it generates the timetable for a single-depot vehicle schedule. GCC is an earlier version[1] of gcc, it includes only the C compiler which takes as input preprocessed C files and generates as output 88100 assembly code files. VORTEx is a single-user object-oriented database transaction benchmark. Gap implements a language and library designed mostly for computing in groups (GAP is an acronym for Groups, Algorithms and Programming). And the twolf placement and global routing package is used in the process of creating the lithography artwork needed for the production of microchips.

The compilation flags used for aggressive optimization where `-O3`, `-fomit-frame-pointer` and `-fstrict-aliasing`. The last flag instructs the compiler that a pointer will only point to its declared type. This flag had to be used as otherwise GCC assumes that any pointer can point to any memory location, and skips the alias analysis phase.

MCF, gcc, gap [2], and vortex had good candidate functions. By converting these functions to ACL, we achieved improvement in the execution time as reported in Tables 3.1, 3.2, 3.3, and 3.4 respectively.

After profiling 'parser' and 'twolf', we found that most of the candidate functions to be converted to ACL where not consuming a lot of the execution time, or did not benefit from the optimization. ACL did not help in that case, and the results were as shown in Table 3.5

[1] The benchmark was based on GCC version 2.7.2.2

[2] When running gap, we ran it only once, instead of three times.

and Table 3.6 respectively.

To see the effect of disabling static dispatch, we benchmarked gcc with only the `-O2` flag. This in effect instructs the compiler to do less optimization than the previous tests, and without doing alias analysis. In that run, the original code scored 26.3, while the ACL code scored 25.5, a performance loss of 3%. This may indicate that even the overhead from dynamic dispatch is acceptable, if ACL is used for other reasons besides optimization, such as better ability to reason about the code.

Table 3.1 Results for mcf

		Original		With ACL		
Run	Ref. Time	Run Time	Ratio	Run Time	Ratio	
1	1800	4086	44.1	4177	43.1	
2	1800	4278	42.1	3930	45.8	
3	1800	4255	42.3	3870	46.5	
Reported	1800	4255	42.3	3930	45.8	8.3%

Table 3.2 Results for gcc

		Original		With ACL		
Run	Ref. Time	Run Time	Ratio	Run Time	Ratio	
1	1100	4089	26.9	3746	29.4	
2	1100	4106	26.8	3835	28.7	
3	1100	3582	30.7	4027	27.3	
Reported	1100	4089	26.9	3835	28.7	6.7%

Table 3.3 Results for gap

		Original		With ACL		
Run	Ref. Time	Run Time	Ratio	Run Time	Ratio	
Reported	1100	12985	8.47	12719	8.65	2.1%

Table 3.4 Results for vortex

		Original		With ACL		
Run	Ref. Time	Run Time	Ratio	Run Time	Ratio	
1	1900	4171	45.5	3196	59.4	
2	1900	3956	48.0	3218	59.0	
3	1900	4206	45.2	3327	57.1	
Reported	1900	4171	45.5	3218	59.0	29.6%

Table 3.5 Results for parser

		Original		With ACL		
Run	Ref. Time	Run Time	Ratio	Run Time	Ratio	
1	1800	4016	44.8	4055	44.4	
2	1800	3668	49.1	3832	47.0	
3	1800	3648	49.3	4031	44.7	
Reported	1800	3668	49.1	4031	44.7	-8.9%

Table 3.6 Results for twolf

		Original		With ACL		
Run	Ref. Time	Run Time	Ratio	Run Time	Ratio	
1	3000	6240	48.1	6827	43.9	
2	3000	6423	46.7	6620	45.3	
3	3000	6171	48.6	6569	45.7	
Reported	3000	6240	48.1	6620	45.3	-5.8%

CHAPTER 4 DISCUSSION

4.1 Decisions we made in our implementation

The paper on which this experiment was based [10] had special treatment for arrays. It counted an argument to be an alias to the array if it is an alias to one of its members. But since in C arrays are essentially pointers, we did not have the same behavior in our implementation. Our notion of aliasing is when two pointers are pointing to the same memory cell, i.e. a pointer will be an alias to an array only if the pointer points to the first element of the array.

When implementing the code that handles the alias list, we had to decide what should be allowed to be aliases, and what should not. One question was whether or not different types could be aliases or not. For example, should we allow a pointer to an integer to be an alias to a pointer to a char? And whether or not pointers with different type qualifiers should be allowed to be aliases. For example, should we allow a pointer to an integer to be an alias to a constant pointer to an integer? In keeping with the spirit of C, which puts all the power in the hands of the developer, we allowed both situations, giving only warnings when they occur. But it should be noted that these should handled with extreme care, are they can lead to unexpected results when used with aggressive optimization, as the compiler assumes that a pointer will only point to its type, and a constant will actually be constant.

4.2 Pragmatic value of ACL

Beside measuring the performance improvement that we can gain from using ACL, we were interested in finding out whether it will be easy to adopt ACL. And if it can expand beyond the theory into the practical world. There is two main hurdles that we thought could face someone embracing ACL.

Firstly, by having to write new bodies to a function may mean that the semantics of the

function must be understood. This can be a big deterrent from ACL to other optimization techniques that do not require human intervention. But from our experience in applying ACL to the benchmarks, we found that this is not true. In most cases, the additional bodies will be very simple. Even when the bodies can not be deduced easily from the original function, one can copy the same function to all the bodies, and substitute the head of each alias list for that list members.

Secondly, there is the problem of choosing which functions to convert. We found that by profiling the program with a representative input data, and finding the bottlenecks, candidate functions can be easily found in many programs. But we found that it is sometimes cumbersome to convert functions with more than three or four parameters, as the number of required bodies grows exponentially with the number of parameters.

CHAPTER 5 CONCLUSION AND FUTURE WORK

In this paper, we investigated the efficiency of code generated by GCC when actual parameters to a procedure are guaranteed not to have any aliasing between each other and between them and global variables. Unlike some of the other approaches for dealing with aliases, parameters in each body are guaranteed to be alias-free. In the same time, our approach does not limit the aliasing pattern at the call site, thus giving extra freedom to the developer.

Our experiment showed that it is fairly easy to convert regular C functions into ACL-style functions. This can be achieved in some cases, even without understanding the semantics of the converted function.

By choosing suitable functions to convert, improvement in the performance can be achieved. This improvement is attributed mainly to the ability of the compiler to optimize the function knowing that it has alias-free parameters. The overhead from dynamic dispatch can be minimized by using static dispatch whenever possible. When the improvement from optimization outgrows the dynamic dispatch overhead, a net gain in code efficiency is achieved.

There are some directions in which this experiment can be extended. We are planning to study the effect of varying the number of functions converted to ACL. We will also make some experiments to find the functions and optimizations that benefit most from ACL. This can broaden the criteria upon which functions are chosen to be converted to ACL. We are also planning to study the effect of converting the C library to ACL on the overall code performance.

In this paper we were mainly interested in code optimization. But having alias free parameters can also be useful in program verification. It would be interesting to study the effect of our work in making verification easier. It would also be interesting to see if some tools can do the conversion to ACL automatically. Finally, the same experiment can be applied to an object oriented language to study its applicability in more complex situations.

APPENDIX SOURCE CODE

This appendix contains source code that is relevant to the discussion in the paper. These code segments are represented in diff-style; i.e., whenever a source line is deleted from the code it will be preceded by a '-' sign, and whenever a line is added it will be preceded by a '+' sign. This code is first copied verbatim from the GCC source code, it is then formatted using the `indent` program so that it fits the page width, and finally the diff signs are added manually. In some cases, when only few lines are added to very large functions, only the parts around the added lines are shown. Whenever code from a function is not shown, a comment saying "code removed" will be put in its place.

