Using Specifications to Check Source Code

David Evans
8 June, 1994

©Massachusetts Institute of Technology, 1994

This report is a revised version of the author's thesis, submitted to the Department of Electrical Engineering and Computer Science on 12 May, 1994 in partial fulfillment of the requirements for the degrees of Master of Science and Bachelor of Science. The thesis was supervised by Professor John V. Guttag. The research was supported in part by ARPA (N00014-89-J-1988), NSF (9115797-CCR), and DEC ERP. The author's current address is: MIT Lab for Computer Science, 545 Technology Square, Cambridge, MA 02139. Internet: `evs@lcs.mit.edu`.

Abstract

Traditional static checkers are limited to detecting simple anomalies since they have no information regarding the intent of the code. Program verifiers are too expensive for nearly all applications. This thesis investigates the possibilities of using specifications to do lightweight static checks to detect inconsistencies between specifications and implementations. A tool, LCLint, was developed to do static checks on C source code using LCL specifications. It is similar to traditional lint, except it uses information in specifications to do more powerful checks. Some typical problems detected by LCLint include violations of abstraction barriers and modifications of caller-visible state that are inconsistent with the specification. Experience using LCLint to check a specified program and to understand and maintain a program with no specifications illustrate some applications of LCLint and suggest future directions for using specifications to check source code.

Keywords: Formal Specifications, Checking, Debugging, Software Maintenance, Programming Conventions, Larch, C, LCL, LCLint.

Acknowledgements

I would like to thank John Guttag for his insights, direction and perspective. Without him there is no doubt this work would not have been started, without his guidance and encouragement it would not have been completed, and without his thorough reading of drafts of this thesis it would not be as understandable. John and Jim Horning developed the original ideas behind LCLint, provided invaluable advice on its functionality and design, contributed the `dbase` example, and acted as its most enthusiastic users. I also thank Jim for hosting me during my visit to DEC SRC.

Yang Meng Tan contributed many good ideas for improving LCLint, and explained LCL objects to me on more than one occasion. LCLint incorporates his LCL checker, and he helped me understand its intricacies. LCLint also incorporates code from the original LCL checker written by Gary Feldman, Steve Garland, and Joe Wild. Nate Osgood provided the original C grammar for LCLint. The `quake` example was provided by Steve Harrison. If it were not for his careful coding and good programming style, this thesis would have been much longer.

I would like to thank my officemates, Yang Meng and Anna Pogosyants for providing a pleasant atmosphere, as well as many interesting diversions and technical discussions. I would like to thank the other members of the SPD group at MIT, Steve Garland, Raymie Stata, and Mark Vandevoorde, for their helpful feedback and ideas.

Finally, I thank my parents for many years of support and inspiration. My mother challenged me to finish my thesis before she finished hers, and my father taught me more about science looking through telescopes as a youngster than all of MIT's courses had to offer.

Contents

1 Introduction **11**

1.1 Design Goals 12

1.2 Background 13

1.2.1 C 13

1.2.2 Larch 13

1.2.3 LCL 13

1.2.4 Programming Conventions 16

1.3 Related Work 17

1.3.1 Program Verifiers 18

1.3.2 Unaided Static Checking 18

1.3.3 Static Checkers Employing Specifications 20

1.4 Overview of Thesis 23

2 Checks **25**

2.1 Abstract Types 25

2.2 Globals Checking 28

2.3 Modifies Checking 28

2.3.1 Unseen Modifications 31

2.3.2 Aliasing 32

2.3.3 Specification Aliasing 34

2.3.4 Missing Specifications 35

2.4 Use before Definition Checking 36

2.5 Macro Checking 38

2.6 Other Checks 39

2.7 LCLint Messages 40

3 Checking Specified Programs **41**

3.1 Code Checks 42
3.2 Specification Derived Checks 44

4 Maintaining Programs **51**
4.1 No Specifications 51
4.2 Adding Minimal Specifications 55
4.3 Developing the Specifications 65
4.4 Summary 70

5 Conclusions **73**
5.1 Design Goals 74
5.1.1 Efficiency 74
5.1.2 Flexibility 76
5.1.3 Incremental Effort and Gain 76
5.1.4 Easy to Learn and Use 77
5.2 Extensions 78
5.2.1 Improvements 78
5.2.2 Using More of the Specification 79
5.2.3 Augmenting the Specification Language 80
5.3 Summary 81

A User's Guide **83**
A.1 Type Access 83
A.2 Libraries 84
A.3 Make 84
A.4 Emacs 84
A.5 Control Flags 84
A.6 Messages 90
A.7 Availability 94

List of Figures

1-1 Example LCL specification 14

2-1 Type abstraction violations 26
2-2 Global usage errors 29
2-3 Modification errors 30
2-4 Modifications in the presence of aliasing 33
2-5 `hideSet.lcl` 35
2-6 Use before definition errors involving `out` parameters 37
2-7 Macro checking 39

