Static Detection of Dynamic Memory Errors

David Evans
evs@larch.lcs.mit.edu
MIT Laboratory for Computer Science

Abstract

Many important classes of bugs result from invalid assumptions about the results of functions and the values of parameters and global variables. Using traditional methods, these bugs cannot be detected efficiently at compile-time, since detailed cross-procedural analyses would be required to determine the relevant assumptions. In this work, we introduce annotations to make certain assumptions explicit at interface points. An efficient static checking tool that exploits these annotations can detect a broad class of errors including misuses of null pointers, uses of dead storage, memory leaks, and dangerous aliasing. This technique has been used successfully to fix memory management problems in a large program.

1 Introduction

The LCLint checking tool [4, 2] has been used effectively in both industry and academia to detect errors in programs, facilitate enhancements to legacy code, and support a programming methodology based on abstract types and explicit interfaces in C. In this work, we extend LCLint to detect a broad class of important errors including misuses of null pointers, failures to allocate or deallocate memory, uses of undefined or deallocated storage, and dangerous or unexpected aliasing. These errors are particularly difficult to detect and correct through testing, since their symptoms are often platform dependent and may be far-removed from the actual problem. Since these errors typically involve violations of non-local constraints, they cannot be detected efficiently at compile-time by traditional methods.

Consider the sample code fragment in Figure 1. The function setName assigns the formal parameter pname to the global variable gname. This code may be a correct implementation of some function, but it depends on many assumptions not apparent from the implementation:

1. before the call, gname must not be the sole reference to allocated storage. Otherwise, the assignment statement on line 4 loses the last reference to this storage and it can never be deallocated.
2. after the call, the actual parameter and the global gname are aliased. The caller must not deallocate the storage pointed to by the parameter if any code executed later depends on gname and vice versa.
3. after the call, gname may not be dereferenced if the parameter was a null pointer. Further, gname may not be dereferenced as an rvalue if the parameter did not point to defined storage.

As is, we cannot determine if a call to setName will cause the program to crash or leak memory without careful analysis of the entire program. This analysis would be infeasible for all but the most trivial programs.

To enable local reasoning, we need more information about the code. We extend the LCL interface specification language [5, 9] to provide ways of expressing assumptions about memory allocation, initialization and sharing, and introduce annotations to make it convenient to express these assumptions using qualifiers on declarations in C programs.

There have been many academic and commercial projects aimed at producing tools that detect these kinds of errors at run-time (dmalloc [10], mprof [11], and Purify [Pure, Inc.]). These tools can be effective in localizing the symptom of a bug — where a null pointer is dereferenced or where leaking memory is being allocated. In some cases, this is enough to discover the actual bug in the code. In others, however, it may only be the beginning of the search. Run-time checking also suffers from the flaw that its effectiveness depends entirely on running the right test cases to reveal the problems. This is especially problematic since these tools are expensive and intrusive enough that they are often not used when the code is run in production.

In our work, annotations are used to make assumptions about function interfaces, variables and types explicit. Constraints necessary to satisfy these assumptions are checked at compile-time. Places where the constraints are violated are anomalies in the code, which
typically indicate bugs in the program or undocumented or incorrect assumptions. Section 2 describes how checking works at a high level, and Section 5 describes the analysis in more detail. Section 3 describes the storage model and what kinds of uses of storage are irregular. Section 4 describes some of the annotations that can be added to programs to make certain assumptions explicit, and checking associated with each annotation. Section 6 illustrates the process of adding annotations and detecting errors using a small example program. Section 7 relates experience using this approach to fix memory management problems and replace garbage collection with explicit deallocation in a large program.

2 Analysis Overview

Since LCLint is run frequently and on large programs, it is essential that the checking be efficient and scale approximately linearly with the size of the program. Hence, full interprocedural analysis is too expensive to be practical. Instead, each procedure is checked independently, but using more detailed interface information then is normally available. This information may include constraints on the aliases that may be introduced by a called function, constraints on how storage for a parameter or global variable must be defined before a call and how it will be defined after a call, whether parameters and return values may be null or may share storage with other references, and other constraints on what may be modified or used by a called function and how the result of a function call relates to the values of its parameters. This information is available from annotations added to the program.

When a function body is checked, annotations on its parameters and the global variables it uses are assumed to be true when the function is entered. The function body is checked using these assumptions. At all return points, the function must satisfy the constraints implied by the annotations on its return value, parameters, and the global variables it uses.

When a function call site is encountered, LCLint checks that the arguments and global variables used by the function satisfy the assumptions made by the implementation of the called function. The result of the function and the states of parameters and global variables after the call are assumed to satisfy the constraints implied by the function declaration.