A.1 Changes made to the grammar

From GCC/gcc/c-parse.y

```
 fndef:
        typed_declspecs setspecs declarator
+       acl_imports
          { if (! start_function (current_declspecs, $3,
                      prefix_attributes, NULL_TREE, 0))
              YYERROR1;
            reinit_parse_for_function (); }
        old_style_parm_decls
+       compstmt_start
          {
            store_parm_decls ();
+           if($4){
+             acl_start_function();
+             acl_save_globals(TREE_CHAIN($4));
+             if(!acl_start_first_body(0))
+               YYERROR1;
+             reinit_parse_for_function();
+           }
          }
```

```
-       compstmt_or_error
+       acl_compstmt_or_error
          {
+           if(DECL_ACL_FLAG(acl_last_function_decl))
+             compstmt_count=
+               acl_finish_function(0,compstmt_count);
            finish_function (0);
            current_declspecs = TREE_VALUE (declspec_stack);
            prefix_attributes = TREE_PURPOSE (declspec_stack);
            declspec_stack = TREE_CHAIN (declspec_stack);
            resume_momentary($2);
          }
      | typed_declspecs setspecs declarator error
          { current_declspecs = TREE_VALUE (declspec_stack);
            prefix_attributes = TREE_PURPOSE (declspec_stack);
            declspec_stack = TREE_CHAIN (declspec_stack);
            resume_momentary ($2); }
      | declmods setspecs notype_declarator
+       acl_imports
          { if (! start_function (current_declspecs, $3,
                                  prefix_attributes, NULL_TREE, 0))
              YYERROR1;
            reinit_parse_for_function (); }
          old_style_parm_decls
+       compstmt_start
          {
            store_parm_decls ();
+           if($4){
+             acl_start_function();
+             acl_save_globals(TREE_CHAIN($4));
+             if(!acl_start_first_body(0))
+               YYERROR1;
+             reinit_parse_for_function();
+           }
          }
-       compstmt_or_error
+       acl_compstmt_or_error
          {
+           if(DECL_ACL_FLAG(acl_last_function_decl))
+             compstmt_count=
+               acl_finish_function(0,compstmt_count);
            finish_function (0);
            current_declspecs = TREE_VALUE (declspec_stack);
            prefix_attributes = TREE_PURPOSE (declspec_stack);
            declspec_stack = TREE_CHAIN (declspec_stack);
            resume_momentary($2);
```

```
          }
      | declmods setspecs notype_declarator error
          { current_declspecs = TREE_VALUE (declspec_stack);
            prefix_attributes = TREE_PURPOSE (declspec_stack);
            declspec_stack = TREE_CHAIN (declspec_stack);
            resume_momentary ($2); }
      | setspecs notype_declarator
+       acl_imports
          { if (! start_function (NULL_TREE, $2,
                      prefix_attributes, NULL_TREE, 0))
              YYERROR1;
            reinit_parse_for_function (); }
        old_style_parm_decls
+       compstmt_start
          {
            store_parm_decls ();
+           if($3){
+             acl_start_function();
+             acl_save_globals(TREE_CHAIN($3));
+             if(!acl_start_first_body(0))
+               YYERROR1;
+             reinit_parse_for_function();
+           }
          }
-       compstmt_or_error
+       acl_compstmt_or_error
          {
+           if(DECL_ACL_FLAG(acl_last_function_decl))
+             compstmt_count=acl_finish_function(0,compstmt_count);
            finish_function (0);
            current_declspecs = TREE_VALUE (declspec_stack);
            prefix_attributes = TREE_PURPOSE (declspec_stack);
            declspec_stack = TREE_CHAIN (declspec_stack);
            resume_momentary($1);
          }
      | setspecs notype_declarator error
          { current_declspecs = TREE_VALUE (declspec_stack);
            prefix_attributes = TREE_PURPOSE (declspec_stack);
            declspec_stack = TREE_CHAIN (declspec_stack);
            resume_momentary ($1); }
      ;

  initdcl:
-       declarator maybeasm maybe_attribute '='
+       declarator acl_imports maybeasm maybe_attribute '='
          { $<ttype>$ = start_decl ($1, current_declspecs, 1,
```

```
-                       $3, prefix_attributes);
-           start_init ($<ttype>$, $2, global_bindings_p ()); }
+                       $4, prefix_attributes);
+           start_init ($<ttype>$, $3, global_bindings_p ()); }
         init
           {
             finish_init ();
-           finish_decl ($<ttype>5, $6, $2);
+           finish_decl ($<ttype>6, $7, $3);
+           if($2){
+             DECL_ACL_GLOBS($<ttype>6)=TREE_CHAIN($2);
+             DECL_ACL_FLAG($<ttype>6)=1;
+           }
+           $$ = $<ttype>6;
           }
-     | declarator maybeasm maybe_attribute
+     | declarator acl_imports maybeasm maybe_attribute
           { tree d = start_decl ($1, current_declspecs, 0,
-                                 $3, prefix_attributes);
-           finish_decl (d, NULL_TREE, $2);
+                                 $4, prefix_attributes);
+           finish_decl (d, NULL_TREE, $3);
+           if($2){
+             DECL_ACL_GLOBS(d)=TREE_CHAIN($2);
+             DECL_ACL_FLAG(d)=1;
+           }
+           $$ = d;
           }
     ;


  notype_initdcl:
-       notype_declarator maybeasm maybe_attribute '='
+       notype_declarator acl_imports maybeasm maybe_attribute '='
           { $<ttype>$ = start_decl ($1, current_declspecs, 1,
-                       $3, prefix_attributes);
-           start_init ($<ttype>$, $2, global_bindings_p ()); }
+                       $4, prefix_attributes);
+           start_init ($<ttype>$, $3, global_bindings_p ()); }
         init
           {
             finish_init ();
-           decl_attributes ($<ttype>5, $3, prefix_attributes);
-           finish_decl ($<ttype>5, $6, $2);
+           decl_attributes ($<ttype>6, $4, prefix_attributes);
+           finish_decl ($<ttype>6, $7, $3);
```

```
+           if($2){
+             DECL_ACL_GLOBS($<ttype>6)=TREE_CHAIN($2);
+             DECL_ACL_FLAG($<ttype>6)=1;
+           }
+           $$ = $<ttype>6;
          }
-     | notype_declarator maybeasm maybe_attribute
+     | notype_declarator acl_imports maybeasm maybe_attribute
          {
            tree d = start_decl ($1, current_declspecs, 0,
-                                $3, prefix_attributes);
-           finish_decl (d, NULL_TREE, $2);
+                                $4, prefix_attributes);
+           finish_decl (d, NULL_TREE, $3);
+           if($2){
+             DECL_ACL_GLOBS(d)=TREE_CHAIN($2);
+             DECL_ACL_FLAG(d)=1;
+           }
+           $$ = d;
          }
    ;

  nested_function:
        declarator
+       acl_imports
          { push_c_function_context ();
            if (! start_function (current_declspecs, $1,
                      prefix_attributes, NULL_TREE, 1))
              {
                pop_c_function_context ();
                YYERROR1;
              }
            reinit_parse_for_function (); }
        old_style_parm_decls
+       compstmt_start
          {
            store_parm_decls ();
+           if($2){
+             acl_start_function();
+             acl_save_globals(TREE_CHAIN($2));
+             if(!acl_start_first_body(1))
+               YYERROR1;
+             reinit_parse_for_function();
+           }
          }
  /* This used to use compstmt_or_error.
```

```
     That caused a bug with input `f(g) int g {}',
     where the use of YYERROR1 above caused an error
     which then was handled by compstmt_or_error.
     There followed a repeated execution of that same rule,
     which called YYERROR1 again, and so on.  */
-       compstmt
+       acl_compstmt
          {
+           if(DECL_ACL_FLAG(acl_last_function_decl))
+             compstmt_count=
+               acl_finish_function(1,compstmt_count);
            finish_function (1);
            pop_c_function_context ();
+           acl_last_function_decl=current_function_decl;
          }
      ;

  notype_nested_function:
        notype_declarator
+       acl_imports
          { push_c_function_context ();
            if (! start_function (current_declspecs, $1,
                      prefix_attributes, NULL_TREE, 1))
              {
                pop_c_function_context ();
                YYERROR1;
              }
            reinit_parse_for_function (); }
        old_style_parm_decls
+       compstmt_start
          {
            store_parm_decls ();
+           if($2){
+             acl_start_function();
+             acl_save_globals(TREE_CHAIN($2));
+             if(!acl_start_first_body(1))
+               YYERROR1;
+             reinit_parse_for_function();
+           }
          }
  /* This used to use compstmt_or_error.
     That caused a bug with input `f(g) int g {}',
     where the use of YYERROR1 above caused an error
     which then was handled by compstmt_or_error.
     There followed a repeated execution of that same rule,
     which called YYERROR1 again, and so on.  */
```

```
-       compstmt
+       acl_compstmt
          {
+           if(DECL_ACL_FLAG(acl_last_function_decl))
+             compstmt_count=
+               acl_finish_function(1,compstmt_count);
            finish_function (1);
            pop_c_function_context ();
+           acl_last_function_decl=current_function_decl;
          }
      ;

  /* This is the body of a function definition.
     It causes syntax errors to ignore to the next openbrace.*/
- compstmt_or_error:
+ acl_compstmt_or_error:
-       compstmt
+       compstmt_rest
          {
+           if(DECL_ACL_FLAG(acl_last_function_decl))
+              acl_finish_body();
          }
      | error compstmt
+         {
+           if(DECL_ACL_FLAG(acl_last_function_decl))
+              acl_finish_body();
+         }
+     | acl_compstmt_or_error '|' acl_alias_compstmt
+         { acl_finish_body(); }
      ;


+ acl_compstmt:
+       compstmt_rest
+         {
+           if(DECL_ACL_FLAG(acl_last_function_decl))
+              acl_finish_body();
+         }
+     | acl_compstmt '|' acl_alias_compstmt
+         { acl_finish_body(); }
+     ;


  compstmt_start: '{' { compstmt_count++; }

  compstmt:
```

```
+       compstmt_start compstmt_rest
+         { $$ = $2 ; }
+     ;
+
+ compstmt_rest:
-       compstmt_start '}'
+       '}'
          { $$ = convert (void_type_node, integer_zero_node); }
-     | compstmt_start pushlevel maybe_label_decls decls xstmts '}'
+     | pushlevel maybe_label_decls decls xstmts '}'
          { emit_line_note (input_filename, lineno);
            expand_end_bindings (getdecls (), 1, 0);
            $$ = poplevel (1, 1, 0);
            if (yychar == CONSTANT || yychar == STRING)
              pop_momentary_nofree ();
            else
              pop_momentary (); }
-     | compstmt_start pushlevel maybe_label_decls error '}'
+     | pushlevel maybe_label_decls error '}'
          { emit_line_note (input_filename, lineno);
            expand_end_bindings (getdecls (), kept_level_p (), 0);
            $$ = poplevel (kept_level_p (), 0, 0);
            if (yychar == CONSTANT || yychar == STRING)
              pop_momentary_nofree ();
            else
              pop_momentary (); }
-     | compstmt_start pushlevel maybe_label_decls stmts '}'
+     | pushlevel maybe_label_decls stmts '}'
          { emit_line_note (input_filename, lineno);
            expand_end_bindings (getdecls (), kept_level_p (), 0);
            $$ = poplevel (kept_level_p (), 0, 0);
            if (yychar == CONSTANT || yychar == STRING)
              pop_momentary_nofree ();
            else
              pop_momentary (); }
      ;

+ acl_imports:
+       /* empty */
+         {
+           $$ = NULL_TREE ;
+         }
+     | IMPORTS '(' acl_ids_or_parms_list ')'
+         {
+           $$ = chainon (build_tree_list (NULL_TREE,
+                                          NULL_TREE), $3);
```

```
+         }
+     ;
+
+ acl_ids_or_parms_list:
+       /* empty */
+         { $$ = NULL_TREE ; }
+     | acl_ids_or_parms
+         { $$ = $1 ; }
+     ;
+
+ acl_ids_or_parms:
+       parm
+         { $$ = build_tree_list (NULL_TREE, $1); }
+     | IDENTIFIER
+         { $$ = build_tree_list (NULL_TREE, $1); }
+     | acl_ids_or_parms ',' parm
+         { $$ = chainon ($1, build_tree_list (NULL_TREE, $3)); }
+     | acl_ids_or_parms ',' IDENTIFIER
+         { $$ = chainon ($1, build_tree_list (NULL_TREE, $3)); }
+     ;
+
+ acl_alias_compstmt:
+       ALIAS '(' acl_alias_lists ')'
+         {
+           if(!DECL_ACL_FLAG(acl_last_function_decl))
+             fatal("ACL - ACL functions must "
+                   "have an ‘imports’ clause");
+
+           if(!acl_start_body($3))
+             YYERROR1;
+           reinit_parse_for_function();
+         }
+       compstmt
+         { }
+     ;
+
+ acl_alias_lists:
+       identifiers
+         { $$ = build_tree_list (NULL_TREE, $1); }
+     | acl_alias_lists ';' identifiers
+         { $$ = chainon ($1, build_tree_list (NULL_TREE, $3)); }
+     ;
```