3-1 Checking `dbase` without using specifications 42
3-2 Checking `dbase` using specifications 45

4-1 Checking `quake` 53
4-2 Checking `Set` is abstract 56
4-3 Original implementation of `ForeachSet` 58
4-4 Revised implementation of `ForeachSet` 59
4-5 `Set.lcl` after including prototypes 66
4-6 `Hash.lcl` after including prototypes 67
4-7 Modification errors reported using `Hash.lcl` 69
4-8 `Execute.lcl` 70
4-9 Checking globals using `Execute.lcl` 71

5-1 Statistics for running LCLint on entire programs 75
5-2 Statistics for running LCLint on single source files 75

A-1 Sample makefile 85
A-2 Mode settings 89

Chapter 1

Introduction

Programmers spend large amounts of time trying to detect and fix errors. Bugs that are detected statically can usually be fixed easily since the programmer knows the location and nature of the problem immediately. Bugs that are not detected statically need to be found by running test cases. When a bug is found through testing we know an input that produces the incorrect result, but much more effort may be required before we can localize the problem in the source code and fix it. Worse, the bug may not be revealed during testing, producing potentially disastrous consequences when the program is used in production. Programming languages with redundancy and compilers that check for anomalies can catch some bugs during compilation, but without additional information about the program many bugs cannot be detected statically.

This thesis describes LCLint, a tool that detects inconsistencies between code and specifications. Sometimes these inconsistencies reveal manifest errors in the specifications or code. In other cases, the reported problem indicates a violation of programming conventions. While it may not cause program faults, the code depends on implementation details not apparent in the specification, or violates conventions upon which other parts of the program may rely. It may lead to problems if implementations are changed, and makes the code harder to maintain and understand. By reporting these inconsistencies, LCLint helps programmers produce better programs and decreases the time spent searching for run-time bugs. LCLint cannot find all bugs or make any guarantees about the correctness of the code. It can guarantee that certain types of errors are not present and certain programming conventions are followed, but does not eliminate the need for adequate testing.

The value of LCLint depends on two assumptions about engineering and maintaining large programs: modularity is necessary to manage complexity, and clearly defined module interfaces are useful to abstract details and limit the effects of changes. Programmers who adhere to these assumptions will attempt to write programs that are modular and have clearly defined interfaces. If they do, a tool that can detect violations of the intended interfaces should be useful.

Although LCLint is intended to be a pragmatic and useful tool, the primary motivation

behind its development is to investigate the possibilities of using specifications to do lightweight static checks on source code. By developing LCLint, we hope to learn if using specifications to check source code can be a practical and effective way of improving software quality. We also hope to gain an understanding of how the desire for static code checkers may influence the design of formal specification languages and the adoption of programming conventions. By using LCLint in a variety of ways, we hope to learn if and how such a tool can enable better software engineering, reduce the effort required to develop good programs, and help us understand and maintain existing programs.

1.1 Design Goals

In order for LCLint to be a useful tool for developing and maintaining real programs, it must detect relevant inconsistencies between specifications and source code. In addition, certain attributes were considered essential:

- efficiency — Since the intent is that LCLint be run every time the specification or source code is changed, the time needed to run LCLint should be no more than the time for compilation. This limits the checking to simple checks that do not require global analysis.
- flexibility — LCLint is not intended to impose a specific style of coding, other than one employing abstract types and distinct modules. Hence, it is important that its checking can be customized to a particular style of programming. Users of traditional lint often complain that the number of spurious messages overwhelms the number they consider important. This often leads to significant messages being dismissed, or programmers giving up on the tool entirely. We were wary of this potential flaw in designing LCLint, and tried to include means for user control so that only the desired messages appear.

  This flexibility also enables the use of LCLint to impose a particular coding style. Command line options to LCLint can be prescribed by a project manager, to require that all programmers adopt common conventions such as using C primitive types strictly. Then LCLint can be used to check the code conforms to these conventions.
- incremental effort and gain — Programmers should not have to put much effort into writing specifications to get significant benefits from using LCLint. Benefits should increase as further effort is put into the specifications.
- easy to learn and use — Since LCLint is intended to be an entry into writing formal specifications for programmers who would not otherwise write them, the knowledge of formal specifications (and Larch specifically) needed to start realizing the benefits of LCLint should be minimal. LCLint should be as easy to run as a compiler, and its output should be easy to understand.

1.2 Background

LCLint checks ANSI C source code using LCL specifications. This section briefly describes these foundations. No further knowledge should be necessary to understand this thesis, although some C programming experience is helpful.

1.2.1 C

C is a general-purpose, block-structured, low-level programming language [KR88, p. 1]. Several factors contributed to the choice of C as the target language for LCLint's source code checks. C does not provide any mechanisms for type abstraction — the `typedef` mechanism for defining new types merely introduces a synonym for a concrete type. Hence, C provides more opportunity for added value checking than languages that provide abstract types. This also means C allows added flexibility — we can implement routines having access to more than one abstract type, or routines in the module implementing an abstract type that are not allowed to use the representation.