By exploiting extra interface information in checking, a wide range of errors can be detected through fairly simple procedural analyses. Dataflow values keep track of extra information for variables, as well as references derived from variables (e.g., a field in a structure pointed to by a variable) when appropriate. This information includes whether or not the reference is defined or may be null, what other storage it might alias or be aliased by, and what other references might share its storage. This information may be different on different program paths. Rules are used to combine values at confinement points. In cases where values cannot be sensibly combined an error is reported (e.g., if storage is deallocated on only one of the paths through an if statement).

Certain simplifying assumptions are used to make compile-time analysis feasible and efficient. The key assumptions are: any predicate expression may be true or false, the effects of any while or for loop are identical to those for executing the loop zero or one times, compile-time unknown array indexes (or pointer offsets) are either all the same element of the array or independent elements (depending on an LCLint flag that may be set locally).

LCLint may produce messages for correct code (e.g., a use-before-definition error in a branch that would only be taken if an earlier branch initialized the variable). The alternative would be not reporting many anomalies that are likely errors. Since spurious messages can be suppressed locally by placing stylized comments around the code that produces the message, this unsoundness has rarely been a serious problem in practice.

LCLint may also fail to produce messages for certain kinds of incorrect code in some contexts. For example, if an alias is not detected because it would be produced only after the second iteration of a loop, LCLint will fail to detect an error involving the use of released storage that is only apparent if the alias is detected. It is harder to estimate the costs of undetected errors, since there is no way of knowing how many undetected errors remain.

Since our goal is to detect as many real bugs as possible efficiently and with no programmer interaction, we are willing to accept an analysis that is neither sound nor complete. Instead of using worst-case assumptions, LCLint uses approximations that follow from likely-case assumptions. Clearly, this would be unacceptable in a compiler optimizer or a theorem prover. However, for a static checking tool it allows many more ambitious checks to be done and more errors to be detected with only the occasionally annoying spurious message.

3 Storage Model

This section describes execution-time concepts for describing the state of storage. Some of these concepts correspond to analysis properties used by LCLint. Certain uses of storage are likely to indicate program bugs, and are reported as anomalies.

LCL assumes a CLU-like object storage model. An object is a typed region of storage. Some objects use a fixed amount of storage that is allocated and deallocated automatically by the compiler. Other objects use dynamic storage that must be managed by the program.

Storage is undefined if it has not been assigned a value, and defined after it has been assigned a value. An object is completely defined if all storage that may be reached from it is defined. What storage is reachable from an object depends on the type and value of the object. For example, if p is a pointer to a structure, p is completely defined if the value of p is NULL, or if every field of the structure p points to is completely defined.

When an expression is used as the left side of an assignment expression we say it is used as an lvalue. Its location in memory is used, but not its value. Undefined storage may be used as an lvalue since only its location is needed. When storage is used in any other way, such as on the right side of an assignment, as an operand to a primitive operator (including the indirection operator, *), or as a function parameter, we say it is used as an rvalue. It is an anomaly to use undefined storage as an rvalue.

A pointer is a typed memory address. A pointer is either live or dead. A live pointer is either NULL or an address within allocated storage. A pointer that points to an object is an object pointer. A pointer that points inside an object (e.g., to the third element of an allocated block) is an offset pointer. A pointer that points to allocated storage that is not defined is an allocated pointer. The result of dereferencing a allocated pointer is undefined storage. Hence, it is an anomaly to use it as an rvalue. A dead (or “dangling”) pointer does not point to allocated storage. A pointer becomes dead if the storage it points to is deallocated (e.g., the pointer is passed to the free library function.) It is an anomaly to use a dead pointer as an rvalue.

There is a special object NULL corresponding to the NULL pointer in a C program. A pointer that may have the value NULL is a possibly-

1This is similar to the LISP storage model, except that objects are typed.
2Except sizeof, which does not need the value of its argument.
null pointer. It is an anomaly to use a possibly-null pointer where a non-null pointer is expected (e.g., certain function arguments or the indirect operator).

To allow descriptions of memory constraints, we view each object as having an associated owners set. The owners set indicates which external references may legitimately refer to an object. A reference is a variable or a location derived from a variable (e.g., a field of a structure). Different references may share the same storage. For example, if \( s \) and \( t \) are char pointers, and \( s \) is assigned to \( t \), then the references \( *s \) and \( *t \) are different ways of referring to the same storage. The owners set for the storage \( *s \) includes both \( *s \) and \( *t \). In a function implementation, an external reference is any reference that is visible in the environment of the caller (i.e., a reference to any storage that can be reached from the parameters, global variables, or return value).

The size of the owners set is less than or equal to the traditional reference count since it includes only external references and references that it is valid to dereference (constraints on memory usage may make it invalid to dereference some references, such as those that have been deallocated). It is an anomaly if the owners set for an explicitly allocated object is empty, since this means there are no valid references and the storage associated with the object cannot be released.