A.2 The binding_level structure

From GCC/gcc/c-decl.c

```
  struct binding_level {

    /* A chain of _DECL nodes for all variables, constants,
       functions, and typedef types.  These are in the
       reverse of the order supplied. */
    tree names;

+   /* ACL - the FUNCTION_DECL, if any, associated with this
+      level. I added this here, to connect a binding_level
+      created for a function definition with its
+      FUNCTION_DECL. The only other way I found to link a
+      binding level to its FUNCTION_DECL was to follow
+      c_function_chain, defined in this file, and
+      outer_function_chain defined in function.c, but I
+      found that to be very cumbersome. */
+   tree function_decl;
+
+   /* ACL - acl nesting level. For nested functions, this
+      will tell how deep we are in the nesting of acl-style
+      functions. This is used mainly to implement the
+      acl-aware lookup_name() efficiently */
+   unsigned int acl_level;

    /* A list of structure, union and enum definitions, for
       looking up tag names.  It is a chain of TREE_LIST
       nodes, each of whose TREE_PURPOSE is a name, or
       NULL_TREE; and whose TREE_VALUE is a RECORD_TYPE,
       UNION_TYPE, or ENUMERAL_TYPE node. */
    tree tags;

    /* For each level, a list of shadowed outer-level local
       definitions to be restored when this level is popped.
       Each link is a TREE_LIST whose TREE_PURPOSE is an
       identifier and whose TREE_VALUE is its old definition
       (a kind of ..._DECL node). */
    tree shadowed;

    /* For each level (except not the global one), a chain of
       BLOCK nodes for all the levels that were entered and
       exited one level down. */
    tree blocks;

    /* The BLOCK node for this level, if one has been
       preallocated. If 0, the BLOCK is allocated (if needed)
       when the level is popped. */
    tree this_block;
```

```
  /* The binding level which this one is contained in
     (inherits from). */
  struct binding_level *level_chain;

  /* Nonzero for the level that holds the parameters of a
     function. */
  char parm_flag;

  /* Nonzero if this level "doesn't exist" for tags. */
  char tag_transparent;

  /* Nonzero if sublevels of this level "don't exist" for
     tags. This is set in the parm level of a function
     definition while reading the function body, so that
     the outermost block of the function body will be
     tag-transparent. */
  char subblocks_tag_transparent;

  /* Nonzero means make a BLOCK for this level regardless
     of all else. */
  char keep;

  /* Nonzero means make a BLOCK if this level has any
     subblocks. */
  char keep_if_subblocks;

  /* Number of decls in `names' that have incomplete
     structure or union types. */
  int n_incomplete;

  /* A list of decls giving the (reversed) specified order
     of parms, not including any forward-decls in the
     parmlist. This is so we can put the parms in proper
     order for assign_parms. */
  tree parm_order;
};
```

A.3 The Identifier structure

From GCC/gcc/c-tree.h

```
 struct lang_identifier {
   struct tree_identifier ignore;
+  unsigned int acl_level;
   tree global_value, local_value, label_value,
```

```
      implicit_decl;
    tree error_locus, limbo_value;
  };

+   /* ACL - This represents the last acl_level that imported
+      this identifier */
+ #define IDENTIFIER_ACL_LEVEL(NODE)     \
+   (((struct lang_identifier *)(NODE))->acl_level)
```

A.4 The acl_function structure

From GCC/gcc/c-decl.c

```
+ struct acl_function {
+
+   /*
+    * This is the FUNCTION_DECL with the name used in the
+    * source file
+    */
+   tree decl;
+
+   /* these are the parameters passed to start_function() */
+   tree declarator;
+   tree parmlist;
+   tree declspecs;
+   tree prefix_attributes;
+   tree attributes;
+   int nested;
+
+   /*
+    * This is a pointer to the function name that must be
+    * changed for each body
+    */
+   tree *identifier_p;
+
+   /* acl globals imported in this function. */
+   struct acl_old_decl *globs;
+
+   /* this holds the decision tree for the function */
+   struct acl_decision_tree *dtree;
+
+   /* This points to the first body in the function. */
+   struct acl_body *first;
+
+   /* This points to the last body in the function. */
+   struct acl_body *last;
```

```
+
+   /* This is used to chain ACL functions together */
+   struct acl_function *chain;
+ };
```

A.5 Code to reuse acl_function structures

From GCC/gcc/c-decl.c

```
+ static const struct acl_function clear_acl_function =
+   { NULL, NULL, NULL, NULL, NULL, NULL, NULL, 0, NULL,
+   NULL, NULL, NULL
+ };
+
+ static struct acl_function *free_acl_function;
+ static struct acl_function *current_acl_function;
+
+ static struct acl_function *
+ make_acl_function ()
+ {
+   return (struct acl_function *)
+     xmalloc (sizeof (struct acl_function));
+ }
+
+ /*
+  * ACL - this should be called when we discover that we
+  * are in an acl-style function, it will make the list
+  * that will be used to track the bodies making this
+  * function
+  */
+ static void
+ new_acl_function ()
+ {
+   struct acl_function *p;
+
+   if (free_acl_function) {
+     p = free_acl_function;
+     free_acl_function = p->chain;
+   } else {
+     p = make_acl_function ();
+   }
+   *p = clear_acl_function;
+   p->chain = current_acl_function;;
+   current_acl_function = p;
+ }
+
```

```
+ /*
+  * ACL - after finishing an acl-style function, this
+  * should be called to remove its list from the chain, and
+  * return to the outer list (if we were in a nested acl
+  * function)
+  */
+ static void
+ pop_acl_function ()
+ {
+   struct acl_body *p, *t;
+   struct acl_function *f;
+
+   /* first free the acl_body structs... */
+   p = current_acl_function->first;
+   while (p) {
+     t = p->next;
+     p->next = free_acl_body;
+     free_acl_body = p;
+     p = t;
+   }
+
+   /* ...and then free the acl_function struct */
+   f = current_acl_function;
+   current_acl_function = f->chain;
+   f->chain = free_acl_function;
+   free_acl_function = f;
+ }
```

A.6 The acl_body structure

From GCC/gcc/c-decl.c

```
+ /* ACL - this structure is used to keep track of the
+    multiple bodies that form one acl-style function. */
+ struct acl_body {
+
+   /* This points to the FUNCTION_DECL that implements this
+      body */
+   tree fndecl;
+
+   /* This is the name of the function implementing that
+      body */
+   tree name;
+
+   /* The parameters of the function implementing that body */
+   tree parmlist;
```

```
+
+   /* acl imported globals NOT used in this body.  */
+   struct acl_old_decl *globs;
+
+   /* This will chain to other acl_bodies in the same
+      function */
+   struct acl_body *next;
+ };
```