C is widely used and there are large bodies of existing code which need to be maintained. Most C programmers are aware of many language pitfalls [Koe89]. While experienced programmers in any language make mistakes, C's economical syntax and limited type checking make C programmers particularly prone to simple programming errors that are not detected by the compiler. This makes static checking tools for C especially useful.

1.2.2 Larch

The Larch family of languages is a two-tiered approach to formal specification [GH93]. A specification is built using two languages — the *Larch Shared Language* (LSL), which is independent of the implementation language, and a *Larch Interface Language* designed for the specific implementation language. An LSL specification defines *sorts*, analogous to abstract types in a programming language, and *operators*, analogous to procedures. It expresses the underlying semantics of an abstraction.

The interface language specifies an interface to an abstraction in a particular programming language. It captures the details of the interface needed by a client using the abstraction and places constraints on both correct implementations and uses of the module. The semantics of the interface are described using primitives and sorts and operators defined in LSL specifications. Interface languages have been designed for several programming languages, including C[GH93, p. 15].

1.2.3 LCL

LCL [GH93, Tan94] is a Larch interface language for C. LCL uses a C-like syntax. Traditionally, a C module *M* consists of a source file, *M*.c, and a header file, *M*.h.

```
mutable type intSet;
uses Set(int, intSet);

int nsets;

bool intSet_member (intSet s, int e) {
   ensures result = e ∈ s^;
}

bool intSet_insert (intSet s, int e) {
   modifies s;
   ensures result = e ∈ s^ ∧ s' = insert (e, s^);
}

intSet intSet_create () int nsets; {
   modifies nsets;
   ensures fresh(result) ∧ result' = {} ∧ nsets' = nsets^ + 1;
}

bool intSet_choose (intSet s, out int *choice) {
   modifies *choice;
   ensures if (result) then (*choice)' ∈ s^
           else size(s^) = 0;
}

int intSet_size (intSet s) {
   ensures result = size(s^);
}

void intSet_initMod () int nsets; {
   modifies nsets;
   ensures nsets' = 0;
}
```

Figure 1-1: Example LCL specification

The header file contains prototype declarations for functions, variables and constants exported by *M*, as well as those macro definitions that implement exported functions or constants, and definitions of exported types. In common programming practice, clients of *M* `#include` *M*.h and refer to it for documentation. consult *M*.c.

When using LCL, a module includes two additional files — *M*.lcl, a formal specification of *M*, and *M*.lh, which is derived by LCLint from *M*.lcl. Clients use *M*.lcl for documentation, and should not need to look at any implementation file. The derived file, *M*.lh, contains include directives (if *M* depends on other specified modules), prototypes of functions and declarations of variables as specified in *M*.lcl. The file *M*.h should now include *M*.lh and retain the implementation aspects of the old *M*.h — but is no longer used for client documentation.

LCL supports *exposed types* (equivalent to C types) and *abstract types* (not supported by C). Abstract types may be immutable or mutable. An instance of an immutable

type cannot change value during execution. An integer, for example, is an immutable type. An integer variable may be assigned different values during an execution, but the value of a particular integer never changes. A mutable type is viewed as an *object*, whose value is determined by the computation state. The value of a mutable type may be changed in the course of the execution, but it remains the same object. This corresponds loosely to the C notion of a pointer to a storage location. LCL provides notation for getting the value of an object (e.g., x) in the state before (x ^) or after (x') a function is invoked. Global variables are declared using C syntax and are also viewed as objects since their value depends on the computation state.

Figure 1-1 is an example LCL specification for an abstract type to represent mutable sets of integers. The first line declares `intSet` to be a mutable abstract type. By declaring it as an abstract type, the specification leaves it up to the implementation to decide on a representation and hides implementation details from clients of the module. The next line incorporates the `Set` trait, using `int` as the name of an element in the set, and `intSet` as the name of the set. `Set` is a trait in the LSL handbook [GH93, p. 166-167] for describing a mathematical set abstraction. It provides operators such as $\in$ and `insert` on the underlying specification of `intSet` to provide semantics for the specification.

The next line declares a global variable of type `int`. The global variable `nsets` represents the number of live `intSets`. This may be useful for analyzing program performance or detecting storage leaks.

The remainder of the specification consists of function specifications. LCL function specifications are similar to C function definitions, consisting of a header and a body. The header is identical to an ANSI C function prototype, except global variables may be listed after the parameter list and function parameters may be preceded by the `out` type qualifier. The global list limits the globals which may be used in the implementation of a function. An `out` parameter constrains the use of a parameter in the function body. The pre-state value pointed to by an `out` parameter should not be used by the function. Since C does not support multiple return values, typically functions return additional values by storing them in locations passed as pointer parameters. LCL specifications make this convention explicit by declaring the parameters with `out`.