Failures to free storage are relevant only when memory is explicitly deallocated by the programmer, and could be avoided by using a garbage collector [1]. If LCLint is used to check programs designed for use with a garbage collector, flags can be used to adjust checking so only those errors relevant in a garbage-collected environment are reported.

4 Annotations

Annotations provide a convenient way of expressing interface assumptions. Although many of the same assumptions are expressible in LCL function specifications, annotations are easier to write and have the important advantage that they can be used to determine appropriate static checking in a straightforward way. We can use annotations in LCL specifications, or directly in the source code as syntactic comments (_COMPANY.Company_Name_). For example, \( \text{null} \) in an LCL specification or \( \text{indhoven} \) in a C source file may be used in a variable declaration to indicate the variable is a possibly-null pointer (i.e., it may have the value \( \text{null} \)).

Annotations may be used in a type declaration to constrain all instances of a type, in function parameter or return value declarations to constrain the use and value of parameters and results, and in global and static variable declarations to constrain the value and use of the variable.

Annotations are syntactically similar to C type qualifiers. More than one annotation may be used with a given declaration, although certain combinations of annotations are incompatible and will produce static errors. An annotation applies only to the outer level of a declaration (e.g., \text{null} char **pname means that the char ** referenced by \text{pname} is a possibly-null pointer, but the char * referenced by \text{pname} is unqualified.) A type definition can be used to apply annotations to non-outlet level declarations.

The idea of keeping additional state information on variables is similar to that used by the NIL compiler. The NIL compiler [8] extends type checking to also check typestates. Each type has a set of typestates defined by the programming language that can be determined by the compiler at any point in the code. An object can be in only one typestate at a given point in the code, but may change typestates during execution. A subset of all operations of a type are permitted on an object in a particular typestate and operations may be declared to change the typestate of an object. The NIL compiler detects execution sequences that violate typestate constraints at compile time. Some of the memory annotations used by LCLint could be emulated using typestates.

Annotations used by LCLint are simple since our main focus is detecting errors at interface points. ADDS [6] presents an approach for dealing with recursive data structures by constraining possible aliasing relationships within datatypes. Better checking of internal aliasing would improve LCLint checking, but since our focus here is on detecting errors at interface boundaries, the annotations we use are sufficient to detect a wide range of errors.

The remainder of this section describes some of the annotations and associated checking done by LCLint. A complete list of the annotations related to memory checking is found in Appendix B.

Null Pointers

A common cause of program failures is when a null pointer is dereferenced. LCLint detects these errors by distinguishing possibly-null pointers at interface boundaries, and checking that a possibly-null pointer is not dereferenced or used where a non-null pointer is required.

In Figure 2, the \text{null} annotation is used to indicate that a possibly-null pointer may be passed as the parameter \text{pname}. LCLint will report an error if there is a path leading to a dereference of the pointer along which there is no check to ensure the pointer is not null. Code can check that a possibly-null pointer is not null by using a simple comparison (e.g., \text{x != NULL}) or a function call. To indicate that a function returns true when its argument is null the \text{true} annotation is used on the return value; \text{false} is used to indicate that a function returns true only if the argument is not null.

Running LCLint on the version of \text{sample.c} in Figure 2 produces the message\(^1\):

\begin{verbatim}
    sample.c:6: Function returns with non-null global gname referencing null storage
    sample.c:5: Storage gname may become null
\end{verbatim}

The error is reported at the exit point. It would not be an anomaly to assign \text{gname} to \text{NULL} in the body of \text{setName}, as long as it is re-assigned to a non-null value before the function returns or another function using the global \text{gname} is called.

The error can be fixed by removing the \text{null} annotation on the parameter (which would produce messages elsewhere if \text{setName} is called with a possibly null value) or adding a \text{null} annotation to the declaration of \text{gname} (which would produce messages if \text{gname} is dereferenced without first checking it is not null). Another fix is shown in Figure 3. Here, a \text{true} annotation is called to test

---

1LCLint messages often include extra information describing the anomaly detected. In this message, the first part explains the anomaly and where it is detected (line 6). The indented part shows where the value may become null (line 5).
extern char *gname;
extern /*@truenull@*/
   isNull /*@null@*/ char *x);

void setName(/*@null@*/ char *pname)
{  
   if (!isNull (pname)) { gname = pname; }
}

Figure 3: Fixing sample.c by calling a truenull function.

whether pname is null, and the assignment is only done for non-null values.

A variable of a pointer type with no annotation is interpreted as non-null, unless the type was declared using null. In these cases, the type’s null annotation may be overridden for specific declarations of the type using the notnull annotation. This is particularly useful for parameters to hidden (static) operations of abstract types where the null test has already been done before the function is called, and for return values that are never null.

An additional annotation, relnull may be used to relax null checking. A relnull pointer is assumed to be non-null when it is used, but no error is reported if a possibly null value is assigned to it. This is generally used for structure fields that may or may not be null depending on some other constraint. It is up to the programmer to ensure that this constraint is satisfied before the pointer is dereferenced.