A.7 Name lookup functions

From GCC/gcc/c-decl.c

```
+ /* ACL - this goes up one binding level before calling
+    lookup_name because we see the imports list after
+    starting the function, but we want to make the lookup in
+    the context containing the function not the function
+    itself */
+ tree
+ lookup_name_in_acl_imports (tree name)
+ {
+   struct binding_level *cl;
+
+   tree t;
+
+   cl = current_binding_level;
+   if (cl->level_chain)
+     current_binding_level = cl->level_chain;
+   t = lookup_name (name);
+   current_binding_level = cl;
+   return t;
+ }

  /* Look up NAME in the current binding level and its
     superiors in the namespace of variables, functions and
     typedefs. Return a ..._DECL node of some kind
     representing its definition, or return 0 if it is
     undefined.  */

+ /* ACL - made lookup_name acl-aware */
  tree
  lookup_name (name)
  tree name;

  {
-   register tree val;
```

```
+   register tree val, t = NULL_TREE;
+   struct binding_level *chain;
+   unsigned int acl_level;

    if (current_binding_level != global_binding_level
        &&IDENTIFIER_LOCAL_VALUE (name))
      val = IDENTIFIER_LOCAL_VALUE (name);

    else
      val = IDENTIFIER_GLOBAL_VALUE (name);

+   acl_level = IDENTIFIER_ACL_LEVEL (name);
+
+   /* ACL - if there is an acl-style function in the chain
+      we need to make sure that either it imports this name
+      or the name was declared after that function */
+   if (val && current_binding_level->acl_level
+       && current_binding_level->acl_level != acl_level
+       && (TREE_CODE (val) == VAR_DECL
+           || TREE_CODE (val) == PARM_DECL)) {
+     for (chain = current_binding_level; chain
+          &&(chain->acl_level ==
+             current_binding_level->acl_level);
+          chain = chain->level_chain) {
+       if (chain->
+           function_decl && (chain->function_decl ==
+                             DECL_CONTEXT (val))) {
+         t = val;
+         break;
+       }
+     }
+   } else
+     t = val;

-   return val;
+   return t;
  }

  /* Similar to `lookup_name' but look only at current
     binding level.  */
  tree
  lookup_name_current_level (name)
  tree name;

  {
    register tree t;
```

```
  if (current_binding_level == global_binding_level)
    return IDENTIFIER_GLOBAL_VALUE (name);
  if (IDENTIFIER_LOCAL_VALUE (name) == 0)
    return 0;
  for (t = current_binding_level->names; t;
       t = TREE_CHAIN (t))
    if (DECL_NAME (t) == name)
      break;
  return t;
}
```

A.8 Code to store and restore the globals from the imports clause

From GCC/gcc/c-decl.c

```
+ /* ACL - Before Going out of an acl-style function, this
+    will restore the original types of the used globals */
+ void
+ acl_restore_globals ()
+ {
+   struct acl_old_decl *globs = current_acl_function->globs;
+   register struct acl_old_decl *t, *p;
+
+   t = globs;
+   while (t) {
+     p = t->chain;
+     TREE_TYPE (t->decl) = t->old_type;
+     IDENTIFIER_ACL_LEVEL (DECL_NAME (t->decl)) =
+       t->old_level;
+     t->chain = free_acl_old_decl;
+     free_acl_old_decl = t;
+     t = p;
+   }
+ }
+
+ /* ACL - This will traverse a list of globals and save each
+    one of them. */
+ void
+ acl_save_globals (tree globs)
+ {
+   int flag;
+
+   current_acl_function->init_globs = globs;
+   while (globs) {
+     if (TREE_CODE (TREE_VALUE (globs)) == IDENTIFIER_NODE)
```

```
+       flag = 0;
+
+     else
+       flag = 1;
+     acl_save_global (TREE_VALUE (globs), flag);
+     globs = TREE_CHAIN (globs);
+   }
+ }
+
+ /* ACL - This will save the original type of an imported
+    global, and replace it with the type declared in the
+    imports clause */
+ void
+ acl_save_global (tree parm, int flag)
+ {
+   tree decl, glob;
+   struct acl_old_decl *acl, *tmp;
+
+   /* If there was a type declared in the imports clause we
+      need to process it, otherwise the global will keep its
+      original type inside the function */
+   if (flag) {
+     decl =
+       grokdeclarator (TREE_VALUE (TREE_PURPOSE (parm)),
+                       TREE_PURPOSE (TREE_PURPOSE (parm)),
+                       NORMAL, 0);
+     decl_attributes (decl, TREE_VALUE (TREE_VALUE (parm)),
+                      TREE_PURPOSE (TREE_VALUE (parm)));
+     glob = lookup_name_in_acl_imports (DECL_NAME (decl));
+
+     /* ACL - check that the new type is compatible to and
+        more restricitve than the original type */
+     if ((TREE_READONLY (glob) && !TREE_READONLY (decl))
+         || !acl_comptypes (TREE_TYPE (glob),
+                            TREE_TYPE (decl)))
+       fatal ("ACL - the type given to `%s' in the imports"
+              " clause is not compatible with its previous type",
+              IDENTIFIER_POINTER (DECL_NAME (decl)));
+   } else {
+     decl = glob = lookup_name_in_acl_imports (parm);
+   }
+   if (!glob)
+     fatal ("ACL - `%s' used in `imports' clause while not"
+            " global for current binding level",
+            IDENTIFIER_POINTER (DECL_NAME (decl)));
+
```

```
+   /* ACL - check that we did not import the same variable
+      before */
+   for (tmp = current_acl_function->globs; tmp;
+        tmp = tmp->chain) {
+     if (glob == ACL_GLOBAL_DECL (tmp->decl)) {
+       fatal ("ACL - `%s' is imported more than once",
+              IDENTIFIER_POINTER (DECL_NAME (glob)));
+     }
+   }
+   acl = new_acl_old_decl ();
+   acl->old_type = TREE_TYPE (glob);
+   acl->decl = glob;
+   acl->old_level = IDENTIFIER_ACL_LEVEL (DECL_NAME (glob));
+   TREE_TYPE (glob) = TREE_TYPE (decl);
+   IDENTIFIER_ACL_LEVEL (DECL_NAME (glob)) =
+     current_binding_level->acl_level;
+   tmp = current_acl_function->globs;
+
+   /* Insert this to the globs list in the acl_function
+      struct, when we insert a global in this list, we
+      insert it in its alphabetical order to ensure unique
+      names for the generated functions independent of the
+      order in which the globals are given in the imports
+      clause */
+   if (!tmp
+       || strcmp (IDENTIFIER_POINTER
+                  (DECL_NAME (tmp->decl)),
+                  IDENTIFIER_POINTER (DECL_NAME
+                                      (acl->decl))) > 0) {
+     acl->chain = tmp;
+     current_acl_function->globs = acl;
+   } else {
+     while (tmp->chain
+            && strcmp (IDENTIFIER_POINTER
+                       (DECL_NAME (tmp->decl)),
+                       IDENTIFIER_POINTER (DECL_NAME
+       (acl->decl))) < 0)
+       tmp = tmp->chain;
+     acl->chain = tmp->chain;
+     tmp->chain = acl;
+   }
+ }
```

From GCC/gcc/c-typecheck.c

```
 /* Return 1 if TYPE1 and TYPE2 are compatible types for
    assignment or various other operations.  Return 2 if
```

```
     they are compatible but a warning may be needed if you
     use them together.  */
  int
  comptypes (type1, type2)
       tree type1, type2;

  {
+   return actual_comptypes (type1, type2, 0);
+ }
+
+ int
+ acl_comptypes (type1, type2)
+      tree type1, type2;
+
+ {
+   return actual_comptypes (type1, type2, 1);
+ }
+
+ int
+ actual_comptypes (type1, type2, acl_flag)
+      tree type1, type2;
+      char acl_flag;
+
+ {
    register tree t1 = type1;
    register tree t2 = type2;
    int attrval, val;

    /* Suppress errors caused by previously reported errors.
       */
    if (t1 == t2 || !t1 || !t2
        || TREE_CODE (t1) == ERROR_MARK
        || TREE_CODE (t2) == ERROR_MARK)
      return 1;

    /* Treat an enum type as the integer type of the same
       width and signedness.  */
    if (TREE_CODE (t1) == ENUMERAL_TYPE)
      t1 =
        type_for_size (TYPE_PRECISION (t1),
                       TREE_UNSIGNED (t1));
    if (TREE_CODE (t2) == ENUMERAL_TYPE)
      t2 =
        type_for_size (TYPE_PRECISION (t2),
                       TREE_UNSIGNED (t2));
    if (t1 == t2)
```

```
    return 1;

  /* Different classes of types can't be compatible.  */
  if (TREE_CODE (t1) != TREE_CODE (t2))
    return 0;

  /* Qualifiers must match.  */
+ /* ACL - this is modified to work with acl-style imports,
+    mainly it counts the TYPE_QUALS of t1 to be matching
+    those of t2 if t1's only miss a const */
  if ((TYPE_QUALS (t1) != TYPE_QUALS (t2))
+     && (!acl_flag
+         || ((TYPE_QUALS (t1) | TYPE_QUAL_CONST) !=
+             TYPE_QUALS (t2)))
      )
    return 0;

  /* Allow for two different type nodes which have
     essentially the same definition.  Note that we already
     checked for equality of the type qualifiers (just
     above).  */
  if (TYPE_MAIN_VARIANT (t1) == TYPE_MAIN_VARIANT (t2))
    return 1;

  /*
   *
   *     code removed
   *
   */

  return attrval == 2 && val == 1 ? 2 : val;
}
```