The body of the specification contains clauses constraining both the implementation and use of the function. All clauses are optional, and the semantics for missing clauses is defined. A *requires* clause gives the pre-conditions — it places constraints on the parameters and state when the function is called. An omitted requires clause means there are no constraints on the caller, other than the implied constraint that all parameters that are not specified `out` must be defined before the call. A *modifies* clause lists those parts of the visible state that the function may change. This includes global variables, parameters that are mutable abstract types, or values pointed to by reference parameters. Items in the modifies clause may be specific fields of structures or elements of arrays. A missing modifies clause means nothing visible may be changed. An *ensures* clause gives the post-conditions on valid calls of the function. If the requires clause is satisfied, the return value and post-state of the function must

satisfy its ensures clause. A missing ensures clause means the result and behavior is unconstrained, except for not modifying anything not given in the modifies clause.

The specification for `intSet_member` denotes a function that takes `intSet` and `int` parameters and returns a `bool`. No globals are listed, so no global variable may be used in its implementation. There is no requires clause, so there is no obligation on the caller other than the implicit obligation that the actual arguments be defined before the call. There is no modifies clause, so the function must not modify any visible state. The ensures clause constrains the value returned by the function to be equal to `e ∈ s ^`. So, its result is true if and only if `e` is an element of the pre-state value of `s`.

The specification for `intSet_insert` is similar, except that it also includes a modifies clause, indicating that `s` may be modified by the function. The first conjunct of the ensures clause is identical to the ensures clause for `intSet_member`. The second conjunct constrains the value of `s` when the function returns — the post-state value of `s` is the result of inserting `e` into its pre-state value.

The next function specification, `intSet_create`, illustrates the use of global variables. We wish to maintain `nsets` as a count of the number of live sets. When a new `intSet` is created, `nsets` should increase by one. Hence, `intSet_create` lists `nsets` in its global list and modifies clause, and constrains the value of `nset` after the call to be one more than its value before the call.

The specification for `intSet_choose` illustrates the use of `out` parameters to return values. If the result is `TRUE`, an element of `s` is returned through the out parameter `choice`. If the set it empty, `FALSE` is returned. The implementation of `intSet_choose` may not assume the value pointed to by `choice` is defined when the function is called.

The final operation, `intSet_initMod` initializes the module by setting the post-state value of `nsets` to `0`. According to LCL conventions, if a module provides an `initMod` operation, a client of the module should call it to initialize module state before using any of its other operations.

All the semantic content of a function specification can be given using requires, modifies and ensures clauses. Two additional clauses are provided for clarity and redundancy. A *checks* clause can be used to describe an obligation on the implementation to test certain conditions and report an error if they are not met. Although the checks clause provides convenient notation, anything expressed by a checks clause could be stated explicitly in the ensures clause. A *claims* clause provides an assertion that must follow from the specification. It adds redundancy and clarity to the specification.

The only parts of the function specification used by the current version of LCLint are the header and the modifies clause. Section 5.2.2 discusses possibilities for improving checking by using more information in the specification.

1.2.4 Programming Conventions

LCLint's effectiveness depends on certain programming conventions. While it may be

run on any C program, it cannot do better checking than a traditional lint unless the program conforms to stylistic guidelines.

LCLint adds type encapsulation to C, but this is only useful if programs are written in a modular style employing data abstractions. Although C does not provide type encapsulation mechanisms, many C programmers adopt a style which emulates abstract types. A well-designed program can usually be broken down into manageable modules, each implementing an abstract type. This makes development easier, and produces a program that can be more easily understood and maintained. The details of a type's representation are hidden where it is used, meaning clients need only understand the specification to use the type. Implementors are now free to change the implementation of the abstract type without fear that new problems will be introduced in clients that use the type. Maintainers can understand a system built using abstract types in small discrete pieces, and fix problems in one abstraction without worrying about introducing problems elsewhere.

Since C lacks mechanisms for type abstraction, programmers must rely on conventions. Typically, an abstract type will be implemented using one source file and one header file. The header file exports the type definition and its operations. Clients using an abstract type access the type through provided operations, but should not manipulate the concrete representation of the type directly. LCLint is flexible in allowing modules to be split across files, although this is usually evidence of a poor design. Ways to control whether code is an implementation or client of an abstract type are described in Appendix A.1.

In standard C all assignments exhibit copy semantics. Variables may be pointers to storage locations and share values using pointer indirection. There is no notion of an object whose value may be mutated. LCL introduces mutable abstract types that denote objects whose value may change during an execution. In order for clients to use mutable types, we need to adopt a convention for assignment semantics. We adopt the convention that assignments of mutable types must have sharing semantics. That is, if s and t are mutable types, after the assignment $s = t$, s and t refer to the same object. Any modification of s will also modify t, and vice versa. Clients may now safely use assignment with abstract types, knowing that sharing semantics are used. It is up to the implementation of an abstract type to ensure that assignments involving the type will have sharing semantics. This is most commonly done by using pointer indirection or another mutable type in the representation, however it may also be done by using handles for indexing into a local array or external files. LCLint gives a warning if a mutable abstract type is not represented using a mutable type or a pointer indirection, however it cannot confirm that sharing semantics are preserved if a handle representation is used.