Definition

There is an implicit constraint that all function parameters and global variables used by a function are completely defined before a call, and that the return value is completely defined after the call. For example, LCLint will report an error if a pointer actual parameter is allocated but the storage it points to is not defined, or if a field in a structure pointed to by the return value is not defined. Function implementations are checked assuming all parameters and global variables are completely defined at entry to the function.

Occasionally, it is desirable to have parameters or return values that reference undefined or partially defined storage. For example, a pointer may be passed as an argument that is intended as an address to store a result, or a memory allocator may return allocated but undefined storage. The out qualifier can be used to denote storage that may be not be completely defined.

An actual parameter that corresponds to a formal parameter with an out annotation must be defined but need not be completely defined. That is, the actual parameter is used as an rvalue so it must be defined, but storage reachable from the actual parameter need not be defined. LCLint does not report an error when allocated storage is passed as an out parameter. After the call, storage that was passed as an out parameter is assumed to be completely defined.

Within the implementation of a function, LCLint will assume that an out formal parameter is allocated but that storage reachable from the parameter is undefined. Hence, an error is reported if storage derived from it is used as an rvalue before it is defined. An error is reported if the implementation does not define all storage reachable from an out parameter before returning.

An analogous annotation, undef, may be used on a global variable in the globals list for a function to indicate that the global variable may be undefined when the function is called.

The partial qualifier can be used to relax checking of structure fields. A structure qualified with partial may have undefined fields. LCLint reports no errors when these fields are used. Similar to relnull, the reldef qualifier is provided to relax definition checking, and is sometimes useful in field declarations.

Allocation

There are two kinds of deallocation errors with which we are concerned: deallocating storage when there are other live references to the same storage, or failing to deallocate storage before the last reference to it is lost. To handle these deallocation errors, we introduce a concept of an obligation to release storage. Every time storage is allocated, it creates an obligation to release the storage. This obligation is attached to the reference to which the storage is assigned. Before the scope of the reference is exited or it is assigned to a new value, the storage to which it points must be released. Annotations can be used to indicate that this obligation is transferred through a return value, function parameter or assignment to an external reference.

The only annotation is used to indicate that a reference is the only pointer to the object it points to. We can view the reference as having an obligation to release this storage. This obligation is satisfied by transferring it to some other reference in one of three ways:

1. pass it as an actual parameter corresponding to a formal parameter declared with an only annotation
2. assign it to an external reference declared with an only annotation
3. return it as a result declared with an only annotation

After the release obligation is transferred, the original reference is a dead pointer and the storage it points to may not be used.

All obligations to release storage stem from allocation routines (e.g., malloc), and are ultimately satisfied by calls to deallocators (e.g., free). The standard library provides some allocation and deallocation routines. The basic allocator, malloc, is specified as

null out only void *malloc (size_t size);

It returns a possibly-null pointer (it returns NULL when the requested memory cannot be allocated) that is not completely defined and is not referenced by any reference other than the function return value. The deallocator, free, is specified as

void free (null out only void *ptr);

The argument to free is a possibly-null, not necessarily completely defined, pointer to unshared storage. Since the parameter is declared using only, the caller may not use the referenced object after the call, and may not pass in a reference to a shared object. There is nothing special about malloc and free — their behavior can be described entirely in terms of the provided annotations.

Other annotations can be used to express different assumptions about memory management. The temp annotation is used on a formal parameter to indicate that the called function may not deallocate the storage the parameter refers to or create new external references to this storage. At a call site where a reference is passed as a temp parameter, the aliases to the storage it references are the same before and after the call.

4The ANSI Standard allows a null pointer to be passed to free. Many older C implementations do not support this, so it may be desirable to use an alternative specification with no null annotation.

5To check that allocated objects are completely destroyed (e.g., all unshared objects inside a structure are deallocated before the structure is deallocated), LCLint checks that any parameter passed as an out only void * does not contain references to live, unshared objects. This makes sense, since such a parameter could not be used sensibly in any way other than deallocating its storage.
One way to fix the problem would be to assign to gname pointer.

The first message reports a memory leak. Because gname is declared using the only annotation, gname is the only reference to an object and after the assignment the storage used by this object can never be reclaimed.

The second error warns of an anomaly that could lead to problems. The only reference gname now references shared storage. If the caller deallocates the actual parameter, gname will become a dead pointer.

One way to fix the problem would be to assign to gname a copy of the object pointed to by pname. Another fix would be to change the declaration of pname from temp to only. This would lead to other messages reporting places where setName is called with an actual parameter that is not an unshared reference or where the value of the actual parameter is used after the call to setName.