A.9 Code to start an ACL function

From GCC/gcc/c-decl.c

```
int
start_function (declspecs, declarator, prefix_attributes,
                attributes, nested)
     tree declarator, declspecs, prefix_attributes,
       attributes;
     int nested;

{
  tree decl1, old_decl;
```

```
    tree restype;
    tree tmp, *prev;
    int old_immediate_size_expand = immediate_size_expand;

+   /* ACL - store parameters to be found by
+      acl_start_function if needed */
+   last_function_decl.declarator =
+     declarator ? copy_list (declarator) : NULL_TREE;
+   last_function_decl.declspecs =
+     declspecs ? copy_list (declspecs) : NULL_TREE;
+   last_function_decl.prefix_attributes =
+     prefix_attributes ? copy_list (prefix_attributes) :
+     NULL_TREE;
+   last_function_decl.attributes =
+     attributes ? copy_list (attributes) : NULL_TREE;
+   last_function_decl.nested = nested;

    /*
     *
     *     code removed
     *
     */

    /* Make the init_value nonzero so pushdecl knows this is
       not tentative. error_mark_node is replaced below (in
       poplevel) with the BLOCK.  */
    DECL_INITIAL (decl1) = error_mark_node;

+   /* ACL - by default, it is a normal c-style function (not
+      acl) */
+   DECL_ACL_FLAG (decl1) = 0;

    /* If this definition isn't a prototype and we had a
       prototype declaration before, copy the arg type info
       from that prototype. But not if what we had before was
       a builtin function.  */
    old_decl = lookup_name_current_level (DECL_NAME (decl1));
    if (old_decl != 0 && TREE_CODE (TREE_TYPE (old_decl)) ==
        FUNCTION_TYPE && !DECL_BUILT_IN (old_decl)
        && (TYPE_MAIN_VARIANT (TREE_TYPE (TREE_TYPE (decl1)))
            ==
            TYPE_MAIN_VARIANT (TREE_TYPE
                               (TREE_TYPE (old_decl))))
        && TYPE_ARG_TYPES (TREE_TYPE (decl1)) == 0) {

+     /* ACL store the globs from the prototype into the decl */
```

```
+     DECL_ACL_GLOBS (decl1) = DECL_ACL_GLOBS (old_decl);

      TREE_TYPE (decl1) = TREE_TYPE (old_decl);
      current_function_prototype_file =
        DECL_SOURCE_FILE (old_decl);
      current_function_prototype_line =
        DECL_SOURCE_LINE (old_decl);
    }

    /*
     *
     *     code removed
     *
     */

    current_function_decl = pushdecl (decl1);

+   /* ACL */
+   acl_last_function_decl = current_function_decl;
    pushlevel (0);
    declare_parm_level (1);
    current_binding_level->subblocks_tag_transparent = 1;

+   /* ACL - here we store the FUNCTION_DECL in the
+      binding_level.  */
+   current_binding_level->function_decl =
+     current_function_decl;
    make_function_rtl (current_function_decl);

    /*
     *
     *     code removed
     *
     */

    return 1;
  }

+ /* ACL - initializes an acl-style function. When we find
+    that we are starting an acl-style function this will be
+    called from c-parse.in, and it will set the appropriate
+    flags. */
+ void
+ acl_start_function ()
+ {
+   DECL_ACL_FLAG (current_function_decl) = 1;
```

```
+   current_binding_level->acl_level++;
+   new_acl_function ();
+   current_acl_function->decl = current_function_decl;
+   current_acl_function->declarator =
+     last_function_decl.declarator;
+   current_acl_function->identifier_p =
+     last_function_decl.identifier_p;
+
+   /* note: current_acl_function->parmlist should be stored
+      after store_parm_decls() is called from c-parse.in */
+   current_acl_function->declspecs =
+     last_function_decl.declspecs;
+   current_acl_function->prefix_attributes =
+     last_function_decl.prefix_attributes;
+   current_acl_function->attributes =
+     last_function_decl.attributes;
+   current_acl_function->nested = last_function_decl.nested;
+ }
```

A.10 Code to finish an ACL function

From GCC/gcc/c-decl.c

```
  /* Finish up a function declaration and compile that
     function all the way to assembler language output.  The
     free the storage for the function definition.

     This is called after parsing the body of the function
     definition.

     NESTED is nonzero if the function being finished is
     nested in another. */

  void
  finish_function (int nested)
  {
+   actual_finish_function (nested, 1);
+ }
+
+ void
+ actual_finish_function (nested, free)
+      int nested, free;
+ {
    register tree fndecl = current_function_decl;

    /* TREE_READONLY (fndecl) = 1; This caused &foo to be of
```

```
      type ptr-to-const-function which then got a warning
      when stored in a ptr-to-function variable. */

    poplevel (1, 0, 1);
    BLOCK_SUPERCONTEXT (DECL_INITIAL (fndecl)) = fndecl;

    /* Must mark the RESULT_DECL as being in this function. */

    DECL_CONTEXT (DECL_RESULT (fndecl)) = fndecl;

    /*
     *
     *    code removed
     *
     */

+   if (DECL_ACL_FLAG (fndecl))
+     flag_argument_noalias = 2;
+   else
+     flag_argument_noalias = acl_org_flag_argument_noalias;

    /* Run the optimizers and output the assembler code for
       this function. */
    rest_of_compilation (fndecl);

+   flag_argument_noalias = acl_org_flag_argument_noalias;

    current_function_returns_null |= can_reach_end;

    /*
     *
     *    code removed
     *
     */

+   /* ACL - free the nodes only if we are not in the middle
+      of an ACL function */
+   if (free) {
      /* Free all the tree nodes making up this function.

         Switch back to allocating nodes permanently until we
         start another function. */
      if (!nested)
        permanent_allocation (1);

      if (DECL_SAVED_INSNS (fndecl) == 0 && !nested) {
```

```
        /* Stop pointing to the local nodes about to be
           freed.

           But DECL_INITIAL must remain nonzero so we know
           this was an actual function definition.

           For a nested function, this is done in
           pop_c_function_context.

           If rest_of_compilation set this to 0, leave it 0.

         */
        if (DECL_INITIAL (fndecl) != 0)
          DECL_INITIAL (fndecl) = error_mark_node;
        DECL_ARGUMENTS (fndecl) = 0;
      }
    }

    /*
     *
     *    code removed
     *
     */

    if (!nested) {
      /* Let the error reporting routines know that we're
         outside a function.  For a nested function, this
         value is used in pop_c_function_context and then
         reset via pop_function_context. */
      current_function_decl = NULL;
    }
  }

+ /* ACL - finilazies an acl-style function. At the end of a
+    multi-body acl-style function this will create the
+    calling stub and possibly warn over missing bodies. */
+ int
+ acl_finish_function (int nested, int compstmt_count)
+ {
+   struct acl_body *body = current_acl_function->first;
+
+   tree fndecl = current_acl_function->decl;
+
+   compstmt_count = acl_make_stub (compstmt_count);
+   acl_restore_globals ();
+   while (body) {
```

```
+     DECL_ACL_FLAG (body->fndecl) = 0;
+     body = body->next;
+   }
+   DECL_ACL_FLAG (current_function_decl) = 1;
+   DECL_ACL_GLOBS (current_function_decl) =
+     current_acl_function->init_globs;
+   reclaim_acl_decision_tree (current_acl_function->dtree);
+
+   /* copied form finish_function() */
+   if (DECL_SAVED_INSNS (fndecl) == 0 && !nested) {
+
+     /* Stop pointing to the local nodes about to be freed.
+        But DECL_INITIAL must remain nonzero so we know this
+        was an actual function definition. For a nested
+        function, this is done in pop_c_function_context. If
+        rest_of_compilation set this to 0, leave it 0. */
+     if (DECL_INITIAL (fndecl) != 0)
+       DECL_INITIAL (fndecl) = error_mark_node;
+     DECL_ARGUMENTS (fndecl) = 0;
+   }
+   pop_acl_function ();
+   return compstmt_count;
+ }
```