1.3 Related Work

The primary goal of LCLint is to help programmers detect and eliminate bugs. Many other approaches to this goal exist, including redundancy in programming language

design, software engineering methodologies, run-time assertions and debugging environments. While these are all useful in developing high quality software, this thesis focuses only on approaches for detecting bugs statically. Most of these have been one of two extremes — program verifiers which use a complete description of the intended behavior to prove that the implementation is correct, and unaided static checkers which have no information about the program aside from the source code itself. Less common are attempts to use partial specifications (often as embedded comments) to do static checking. This middle category encompasses LCLint.

1.3.1 Program Verifiers

Much work has been done in program verification. Program verifiers help in constructing a formal proof to show that an implementation satisfies the constraints given in its specification. Such a proof can be convincing evidence that the program is correct, but it also relies on our confidence in the correctness of the specification and the proof. Unfortunately, the cost of program verification is prohibitive for nearly all projects. It is necessary to write (usually complete) formal specifications before attempting the verification. Automated tools are available to aid in proof construction, but it still requires much effort, ingenuity and knowledge from the programmer to write the complete specification and direct the proof.

1.3.2 Unaided Static Checking

Several tools have been developed for statically checking source code without the assistance of any specifications. Most of these are based on type checking. Static type checking has been a popular means for detecting bugs since Algol-60. Many languages provide type checking, including C, in which types are equivalent if they have the same concrete structure. Programmers may define new names for a type, but they are merely aliases for the original type.

Abstract Types

Many modern languages, including CLU [LAB+81, LG86] and Ada [Ada83], provide mechanisms for defining new types that are distinct from their underlying types. Abstract types hide their representation and implementation details from clients. This increases the number of bugs that may be detected during compilation, and encourages a modular programming style. LCLint adds this functionality to C.

Other methods have been used to provide abstract types in languages that do not provide them. The Fortran Abstract Data (FAD) system [MMS88] supports data abstraction in Fortran by extending the syntax of Fortran and by providing a preprocessor to convert FAD declarations into standard Fortran. The preprocessor prohibits programs from directly manipulating any variable that is declared as an abstract type. The interface to an abstract type can be specified either formally or informally

and implemented using inline substitutions and standard Fortran. The main difference between this approach to adding abstract types and LCLint's approach, is that to use FAD we need to use not only a different programming style, but an extended programming language. Programs written using FAD abstractions cannot be compiled by a standard Fortran compiler or readily understood by a Fortran programmer with no knowledge of FAD. LCL specifications used by LCLint are orthogonal to the code. Although the style of programming may change, the source code is still standard ANSI C.

C++ is a programming language based on C. It adds support for abstract types and data encapsulation within an object-oriented paradigm [Str86]. For programmers who want to use an object-oriented style, using C with LCLint is not a viable alternative to C++. For C programmers who wish to use a style employing abstract types, LCLint provides data encapsulation and type safety without the additional overhead and complexity of C++. Further, LCLint does checks not related to data abstraction that could be useful in both C and C++.

Type States

The NIL compiler [SH83] 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 its typestate may change during execution. A subset of all operations of a type are permitted on an object in a particular typestate. Some operations are declared to change the typestate of an object. For example, a data structure may have typestates *new*, *allocated* and *initialized*. A *new* object may not be read or written, but an `allocate` operation may be applied to it to create an *allocated* object. The *allocated* object may be assigned a value but not read. Interface definitions include declarations of the typestates of the call parameters before and after the call. The NIL compiler determines the typestate of objects using simple rules, and detects execution sequences that violate typestate constraints at compile time.

A similar concept has been applied to ML. Standard ML includes no type declarations, but uses type inference to determine the possible types of an expression. This provides the ability to define polymorphic functions and saves the programmer from having to write type declarations, but gives up the documentation and bug-detection advantages of explicit type declarations. Refinement types attempt to enhance bug detection within the ML type system [FP91]. A type may be refined into several subtypes, akin to typestates. We could refine a *stack* type into two subtypes: *empty* and *non-empty*. Then the `create` operation would have the type signature `create` : $\rightarrow$ *empty*, and a `top` operation that returns the top element of a non-empty stack would have the signature `top` : *non-empty* $\rightarrow$ *t*. Unlike typestates in NIL, objects in ML may have multiple refinement types. Programmers define the refinement types for basic constructors, and they are inferred by the compiler elsewhere.