In real programs it is sometimes necessary to use weaker assumptions about memory use. The owned annotation denotes a reference with an obligation to release storage. Unlike only, however, other external references (marked with dependent annotations) may share this object. It is up to the programmer to ensure that the lifetime of a dependent reference is contained within the lifetime of the corresponding owned reference.

Additional annotations provided for handling reference counted storage, unfreeable shared storage, and exposure for internal references are described in [3].

Aliasing

Program errors often result when there is unexpected aliasing between parameters, return values, and global variables. Since aliasing problems sometimes lead to deallocation errors, the annotations provided for detecting allocation anomalies also detect many of the common aliasing anomalies. Two additional annotations are provided to improve alias analysis and to detect other problems involving aliases.

The returned qualifier can be used in a formal parameter declaration to indicate that the return value may alias this parameter. It may be used in conjunction with the allocation qualifiers, and is commonly used with temp to indicate that no new aliases for the parameter will be created except for the return value.

The unique qualifier is similar to only except it does not transfer the obligation to release storage and does not prevent aliasing that is invisible to the called function.

Listing 4. Gnat-issued partial list_addh example:

```c
typedef /*@null@*/ struct _list
{ /*@only@*/ char *this;
  struct _list *next;
} *list;

void list_addh(/*@temp@*/ list l, /*@only@*/ char *e)
{ /*@out@*/
  l->next = (list) smalloc(sizeof(*l->next));
  l->next->this = e;
}
```

Figure 5: Buggy list_addh implementation.

5 Analysis

The annotations and type definitions determine the initial dataflow values of variables and constrain the acceptable values for parameters, global variables, and function results at interface points. Three values are associated with each reference: the definition state (defined, partially defined, allocated, etc.), the null state (definitely null, possibly null, not null, etc.), and the “allocation” state (corresponding to the allocation annotation, e.g., only,temp). These values may change when assignments or function calls occur in the program. An anomaly is reported if values are inconsistent at an interface point.

Figure 5 shows a buggy program to add a node at the end of a linked list. There are two problems: the case where the parameter l is null is not handled correctly and the next field of the new node allocated on line 21 is never defined. Figure 6 shows the control flow graph that corresponds to list_addh. The circled numbers are used to refer to execution points.

Point 1 is the function entry point. Here, the dataflow values are set according to the annotations and type definitions. For parameter l, the type definition for list has a null annotation so its null state is possibly-null. It has no definition annotation, so it is completely-defined. Because of the temp annotation, its allocation state is temp. Similarly, the parameter e is characterized as completely-defined, not-null, and only.

Since the function parameter may be assigned to a new value in the function implementation, we need a way of distinguishing a reference that corresponds to the actual parameter from the parameter inside the function body. We introduce a local variable l to represent the parameter in the function body. In this discussion, we use l to refer to the local variable and argl to refer to the externally visible parameter. At the function entrance, l aliases argl.

At point 2, the null state of l is not null. Because of the if statement in line 14, we know at compile-time that l is non-null if point 2 is
is annotated with only. So, the allocation state of \( e \) becomes kept. This means its obligation to release storage has been satisfied, but it can still be safely used. (If it had been passed as an only parameter instead, its definition state would become dead to indicate that is may not be used.) Since \( e \) aliases \( \text{arg2} \), the allocation state of \( \text{arg2} \) is also set to kept, and the obligation to released storage implied by the only annotation on the parameter \( e \) has been satisfied on this path. After the assignment in line 23, \( l->\text{next} \rightarrow \text{this} \) is defined. As before, this definition propagates to its base storage, and \( l->\text{next} \) and \( l \) (which is already partially-defined) are marked partially-defined.

At point 10, the two branches merge. On the true branch, the allocation state of \( e \) is kept. On the false branch, it is only. This is a confluence error since there is no sensible way to combine the allocation states — one means the storage must be released, and the other means it must not be released. LC-LInt reports this as a program anomaly. To prevent further errors, the allocation state of \( e \) is set to a special error marker.

Also at point 10, we need to merge the dataflow values associated with \( l \) and \( \text{arg1} \). On the true branch from point 9, \( l \) and \( l->\text{next} \) are partially-defined, \( l->\text{next} \rightarrow \text{this} \) is defined, and \( l->\text{next} \rightarrow \text{next} \) is undefined. On the false branch, \( l \) is completely defined. Definition states are combined using the weakest assumption. Hence, at point 10, \( l \) and \( l->\text{next} \) are partially-defined, and \( l->\text{next} \rightarrow \text{next} \) is undefined. The definition states for \( \text{arg1} \) and its derived storage are handled similarly.

Point 11 is the function exit. LC-LInt checks that the function implementation satisfies the external constraints. One implicit constraint is that \( \text{arg1} \) must be completely defined when the call returns. Since the definition state of \( \text{arg1} \) is partially-defined, LC-LInt checks that all storage derivable from \( \text{arg1} \) is defined. Since \( \text{arg1} \rightarrow \text{next} \rightarrow \text{next} \) is undefined, LC-LInt produces an error reporting an incomplete definition anomaly.