A.11 Code used to start ACL bodies

From GCC/gcc/c-decl.c

```
+ static void
+ new_acl_body ()
+ {
+   struct acl_body *p;
+
+   if (free_acl_body) {
+     p = free_acl_body;
+     free_acl_body = p->next;
+   } else {
+     p = make_acl_body ();
+   }
+
+   *p = clear_acl_body;
+   p->parmlist = copy_list (current_acl_function->parmlist);
+
+   if (current_acl_function->first)
+     current_acl_function->last->next = p;
+   else
```

```
+     current_acl_function->first = p;
+
+   current_acl_function->last = p;
+ }
+
+ /* ACL - this should be called to start the first body in
+    an acl-style function. note: this must be called after
+    store_parm_decls() that's why it is not integrated with
+    acl_start_function() */
+ int
+ acl_start_first_body (int nested)
+ {
+   tree fndecl = current_function_decl;
+
+   current_acl_function->parmlist = DECL_ARGUMENTS (fndecl);
+
+   /* interrupt the current function processing to start a
+      new function for this body */
+   poplevel (1, 0, 1);
+
+   /* return to the outer context */
+   current_function_decl = DECL_CONTEXT (fndecl);
+
+   /* remove the definition that was added to the symbol
+      table */
+   if (current_binding_level == global_binding_level)
+     IDENTIFIER_GLOBAL_VALUE (DECL_NAME (fndecl)) =
+       NULL_TREE;
+   else
+     IDENTIFIER_LOCAL_VALUE (DECL_NAME (fndecl)) = NULL_TREE;
+
+   return acl_start_body (NULL_TREE);
+ }
+
+ /* ACL - this should be called at the start of subsequent
+    bodies of an acl-style function */
+ int
+ acl_start_body (tree alias_lists)
+ {
+   int ret, i = 0;
+
+   tree fn_name, *body;
+   tree call_exp;
+
+   new_acl_body ();
+
```

```
+   /* This will construct the name of this function, and
+      build the decision tree. */
+   process_alias_lists (alias_lists);
+
+   /* if this is not the first body, otherwise
+      push_c_function_context will be called from c-parse.in
+    */
+   if (alias_lists && current_acl_function->nested)
+     push_c_function_context ();
+   call_exp = current_acl_function->declarator;
+
+   /* In case this function returns pointers, find the level
+      of indirection */
+   while (TREE_CODE (call_exp) == INDIRECT_REF) {
+     i++;
+     call_exp = TREE_OPERAND (call_exp, 0);
+   }
+   call_exp =
+     build_nt (CALL_EXPR, current_acl_function->last->name,
+               acl_make_parmlist (current_acl_function->
+                                  last->parmlist));
+
+   /* Now make the constructed function return pointers too */
+   while (i > 0) {
+     i--;
+     call_exp =
+       make_pointer_declarator (NULL_TREE, call_exp);
+   }
+
+   /* Create the function */
+   if (!start_function
+       (current_acl_function->declspecs, call_exp,
+        current_acl_function->prefix_attributes,
+        current_acl_function->attributes,
+        current_acl_function->nested))
+     return 0;
+   store_parm_decls ();
+   DECL_ACL_FLAG (current_function_decl) = 1;
+   current_acl_function->last->fndecl =
+     current_function_decl;
+   return 1;
+ }
```

A.12 Code to finish an ACL body

From GCC/gcc/c-decl.c

```
+ /* ACL - this should be called at the end of each body of
+    an acl-style function */
+ void
+ acl_finish_body ()
+ {
+   struct acl_old_decl *t1, *t2;
+
+   t2 = current_acl_function->last->globs;
+   actual_finish_function (current_acl_function->nested, 0);
+   if (current_acl_function->nested)
+     pop_c_function_context ();
+
+   /* restore the imported globals to their state before
+      entering this body i.e. this is the status just after
+      the 'imports' clause */
+   while (t2) {
+     t1 = t2;
+     t2 = t2->chain;
+     TREE_TYPE (t1->decl) = t1->old_type;
+     IDENTIFIER_ACL_LEVEL (DECL_NAME (t1->decl)) =
+       t1->old_level;
+     t1->chain = free_acl_old_decl;
+     free_acl_old_decl = t1;
+   }
+ }
```

A.13 The decision tree structure

From GCC/gcc/c-decl.c

```
+ /* ACL - this structure is used to build the decision tree
+    used in constructing the dynamic dyspatch code */
+ struct acl_decision_tree {
+
+   /* the character representing the first var compared */
+   char v1;
+
+   /* the character representing the second var compared */
+   char v2;
+
+   /* a link to the parent of this node */
+   struct acl_decision_tree *up;
+
+   /* a link to the struct when the vars are equal (aliases)
+    */
+   struct acl_decision_tree *eq;
```

```
+
+   /* a link to the struct when they are not equal */
+   struct acl_decision_tree *ne;
+
+   /* a flag that this node is not needed in the decision of
+      that body */
+   struct acl_body *ignore;
+
+   /* the body to call when we reach this point */
+   struct acl_body *body;
+   struct acl_decision_tree *chain;
+ };
```

A.14 The process_alias_lists function

From GCC/gcc/c-decl.c

```
+ /* ACL - this takes a list of lists of the aliases, and
+    constructs a canonical name for the corresponding body.
+    each list is sorted alphabeticaly (parameters first then
+    globals) and then the list are sorted alphabeticaly
+    start by an array of 52 chars all zero, these correspond
+    to the params (0->25) and globals (26->51)  each alias
+    list will be represented by a linked list in this array
+    and the linked list will be sorted, and each list will be
+    terminated by -1  after processing all the lists, the
+    array will be scanned  to generate the function name
+    (which will be canonical), this is mainly because each
+    var can appear only once in all the alias lists for a
+    body  */
+ void
+ process_alias_lists (tree al)
+ {
+   tree t1, t2;
+   char h, c, e, t;
+   char *id, *p;
+   static char lv[52] =
+     { 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
+       0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
+       0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
+       0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0};
+   int len, i;
+   struct acl_decision_tree **dtree =
+     &current_acl_function->dtree;
+   struct acl_decision_tree *d;
+   struct acl_old_decl *t3, *t4;
```

```
+   struct acl_decision_tree *dt_head, *dt_ref, **dt_cur,
+     *dt_up;
+   id = xmalloc (90);
+   p = IDENTIFIER_POINTER (DECL_NAME
+                           (current_acl_function->decl));
+   len = strlen (p);
+   id[0] = '_';
+   memcpy (id + 1, p, len);
+   p = id + len + 1;
+   *p++ = '$';
+   *p++ = 'a';
+   *p++ = 'c';
+   *p++ = 'l';
+   for (t1 = al; t1; t1 = TREE_CHAIN (t1)) {
+     t2 = TREE_VALUE (t1);
+     h = c = acl_get_id_symbol (TREE_VALUE (t2));
+     if (lv[c])
+       fatal ("ACL - `%s' used twice in alias list",
+              IDENTIFIER_POINTER (TREE_VALUE (t2)));
+     lv[c] = -1;
+     for (t2 = TREE_CHAIN (t2); t2; t2 = TREE_CHAIN (t2)) {
+       c = acl_get_id_symbol (TREE_VALUE (t2));
+       if (lv[c])
+         fatal ("ACL - `%s' used twice in alias list",
+                IDENTIFIER_POINTER (TREE_VALUE (t2)));
+       if (c < h) {
+         fatal
+           ("ACL - alais list not written in the required order");
+
+           /* use the code below instead of the error if
+              the order in the list is not important this
+              may cause problems with static dispatch
+              lv[c]=h; h=c;  */
+       } else {
+         t = h;
+         while (lv[t] != -1 && lv[lv[t]] < c)
+           t = lv[t];
+         lv[c] = lv[t];
+         lv[t] = c;
+       }
+       acl_remove_id (TREE_VALUE (t2));
+     }
+   }
+
+     /* if al==NULL this means that we are in the first
+        body, so build the main branch of the decision
```

```
+        tree, in which there is no aliases at all  */
+     if (!al) {
+     t1 = current_acl_function->parmlist;
+     dt_up = NULL;
+     while (t1) {
+       if (!POINTER_TYPE_P (TREE_TYPE (t1))) {
+         t1 = TREE_CHAIN (t1);
+         continue;   /* don't include non-pointer parms */
+       }
+       h = acl_get_id_symbol (DECL_NAME (t1));
+       t2 = TREE_CHAIN (t1);
+       while (t2) {
+         if (!POINTER_TYPE_P (TREE_TYPE (t2))) {
+           t2 = TREE_CHAIN (t2);
+           continue;  /* don't include non-pointer parms */
+         }
+         c = acl_get_id_symbol (DECL_NAME (t2));
+         d = new_acl_decision_tree ();
+         d->v1 = h + 'a';
+         d->v2 = c + 'a';
+         d->up = dt_up;
+         dt_up = d;
+         *dtree = d;
+         dtree = &d->ne;
+         t2 = TREE_CHAIN (t2);
+       }
+       t4 = current_acl_function->globs;
+       while (t4) {
+         c = acl_get_id_symbol (DECL_NAME (t4->decl));
+         d = new_acl_decision_tree ();
+         d->v1 = h + 'a';
+         d->v2 = c - 26 + 'A';
+         d->up = dt_up;
+         dt_up = d;
+         *dtree = d;
+         dtree = &d->ne;
+         t4 = t4->chain;
+       }
+       t1 = TREE_CHAIN (t1);
+     }
+     t3 = current_acl_function->globs;
+     while (t3) {
+       h = acl_get_id_symbol (DECL_NAME (t3->decl));
+       t4 = t3->chain;
+       while (t4) {
+         c = acl_get_id_symbol (DECL_NAME (t4->decl));
```

```
+         d = new_acl_decision_tree ();
+         d->v1 = h - 26 + 'A';
+         d->v2 = c - 26 + 'A';
+         d->up = dt_up;
+         dt_up = d;
+         *dtree = d;
+         dtree = &d->ne;
+         t4 = t4->chain;
+       }
+       t3 = t3->chain;
+     }
+     d = new_acl_decision_tree ();
+     *dtree = d;
+     d->up = dt_up;
+     dt_cur = &d;
+   } else {
+     dt_ref = dt_head = current_acl_function->dtree;
+     dt_up = NULL;
+     dt_cur = &dt_head;
+     for (i = 0; i < 52; i++) {
+       if (lv[i]) {
+         *p++ = '_';
+         *p++ = h = i < 26 ? i + 'a' : i - 26 + 'A';
+         while (dt_head->v1 != h) {
+           if (dt_head->ignore !=
+                current_acl_function->last) {
+             if (!*dt_cur) {
+               *dt_cur = new_acl_decision_tree ();
+               (*dt_cur)->v1 = dt_head->v1;
+               (*dt_cur)->v2 = dt_head->v2;
+               (*dt_cur)->up = dt_up;
+             }
+             dt_up = *dt_cur;
+             dt_cur = &(*dt_cur)->ne;
+           }
+           dt_head = dt_head->ne;
+         }
+         e = t = lv[i];
+         lv[i] = 0;
+         while (t != -1) {
+           *p++ = c = t < 26 ? t + 'a' : t - 26 + 'A';
+           t = lv[t];
+           lv[e] = 0;
+           e = t;
+
+             /* build the decision tree part  */
```

```
+             while (dt_head->v2 != c) {
+             if (dt_head->ignore !=
+                  current_acl_function->last) {
+               if (!*dt_cur) {
+                 *dt_cur = new_acl_decision_tree ();
+                 (*dt_cur)->v1 = dt_head->v1;
+                 (*dt_cur)->v2 = dt_head->v2;
+                 (*dt_cur)->up = dt_up;
+               }
+               dt_up = *dt_cur;
+               dt_cur = &(*dt_cur)->ne;
+             }
+             dt_head = dt_head->ne;
+           }
+           if (dt_head->ignore !=
+                current_acl_function->last) {
+             if (!*dt_cur) {
+               *dt_cur = new_acl_decision_tree ();
+               (*dt_cur)->v1 = dt_head->v1;
+               (*dt_cur)->v2 = dt_head->v2;
+               (*dt_cur)->up = dt_up;
+             }
+             dt_up = *dt_cur;
+             dt_cur = &(*dt_cur)->eq;
+           }
+           dt_head = dt_head->ne;
+           dt_ref = dt_head;
+           while (dt_ref) {
+             if (dt_ref->v1 == c || dt_ref->v2 == c)
+               dt_ref->ignore = current_acl_function->last;
+             dt_ref = dt_ref->ne;
+           }
+         }
+       }
+     }
+
+       /* build the final part of this branch  */
+       while (dt_head->ne) {
+       if (dt_head->ignore != current_acl_function->last) {
+         if (!*dt_cur) {
+           *dt_cur = new_acl_decision_tree ();
+           (*dt_cur)->v1 = dt_head->v1;
+           (*dt_cur)->v2 = dt_head->v2;
+           (*dt_cur)->up = dt_up;
+         }
+         dt_up = *dt_cur;
```

```
+         dt_cur = &(*dt_cur)->ne;
+       }
+       dt_head = dt_head->ne;
+     }
+     d = new_acl_decision_tree ();
+     *dt_cur = d;
+     d->up = dt_up;
+   }
+   *p = 0;
+   current_acl_function->last->name = get_identifier (id);
+   free (id);
+   d->body = current_acl_function->last;
+ }
```