The current version of LCLint does not have a notion of type states. It is possible to extend LCL to allow specifications of states of an abstract types and specify state

transitions on parameters. This is probably not worth the effort. Most types have only uninitialized and initialized states. Errors involving use of uninitialized variables are detected by simple analysis. More complex conditions can be specified using the requires or checks clause.

Lint

The lack of type checking for function calls in early versions of C prompted the development of lint[Joh78] and its extensive use. Unlike the approaches mentioned above, lint, like LCLint, is meant to be orthogonal to a compiler.

Some of the errors detected by lint result from stricter type checking than C compilers. In addition, lint detects a number of other problems including unreachable statements, variables declared but unused, functions that return on some execution paths but not on others, and inconsistent function argument types. Today, following the standardization of C and improvements in compilers, many C compilers incorporate most of the traditional lint checks. There are still benefits from additional lint checking, especially in writing portable code.

Several academic and commercial systems have been developed to extend or improve lint checking. Check [Spu90], a static checker for ANSI C, provides many useful source checks not performed by standard lint or LCLint. Its most notable similarity to LCLint in contrast to traditional lint is its macro checking (see Section 2.5).

1.3.3 Static Checkers Employing Specifications

Falling between full program verifiers and unaided static checkers are tools that use formal specifications to some degree but fall short of complete verification of the correctness of a program. These tools attempt to maintain the simplicity of use and efficiency of most simple static checkers, while gaining stronger checking using specifications. These systems relate most closely to LCLint.

Sequencing Constraints

Several checkers have been developed to analyze data usage using some form of formal specifications. Many of these involved constraining the order in which operations may be performed — for instance, a variable must be initialized before it is used. Fosdick and Osterweil [FO76] developed DAVE, a system for detecting data flow anomalies in Fortran programs using regular expressions to describe acceptable sequences of actions on data. Typical errors detected by DAVE include using a variable before it is defined. Wilson and Osterweil [WO85] extended these techniques in a similar tool for C. LCLint reports errors when a local variable or `out` parameter is used before it is defined.

More recent extensions to this research led to systems where programmers could write specifications to describe specific sequencing constraints. Cesar [OO89, OO92],

allowed programmers to specify sequencing constraints for an abstract type using a specification language based on regular expressions. For example, a programmer could specify a file type that may be opened, written to multiple times, and closed in that order. Cesar detects violations of the specified constraints — for instance, if the file is written to before it is opened. It is unclear what fraction of bugs are manifest as sequencing constraints — Osterweil and Olender optimistically claim up to 40%, but not enough experience has been attained to verify this. Their claim assumes that programs would be constrained to always check that a pointer is not null before it is referenced unless it is immediately preceded by an initialization, which may be unacceptable to most programmers. The prototype Cesar system was too inefficient to be a useful tool in real software development — it was considerably slower than a compiler. Some of this may be improved with better implementation, but the global analyses required may render these types of checking impractical in the foreseeable future.

Comments Analysis

Another approach, taken by Howden [How90], involves annotating programs with special comments. A specialization of comments analysis, is *flavor analysis*. Objects can be described by *flavors*, similar to typestates, which may change during the execution of a program. Programmers add special comments in the code to assert the flavor of an object at that point of the execution or to denote assumptions about the flavors of an object. The assertion comments are used to construct a finite state model of object flavors along program execution paths. This is then checked for consistency against the assumptions. Although the comments are embedded in the code, they are only analyzed for consistency against other comments — no analysis involving the code is done. A class of decomposition errors involving incorrect assumptions about the state of variables and data structures can be detected, but no attempt is made to verify that the assertions and assumptions are consistent with the code.

Anna

Anna [Luc90] is a specification language for Ada. Annotations are added to Ada programs. These annotations may constrain valid values of a type, describe the behavior of functions, and specify interfaces to packages, the Ada notion of abstract types. Like LCLint, it provides the freedom to specify and annotate as much or as little as the programmer desires. The Anna Transformer [San89] transforms the specifications into run-time assertions that perform consistency checks when the code is executed. Run-time assertions can detect many bugs that are not statically detectable, however their effectiveness depends on the programmer choosing appropriate test cases.

Another tool developed for Anna, is the Anna Package Specification Analyzer [Man93]. This is intended primarily for determining the correctness of a specification. Here, instead of using formal specifications to test a given implementation, the specifications are used to symbolically model the constraints of any implementation. Given

complete enough specifications, the execution of an implementation can be simulated using the specification.

The Stanford Ada Style Checker [WSS91] used Anna tools to develop a system for specifying a style of Ada coding and checking that a program conforms to it. A project manager specifies style guidelines in a *style specification language*, which are used to generate a style checker. The style checker is then run on an Ada program and style violations are reported. In some ways, this corresponds to running LCLint with no specifications using prescribed flag settings.