### 6 Example

This section demonstrates how annotations can be added to an existing program, thereby improving its documentation and maintainability, and detecting errors in the process. For this example, we use the toy employee database program (1000 lines of source code and 300 lines of interface specifications) described in [5]. In [2], we described how LC-LInt without dynamic memory checking was used on the original database program. Here, we start with the database program after correcting the errors described there. (For information on obtaining the complete code used in this example, see Appendix A.)

We start with a program with no annotations. LC-LInt’s interpretations of declarations with no annotations are chosen to make it possible to begin finding errors in an existing program without having to spend a lot of time adding annotations or being overwhelmed by messages. The default interpretations can be controlled by flags, to better suit a particular program.

The interpretation of a declaration with no null pointer or definition annotation is chosen so that the interpretations when annotations are missing place the strictest constraints on actual parameters and return values — they may not be null, and must be completely defined. LC-LInt checking will alert the programmer to places where this is not the case. These may be errors in the code or places where a null or out annotation should be added.

An unqualified formal parameter is assumed to be temp storage. This places the weakest constraints on actual arguments, but constrains how the parameter may be used in the function implementa-
typedef struct _elem {
    ered val; struct _elem *next;
} *ercElem;

typedef struct {
    ercElem *vals; int size;
} *erc;

erc erc_create (void) {
    erc c = (erc) malloc (sizeof (*c));

    if (c == NULL) {
        error ("malloc returned null");
        exit (EXIT_FAILURE);
    }

    c->vals = NULL;
    c->size = 0;
    return c;
}

Figure 7: erc_create from erc.c

Null Pointers

One anomaly involving null pointers is reported for the function erc_create (shown in Figure 7):

erc.c:26: Null storage c->vals derivable from return value: c
erc.c:24: Storage c->vals becomes null

The vals field of c was assigned to NULL on line 24. In this case, the code is correct and the reported anomaly suggests that a null annotation is needed on the vals field in the type definition for erc:

typedef struct {
    /*@null@*/ ercElem *vals; int size;
} *erc;

Running LCLint after this change detects three new anomalies. One is in the macro definition of erc_choose for the parameter c of type erc:

erc.h:14: Arrow access from possibly null pointer c->vals: (c->vals)->val

Since we have added the null annotation to the vals field of erc, c->vals may be a null pointer. So, LCLint detects an anomaly when it is dereferenced by the arrow operator. The specification for erc_choose includes a requires clause\(^6\) constraining the size of the collection to be greater than 0. From this it follows that the value of c->vals is not null. An assertion is added to the code to check that c->vals is not null.

The other two anomalies involve similar problems in other functions. While none of these indicate a bug in the code because of the requires clauses, they do draw our attention to places where there are dependencies on external constraints and the added assertions may be helpful in debugging clients that do not satisfy the requires clauses. The checking has directed us to places where adding assertion checks would be good defensive programming practice. Further, the null annotation on the vals field of the type definition serves as useful documentation.

Allocation

Next, we look for errors involving deallocation. We are starting with a program with no allocation annotations, but using a standard library with annotated versions of malloc and free. For expository purposes, we run LCLint with a command line flag (-allimponly) that turns off the implicit only annotations on return values, global variables, and structure fields. Hence, LCLint will produce a message everywhere newly allocated storage is returned or external storage is deallocated. (It would be impractical to check a real program without using implicit annotations.) Seven anomalies are detected by LCLint, all resulting from missing only annotations.

Two messages concern the return statements in erc_create and erc_print. Both functions return a pointer that was the result of a call to malloc. Since the function result has no only annotation, the obligation to release this storage is not transferred to the caller and a memory leak is suspected. Hence, only annotations are added to the function return value declarations.

Four messages concern assignment of allocated storage to fields of a static variable (eref_pool in erref.c). These are fixed by adding only annotations to two fields of the type declaration.

The remaining message concerns the call to free in erc_final:

erc.c:49: Implicitly temp storage c passed as only param: free (c)

Since c is an external parameter with no only qualifier, an anomaly is detected when it is passed to free since it matches a formal parameter declared with an only annotation. The only annotation needs to be added to the parameter declaration for erc_final.

After the changes, LCLint detects six new anomalies. They result from the only annotations that were added to erc propagating to calling functions. They are similar to those we have already seen and can be fixed by adding only annotations to function declarations.

As before, the new annotations propagate up the call chain to produce more messages. Six memory leaks are detected in the test driver code where variables referencing allocated storage are assigned to new values before the old storage is released. After these are fixed by adding calls to free, no allocation anomalies are detected by LCLint. If we had not used the flag to disable the implicit annotations, these six errors would have been found directly. The only annotations that would be needed are the annotations on the parameters.