A.15 Code to generate the dynamic dispatch

From GCC/gcc/c-decl.c

```
+ /* ACL - This will traverse the decision tree and generate
+    the code that corresponds that tree to call the matching
+    body */
+ int
+ acl_gen_decision_code (struct acl_decision_tree *dt,
+                        int compstmt_count)
+ {
+   tree exp = NULL, exp1, exp2, exp3;
+   tree fndecl;
+   tree parms, args = NULL;
+   int comp;
+
+   /* We reached a leaf in the tree, so we must generate a
+      call */
+   if (!dt->eq && !dt->ne) {
+     fndecl = dt->body->fndecl;
+     assemble_external (fndecl);
+     TREE_USED (fndecl);
+     for (parms = dt->body->parmlist; parms;
+          parms = TREE_CHAIN (parms)) {
+       if (!args)
+         args =
+           build_tree_list (NULL_TREE,
+                            IDENTIFIER_LOCAL_VALUE
+                            (DECL_NAME (parms)));
+
+       else
+         args =
```

```
+           chainon (args,
+                    build_tree_list (NULL_TREE,
+                                     IDENTIFIER_LOCAL_VALUE
+                                     (DECL_NAME (parms))));
+     }
+     exp = build_function_call (fndecl, args);
+
+     /* If the function returns void, we need to create a
+        compund statement which will first call the body
+        then return null */
+     if (TREE_TYPE (DECL_RESULT (fndecl)) == void_type_node) {
+       compstmt_count++;
+
+       /* from c-parse.in [pushlevel:] */
+       emit_line_note (input_filename, lineno);
+       pushlevel (0);
+       clear_last_expr ();
+       push_momentary ();
+       expand_start_bindings (0);
+
+       /* from c-parse.in [nonnull_exprlist: expr_no_commas]
+        */
+       exp = build_tree_list (NULL_TREE, exp);
+
+       /* from c-parse.in [expr: nonull_exprlist] */
+       exp = build_compound_expr (exp);
+
+       /* from c-parse.in [stmt: expr ';'] which corresponds
+          to the function call */
+       emit_line_note (input_filename, lineno);
+       iterator_expand (exp);
+       clear_momentary ();
+
+       /* this is the return; */
+       emit_line_note (input_filename, lineno);
+       c_expand_return (NULL_TREE);
+
+       /* from c-parse.in [compstmt] */
+       emit_line_note (input_filename, lineno);
+       expand_end_bindings (getdecls (), kept_level_p (), 0);
+       poplevel (kept_level_p (), 0, 0);
+       pop_momentary ();
+     } else {
+       emit_line_note (input_filename, lineno);
+       c_expand_return (exp);
+     }
```

```
+   } else if (dt->eq && dt->ne) {
+     exp1 = acl_get_symbol_id (dt->v1);
+     assemble_external (exp1);
+     TREE_USED (exp1);
+     if (dt->v1 < 'a') {
+       exp1 = build_unary_op (ADDR_EXPR, exp1, 0);
+     }
+     exp2 = acl_get_symbol_id (dt->v2);
+     assemble_external (exp2);
+     TREE_USED (exp2);
+     if (dt->v2 < 'a') {
+       exp2 = build_unary_op (ADDR_EXPR, exp2, 0);
+     }
+     exp = parser_build_binary_op (EQ_EXPR, exp1, exp2);
+
+     /* from c-parse.in [if_prefix:] */
+     emit_line_note (input_filename, lineno);
+     c_expand_start_cond (truthvalue_conversion (exp), 0,
+                          compstmt_count);
+     position_after_white_space ();
+     compstmt_count =
+       acl_gen_decision_code (dt->eq, compstmt_count);
+     c_expand_start_else ();
+     position_after_white_space ();
+     compstmt_count =
+       acl_gen_decision_code (dt->ne, compstmt_count);
+
+     /* from c-parse.in [stmt: simple_if ELSE] */
+     c_expand_end_cond ();
+   } else {
+     exp1 = acl_get_symbol_id (dt->v1);
+     assemble_external (exp1);
+     TREE_USED (exp1);
+     if (dt->v1 < 'a') {
+       exp1 = build_unary_op (ADDR_EXPR, exp1, 0);
+     }
+     exp2 = acl_get_symbol_id (dt->v2);
+     assemble_external (exp2);
+     TREE_USED (exp2);
+     if (dt->v2 < 'a') {
+       exp2 = build_unary_op (ADDR_EXPR, exp2, 0);
+     }
+     if (dt->ne) {
+       dt = dt->ne;
+       comp = NE_EXPR;
+     } else {
```

```
+       dt = dt->eq;
+       comp = EQ_EXPR;
+     }
+     exp = parser_build_binary_op (comp, exp1, exp2);
+
+     /* Join all tree nodes with only one child into only
+        one if statement */
+     while ((dt->eq && !dt->ne) || (!dt->eq && dt->ne)) {
+       exp1 = acl_get_symbol_id (dt->v1);
+       assemble_external (exp1);
+       TREE_USED (exp1);
+       if (dt->v1 < 'a') {
+         exp1 = build_unary_op (ADDR_EXPR, exp1, 0);
+       }
+       exp2 = acl_get_symbol_id (dt->v2);
+       assemble_external (exp2);
+       TREE_USED (exp2);
+       if (dt->v2 < 'a') {
+         exp2 = build_unary_op (ADDR_EXPR, exp2, 0);
+       }
+       if (dt->ne) {
+         dt = dt->ne;
+         comp = NE_EXPR;
+       } else {
+         dt = dt->eq;
+         comp = EQ_EXPR;
+       }
+       exp3 = parser_build_binary_op (comp, exp1, exp2);
+       exp =
+         parser_build_binary_op (TRUTH_ANDIF_EXPR,
+                                 truthvalue_conversion
+                                 (default_conversion
+                                  (exp)),
+                                 truthvalue_conversion
+                                 (default_conversion
+                                  (exp3)));
+     }
+
+     /* from c-parse.in [if_prefix:] */
+     emit_line_note (input_filename, lineno);
+     c_expand_start_cond (truthvalue_conversion (exp), 0,
+                          compstmt_count);
+     position_after_white_space ();
+     compstmt_count =
+       acl_gen_decision_code (dt, compstmt_count);
+
```

```
+     /* from c-parse.in [stmt: simple_if] */
+     c_expand_end_cond ();
+   }
+   return compstmt_count;
+ }
+
+ /* ACL - This will generate the function that does the
+    dynamic dispatch */
+ int
+ acl_make_stub (int compstmt_count)
+ {
+   int i = 0;
+
+   tree call_exp;
+
+   if (current_acl_function->nested)
+     push_c_function_context ();
+   call_exp = current_acl_function->declarator;
+   while (TREE_CODE (call_exp) == INDIRECT_REF) {
+     i++;
+     call_exp = TREE_OPERAND (call_exp, 0);
+   }
+   call_exp =
+     build_nt (CALL_EXPR, TREE_OPERAND (call_exp, 0),
+               acl_make_parmlist (copy_list
+                                  (current_acl_function->
+                                   parmlist)));
+   while (i > 0) {
+     i--;
+     call_exp =
+       make_pointer_declarator (NULL_TREE, call_exp);
+   }
+   start_function (current_acl_function->declspecs,
+                   call_exp,
+                   current_acl_function->
+                   prefix_attributes,
+                   current_acl_function->attributes,
+                   current_acl_function->nested);
+   store_parm_decls ();
+   compstmt_count++;
+
+   /* from c-parse.in [pushlevel] */
+   emit_line_note (input_filename, lineno);
+   pushlevel (0);
+   clear_last_expr ();
+   push_momentary ();
```

```
+   expand_start_bindings (0);
+   compstmt_count =
+     acl_gen_decision_code (current_acl_function->dtree,
+                            compstmt_count);
+
+   /* This is the 'Abort' code */
+   id = get_identifier ("abort");
+   fail = lookup_name (id);
+   if (!fail){
+     fail=implicitly_declare (id);
+     assemble_external (fail);
+     TREE_USED (fail) = 1;
+   }
+   call_exp=build_function_call (fail, NULL_TREE);
+   call_exp=build_tree_list(NULL_TREE,call_exp);
+   call_exp=build_compound_expr(call_exp);
+   emit_line_note (input_filename, lineno);
+   iterator_expand (call_exp);
+   clear_momentary ();
+
+   /* from c-parse.in [compstmt] */
+   emit_line_note (input_filename, lineno);
+   expand_end_bindings (getdecls (), kept_level_p (), 0);
+   poplevel (kept_level_p (), 0, 0);
+   pop_momentary ();
+   return compstmt_count;
+ }
```