Inscape

Inscape explores the constructive use of specifications [Per89]. Rather than serving solely for documentation and formal verification, specifications are treated as integral to the development process. The specification language, Instress, can specify pre-conditions and post-conditions of a function, as well as obligations on the caller that must hold at some point following the call (such as closing a returned file). Inscape propagates these specifications through the implementation using a special propagation logic incorporating unknown and possible values. Bugs are detected when a pre-condition or an obligation is contradicted.

LCL has no means for expressing obligations on the caller after the call has been made. Some useful checking could be done if specifications could require that the caller at some point frees a returned object, or that the caller not modify the returned object. It is an open question if and how LCL can be extended to express these constraints, and whether they can be used by LCLint effectively.

Aspect

Aspect [Jac92] is an approach for efficiently detecting bugs in CLU programs based on unsatisfied dependencies. The specification language describes dependencies between "aspects" of objects (such as an array's size) in the post-state and pre-state, and the checker reports when a specified dependency is not present in the implementation. Dependency information in LCL specifications is often not available, or is hidden deep within the ensures clause. Moreover, LCL has no notion of aspects of an abstract type so it cannot do some of the sophisticated checking done by Aspect. Compared to Aspect, LCLint's checks are all somewhat superficial — they are unlikely to find bugs such as using the wrong variable of the correct type or omitting statements that can be found by Aspect.

Aspect decided soundness was crucial to its effectiveness — every error reported is guaranteed to be an error in the code or the specification. LCLint relaxes this restriction — some checks may be unsound, but all unsound checks can be turned off by the user.

Another difference between LCLint and most of the checkers mentioned here including Aspect, Anna and comments analysis, is that LCL specifications are separated from

the source code. The differences between specifications that are separate from the code, and those that are integrated, are more than cosmetic. Each approach has advantages and disadvantages.

Specifications which are integrated into the code can deal with lower level details of the implementation such as constraining local variables. Since they are readily apparent when the code is edited, implementors are more likely to adapt the specifications as they change the code. They are less likely to focus on the client view or provide useful client-level documentation. Even if they function only as client-level specifications, they are likely to be biased towards a particular implementation.

On the other hand, specifications separate from the code cannot refer to implementation details. They describe the function from the client's point of view, and only constrain the pre-state and post-state of the function. Implementations are unconstrained as long as they return with the state satisfying the post-condition. Stand alone specifications are useful for documentation and formal reasoning. They make the boundary between specification and implementation clear.

1.4 Overview of Thesis

The remainder of this thesis demonstrates the use of LCLint and discusses the more general issues involved in using specifications to detect bugs.

Chapter 2 describes some of the specific checks done by LCLint. Chapters 3 and 4 describe experiences using LCLint to check real programs. In Chapter 3, LCLint is used to check a fully specified program taken from the example in [GH93, Chapter 5]. In Chapter 4, LCLint is used to understand and maintain a program with no specifications in incremental steps involving adding specifications and fixing source code. Chapter 5 draws conclusions on the effectiveness of LCLint and speculates on possibilities for using specifications to check source code.

An appendix contains excerpts from the user's guide. This includes information on methods for using LCLint with auxiliary tools, complete descriptions of options, and explanations of LCLint messages.

Chapter 2

Checks

This chapter highlights key checks performed by LCLint, and illustrates them with contrived examples. A complete description of checks done is given in Appendix A. Real examples of bug detection are given Chapters 3 and 4.

LCLint was built on the LCL checker [Tan94]. It includes all the checks on specifications done by the LCL checker. In this thesis, however, we are only concerned with those checks integrating specifications and source code.

Checks done by LCLint are designed to maximize the number of real bugs reported, while minimizing the number of spurious messages. Most checks are sound and complete — it is possible to determine and report exactly those cases where a particular problem is present. Some checks involving modifies checking and use before definition are unsound and incomplete. There are cases where it is impossible or computationally intractable to determine if a suspected problem is present so a message may be issued for a problem that is not present. In other cases, a real problem may go undetected. LCLint resolves these decisions in favor of pragmatic compromises. Given that the goal is to find bugs in real programs, it is acceptable to attempt some checks that are not complete — while no guarantees can be made that all instances of a particular class of bug will be found, finding some instances is still beneficial. Likewise, it may be acceptable in rare circumstances for checks to be unsound — as long as the number of spurious messages produced is small compared to the number and importance of the real bugs that are found. The hope is that the gain incurred by finding some additional bugs far outweighs the annoyance of occasional spurious messages. Command line options and control comments allow users to suppress inapplicable messages.