Aliasing

One aliasing anomaly is reported in employee_setName (shown in Figure 8):

employee.c:13: Parameter 1 (e->name) to function strcpy is declared unique but may be aliased externally by parameter 2 (s)
```c
7 bool
5
6 employee_setName (employee *e, char *s)
7 {
8      (checks size of s)
9   strcpy(e->name, s);
10  return TRUE;
11 }
```

Figure 8: employee_setName from employee.c

The specification of strcpy in the standard library is:

```c
2 char *strcpy
3 (out returned unique char *s1, char *s2);
```

The unique qualifier indicates that s1 must refer to storage that is not shared by any other parameter or accessible globally (in this case, the parameter s2). This is necessary since the behavior of strcpy is undefined if the arguments share storage space. Since the arguments to employee_setName are not qualified, it is possible that e->name and s refer to the same storage. We add a unique qualifier to the parameter declaration for s to document that the parameter must not reference any external storage reachable from this function. Since there are no global variables, this means the parameters e and s must not share any storage. Now, if a client calls employee_setName with dependent parameters, LCLint will report an anomaly.

### Summary

A total of 15 annotations were needed to resolve all detected anomalies — one null annotation on a structure field, one out annotation on a parameter (that was detected through complete definition checking), and 13 only annotations. Of the 13 only annotations, only 2 would have been necessary if we had set command-line flags to use implicit annotations. With minimal effort in adding annotations, a few errors in the code were found and the documentation was improved considerably.

### 7 Experience

Part of the motivation for this work was my own troubles dealing with memory management in the implementation of LCLint. LCLint is over 100,000 lines of source code and incorporates code from at least three different authors employing different memory management styles. The original implementation did not attempt to deallocate memory completely, and a garbage collector was used to reclaim storage. Although this was satisfactory as a research vehicle, it had practical disadvantages and limited the number of platforms to which LCLint could be easily ported. Several earlier attempts to fix LCLint’s memory management by myself and others had failed. One frustrated person who attempted to port LCLint wrote:

...its implementation with regard to memory management is horrible. Memory is allocated willy-nilly without any way to track it or recover it. Malloced pointers are passed and assigned in a labyrinth of complex internal data structures. It becomes impossible to find.

We used the annotations and associated checking described in this paper to make substantial improvements to LCLint. Garbage collection was replaced by explicit memory deallocation, producing a more portable system with improved performance. Numerous bugs relating to null pointer dereferences, incomplete definition (usually forgetting to initialize a structure field), and aliasing were detected. Memory annotations also enabled certain efficiency improvements (such as sharing storage or using NULL to represent the empty string) that were considered too risky to attempt without them. Further, the resulting system is clearly documented with checked memory annotations. This allows maintenance changes to be made easily, and their external effects to be detected quickly.

Annotations were added in an iterative process, similar to that described in Section 6. Running LCLint on the code with no annotations produced on the order of a thousand messages. Nearly all of these messages, however, were quickly eliminated by adding an annotation or making a small change to the code (usually adding a missing free to fix a storage leak). Often, adding a single annotation on a type declaration or parameter would eliminate dozens of messages.

Since LCLint was run repeatedly on the code after changing annotations, it was important that the checking was efficient. It takes less than four minutes (on a DEC 3000/500) to check the entire program. During the later phases, checking became more modular as I focused on subtle problems in a single file. By using libraries to store interface information, a representative 5000 line module is checked in under 10 seconds.

It took a few days (split over several weeks) to add all the annotations and fix the detected problems. This compares favorably to more than a week spent previously trying to fix these problems unsuccessfully using run-time methods. For the most part, adding annotations is a fairly methodical process, and I hope future work will make it possible to automate a large portion of it.

In the course of checking, the need for the relaxed checking annotations (renull, partial, and reldef) became apparent. There were situations where simple annotations were not expressive enough to describe constraints, so checking needed to be relaxed to eliminate spurious messages. This eliminates many messages without much effort, but it also means less checking is done and more errors may be undetected.

Some of the reported messages were considered spurious. There were 75 places where styled comments were used to suppress messages relating to checks described in this paper. The most common problem was where different branches of an if statement used storage inconsistently. Many of these were places where the code was attempting to recover from a failed assertion or handling an error condition (e.g., a new object denoting an error is returned from a function that does not normally return only storage), so LCLint was correct in reporting an anomaly but it was not considered a bug that needed to be fixed. The remaining spurious messages resulted from places where either LCLint’s alias analysis is not good enough to handle the code correctly, LCLint’s execution flow analysis is not good enough to determine that a particular path through the code will never be taken, or where the code violates constraints imposed by the annotations in a way that I believed to be safe because of external constraints. The dangers of suppressing messages became clear when testing revealed that one of these suppressed messages indicated a real bug.