A.16 Other helper functions

From GCC/gcc/c-decl.c

```
+ /* start_function wants to have the parameters as a list
+    containting both the parameters and a list of their
+    types, we construct that list here */
+ static tree
+ acl_make_parmlist (tree parms)
+ {
+   tree new_parms, p;
+
+   new_parms = build_tree_list (parms, NULL);
+   for (p = parms; p; p = TREE_CHAIN (p)) {
+     chainon (new_parms,
+              build_tree_list (NULL, TREE_TYPE (p)));
+   }
+   if (parms)
```

```
+     chainon (new_parms,
+              build_tree_list (NULL, void_type_node));
+   return new_parms;
+ }
+
+ /* when starting an ACL body, given an ID this function
+    will make that ID invisible inside that body. If the
+    ID is that of a global, it will restore that global to
+    its state before entering the ACL function. If it is a
+    parameter, it will remove it from the parameter list
+    that will be given later to start_function. */
+ void
+ acl_remove_id (tree id)
+ {
+   tree bparms = current_acl_function->last->parmlist;
+   struct acl_old_decl *fglobs = current_acl_function->globs;
+   struct acl_old_decl *bglobs =
+     current_acl_function->last->globs;
+   struct acl_old_decl *acl;
+
+   tree prev = NULL;
+   char c = acl_get_id_symbol (id);
+
+   if (c < 26) {                 /* it is a parameter */
+     while (bparms) {
+       if (DECL_NAME (bparms) == id) {
+         if (prev)
+           TREE_CHAIN (prev) = TREE_CHAIN (bparms);
+
+         else
+           current_acl_function->last->parmlist =
+             TREE_CHAIN (bparms);
+         break;
+       }
+       prev = bparms;
+       bparms = TREE_CHAIN (bparms);
+     }
+   } else {                      /* it is a global */
+     while (fglobs) {
+       if (DECL_NAME (fglobs->decl) == id) {
+         acl = new_acl_old_decl ();
+         acl->old_type = TREE_TYPE (fglobs->decl);
+         acl->decl = fglobs->decl;
+         acl->old_level =
+           IDENTIFIER_ACL_LEVEL (DECL_NAME (fglobs->decl));
+         TREE_TYPE (fglobs->decl) = fglobs->old_type;
```

```
+         IDENTIFIER_ACL_LEVEL (DECL_NAME (fglobs->decl)) =
+           fglobs->old_level;
+         if (bglobs)
+           acl->chain = bglobs;
+         current_acl_function->last->globs = acl;
+         break;
+       }
+       fglobs = fglobs->chain;
+     }
+   }
+ }
+
+ /* Given a character, return the DECL corresponding to it. */
+ tree
+ acl_get_symbol_id (char sym)
+ {
+   tree result = NULL;
+   tree parms;
+   struct acl_old_decl *globs;
+   char c;
+
+   if (sym < 'a') {
+     globs = current_acl_function->globs;
+     for (c = 'A'; c < sym && globs;
+          c++, globs = globs->chain) ;
+     if (globs)
+       result = globs->decl;
+   } else {
+     parms = current_acl_function->parmlist;
+     for (c = 'a'; c < sym && parms;
+          c++, parms = TREE_CHAIN (parms)) ;
+     if (parms)
+       result = IDENTIFIER_LOCAL_VALUE (DECL_NAME (parms));
+   }
+   return result;
+ }
+
+ /* Given an IDENTIFIER node return the character that
+    must be used for it in the function name and in the
+    decision tree nodes. */
+ char
+ acl_get_id_symbol (tree id)
+ {
+   int i = -1, ret = -1;
+
+   tree parms = current_acl_function->parmlist;
```

```
+   struct acl_old_decl *globs = current_acl_function->globs;
+
+   while (parms) {
+     i++;
+     if (DECL_NAME (parms) == id) {
+       if (!POINTER_TYPE_P (TREE_TYPE (parms)))
+         fatal
+           ("ACL - `%s' is not a pointer and cannot"
+            " be used in alias list",
+            IDENTIFIER_POINTER (id));
+       ret = i;
+       break;
+     }
+     parms = TREE_CHAIN (parms);
+   }
+   if (ret != -1)
+     return ret;                 /* a parameter */
+   i = -1;
+   while (globs) {
+     i++;
+     if (DECL_NAME (globs->decl) == id) {
+       ret = i;
+       break;
+     }
+     globs = globs->chain;
+   }
+   if (ret != -1)
+     return ret + 26;            /* a global */
+   fatal
+     ("ACL - `%s' not available to be used in alias list",
+      IDENTIFIER_POINTER (id));
+   return -1;
+ }
```

BIBLIOGRAPHY

[1] Alfred V. Aho, Ravi Sethi, and Jeffrey D. Ullman. *Compilers; Principles, Thechniques, and Tools*. Addison-Wesley Publishing Company, 1986.

[2] Andrew W. Appel. *Modern Compiler Implementation in ML*. Cambridge University Press, 1998.

[3] Paolo Bucci, Joseph E. Hollingsworth, Joan Krone, and Bruce W. Weide. Part III: Implementing components in RESOLVE. *ACM SIGSOFT Software Engineering Notes*, 19(4):40–51, Oct 1994.

[4] Stephen H. Edwards, Wayne D. Heym, Timothy J. Long, Murali Sitaraman, and Bruce W. Weide. Part ii: Specifying components in RESOLVE. *ACM SIGSOFT Software Engineering Notes*, 19(4):29–39, Oct 1994.

[5] Free Software Foundation. Using and porting the gnu compiler collection, 2001. http://gcc.gnu.org/onlinedocs/gcc-2.95.2/gcc.html

[6] Douglas E. Harms. *The Influence of Software Reuse on Programming Language Design*. PhD thesis, Dept. of Computer and Information Science, The Ohio State University, Columbus, Ohio, 1990.

[7] IBM Corporation. *AIX Version 3 for RS/6000: Optimization and Tuning Guide for Fortran, C and C++*. SC09-1705.

[8] IBM Corporation. *IBM C Set ++ for AIX/6000 User's Guide Version 2.1*. SC09-1605.

[9] International Organization for Standardization. *Programming Language – C*. ISO/IEC 9899.

[10] Gary T. Leavens and Olga Antropova. ACL — Eliminating Parameter Aliasing with Dynamic Dispatch. Technical Report 98-08a, Department of Computer Science, Iowa State University, Ames, Iowa, 50011, February 1999. Available by anonymous ftp from ftp.cs.iastate.edu, and by e-mail from almanac@cs.iastate.edu.

[11] Steven S. Muchnick. *Advanced Compiler Design and Implementation*. Morgan Kaufmann Publishers, Inc., 1997.

[12] William F. Ogden, Murali Sitaraman, Bruce W. Weide, and Stuart H. Zweben. Part I: The RESOLVE framework and discipline — a research synopsis. *ACM SIGSOFT Software Engineering Notes*, 19(4):23–28, Oct 1994.

[13] E. S. Page and L. B. Wilson. *Information Representation and Manipulation in a Computer*. Cambridge University Press, second edition, 1978.

[14] G. J. Popek, J. J. Horning, B. W. Lampson, J. G. Mitchell, and R. L. London. Notes on the design of Euclid. *ACM SIGPLAN Notices*, 12(3):11–18, March 1977.

[15] Murali Sitaraman and Bruce W. Weide. Special feature: Component-based software using RESOLVE. *ACM SIGSOFT Software Engineering Notes*, 19(4):21–22, Oct 1994.

[16] Katherine E. Stewart. Using the XL compiler options to improve application performance, 1998. http://www.rs6000.ibm.com/resource/technology/options.html