After checking was complete, I tested the program with explicit deallocation. As expected, not all memory management bugs had been detected statically. There were a few errors involving incor-
rectly freeing storage resulting from pointer arithmetic, two errors resulting from freeing static storage, and two errors resulting from missing annotations in the standard library specification, and one error involving a complex dependency on a global variable. Then, run-time tools were used to look for remaining memory leaks. Several were detected, relating to storage reachable from global and static variables that was not deallocated. Since LC-Lint does not do interprocedural program flow analysis, it cannot detect failures to free global storage before execution terminates.\footnote{LC-Lint has since been improved to detect freeing offset pointers and static storage.}

\section{Conclusion}

In this paper, we have seen how annotations can be added to make assumptions about memory management explicit at interface points. The annotations improve program documentation, and can be used by a static checker to detect anomalies that typically indicate bugs or incorrect annotations. We were able to use this approach to fix memory allocation problems in a large program where \textit{ad hoc} and run-time checking methods had failed. Annotations and static checking made it possible to fix memory management problems in a systematic, goal-directed manner. The memory annotations were a great help in maintaining and developing code. It is easy to see the effect of a change in memory sharing by changing an annotation and running LC-Lint.

Static checking cannot detect all errors, and certainly does not eliminate the need for run-time checking and extensive testing. However, a combination of static checking using annotations and run-time checking and testing can help produce reliable code with less effort than traditional methods.

We do not yet have experience using this approach as a new program is developed. I suspect adding annotations while a new program is being developed would not pose a major overhead. Programmers should consider their assumptions about external constraints, and the annotations provide a convenient and precise way of documenting some of these assumptions.

\section*{Acknowledgements}

I thank John Guttag for help with this research and writing this paper, Thomas Reps from the program committee for constructive comments well beyond the call of duty, Raymice Stata for reviewing a draft of this paper, and Sheryl Risacher for help with the abstract.

\section*{References}


\section{A Availability}

The web home page for LC-Lint is http://larch-www.lcs.mit.edu/8001/larch/lclint/index.html

LC-Lint can be downloaded from http://larch-www.lcs.mit.edu/8001/larch/lclint/download.html or obtained via anonymous ftp from ftp://larch.lcs.mit.edu/pub/Larch/lclint/

Several UNIX platforms are supported and source code is available.

LC-Lint can also be run remotely using a form at http://larch-www.lcs.mit.edu/8001/larch/lclint/run.html

The example described in Section 6 is found at http://larch-www.lcs.mit.edu/8001/larch/lclint/samples/db/

To receive announcements of new releases, send a (human-readable) message to lclint-request@arch.lcs.mit.edu.

\section{B Memory Management Annotations}

All annotations may be used in either an LCL specification or in a C source or header file preceded by /\* @. Unless excluded explicitly, annotations can be applied to a type definition, variable declaration, parameter declaration, or function return value. At most one annotation in any category can be used on a given declaration.

\subsection*{Null Pointers}

\begin{verbatim}
null May have the value NULL.
notnull Not permitted to have the value NULL. This is implied if there is no annotation, but may be necessary for some declarations to override null in a type definition.
retnull Relax null checking. Value assumed to be non-NULL when it is used, but may be assigned to NULL.
\end{verbatim}
Definition

out  Referenced storage need not be defined. For parameters, this means passed memory must be allocated but not necessarily defined. For return values, it means the result is allocated but not necessarily defined.

in  Referenced storage is completely defined. (Normally, this is assumed if no other definition annotation is used. Flags can be used to allow the out annotation to be assumed for unannotated parameters where it would prevent a message.)

partial  Referenced storage is partially defined. No errors are reported when incompletely defined storage is transferred as a partial, and no error is reported when storage derived from a partial is used.

reldef  Relax definition checking. Value assumed to be defined when it is used, but need not be assigned to defined storage.

Allocation

only  Refers to unshared storage; confers obligation to release this storage or transfer the obligation.

keep  Like only, except that the caller may safely use the reference after the call. (Function parameters only.)

temp  Temporary storage. Function may not deallocate or add new external references to storage. (Function parameters only.)

owned  Refers to storage that may be shared by dependent references. This reference is responsible for releasing the storage.

dependent  Refers to storage that may be shared by an owned reference. This reference may not release the storage.

shared  Refers to arbitrarily shared storage; may not be deallocated. (For use with garbage collectors.)

Parameter Aliasing

unique  May not share storage with any other function parameter or accessible global. (Function parameters only.)

Returned References

returned  A reference to the parameter may be returned. (Function parameters only.)

Exposure

observer  Returned storage must not be modified (this disallows deallocation also) by caller. (Return values only.)

exposed  Mutable returned storage from internal abstract type or passed mutable storage assigned to field of abstract type. May be modified but not deallocated. (Return values and function parameters only.)