2.1 Abstract Types

Section 1.2.4 described programming conventions for emulating data abstraction in C. Without automated checking, programmers must rely on careful coding and visual inspection to support these conventions. LCLint provides a means for checking these

Specification: (bigger.lcl)

```
imports intSet;

bool bigger1 (intSet s1, intSet s2) {
  ensures result = size(s1^) > size(s2^);
}

/* same for bigger2 and bigger3 */
```

Implementation: (bigger.c)

```
 1  # include "intSet.h"
 2
 3  bool bigger1 (intSet s1, intSet s2)
 4  {
 5    return (s1->size > s2->size);
 6  }
 7
 8  bool bigger2 (intSet s1, intSet s2)
 9  {
10    return (s1 > s2);
11  }
12
13  bool bigger3 (intSet s1, intSet s2)
14  {
15    return (intSet_size(s1) > intSet_size(s2));
16  }
```

LCLint execution:

```
bigger.c:5,11: Arrow access field of abstract type (intSet): s1->size
bigger.c:5,22: Arrow access field of abstract type (intSet): s2->size
bigger.c:10,11: Operands of > are abstract type (intSet): s1 > s2
```

Figure 2-1: Type abstraction violations

conventions.

Programmers may specify types as abstract using LCL. Abstract types are type-checked differently from their concrete representations. In the implementation of an abstract type, the abstract type and its representation are interchangeable. In a client of an abstract type, the abstract type is checked by name. A client should not depend on the concrete representation of the type, only on its provided operations.

Figure 2-1 shows three attempts to write a function to check if one intSet (specified as an mutable abstract type in Figure 1-1) has more elements than another. Standard lint reports no errors. LCLint reports three errors all involving violations of abstraction barriers. Each reveals an instance where the client depends on the concrete implementation of an abstract type.

The first two messages concern `bigger1`, where the `->` operator accesses a field in the structure pointed to by its left operand. The expression `s1->size` produces the `size` field of the structure pointed to by the `intSet` variable `s1`. Given the current implementation of `intSet` as a pointer to a structure containing an `int` field named `size`, this code compiles without error. It may even get the correct result, if `size` represents the number of elements in the `intSet`. However, it depends unacceptably on the representation of an abstract type. Suppose `intSet` is reimplemented using a type that is not a pointer to a structure containing a `size` field. The client code would have to be rewritten. At least in this case, when the client is compiled errors would be detected. It would be worse, however, if `intSet` were reimplemented using the same type, but changing the meaning of the `size` field. In the new implementation, `size` could be the number of elements in the array representing the set as before, but instead of checking for duplicates we insert all elements into the set. Then, `size` is not the number of set elements, but the number of insert operations on the set. The client code would compile without error, but sporadically return incorrect results.

The final messages concern a similar violation of type abstraction in `bigger2`. Two `intSets` are compared directly using the built-in `>` operator. Standard C allows comparison operations on any type except structures and unions, so no error is detected by a C compiler. The result of a comparison involving abstract types depends on the representation of the types. If `intSet` is implemented using a pointer to a structure, the `>` operator compares the addresses of the pointers. The result is likely to be meaningless. One can imagine an implementation of `intSet` using handles to reference an array sorted by size where this would produce the correct result. However, this depends implicitly on the implementation of an abstract type, but changes in the abstract type representation will not produce C compiler errors when the code is compiled. It is likely that difficult to detect bugs will be introduced if a programmer believes the `intSet` type is abstract and changes its representation.

A correct implementation is given by `bigger3`. Here, the abstract `intSet_size` operation is used. As long as `intSet_size` correctly implements its specification, `bigger3` will produce the correct result regardless of the particular representation of an `intSet`.

Only two C operators can be used with abstract types: assignment (`=`), and `sizeof`. Assignment is permitted since its meaning does not depend on the representation of the type, as long as the convention for sharing semantics of mutable types (see Section 1.2.4) is followed.

The permissibility of `sizeof` is based on practical concerns. Programmers often need to use `sizeof` to allocate memory — for instance, if we want to allocate a block of ten elements of an abstract type we need to know the size of each element. This is a legitimate dependence on the representation type, since it is unlikely that changing the representation would cause problems for the client. On the other hand, malicious programmers could easily write clients that depend unacceptably on the representation of an abstract type using `sizeof`. For instance,

```
if (sizeof(x) == 4) crash();
```

Despite this, LCLint does not issue warnings when the `sizeof` operator is used on an abstract type. Remembering that the goal is to detect real bugs without generating spurious messages, it seems appropriate to allow use of `sizeof` on abstract types.

2.2 Globals Checking

As with violating type abstraction barriers, it is problematic for code to depend on global variables not listed in its specification. Clients may have left this variable in an inconsistent state at the time of the function call, unaware that it is used by the called function. It is also likely to be an error if the body of a function does not use each global listed on at least one execution path. This suggests either an unnecessary dependency in the specification, or a missing dependency in the implementation.

LCLint will check that an implementation does not use any global variables that are not listed in its specification, and that each global listed is used somewhere in the function body. LCLint considers a global to be used if it appears in the body of the function or it is listed in the globals list of a called function. Determining exactly what paths through a function may be executed is an undecidable problem, but from an error-checking perspective it is probably more useful to detect textual references. It would be useful if an error could be reported when there is no possible execution of a function that uses a listed global, but this is infeasible.

Figure 2-2 shows typical global usage errors reported by LCLint. The first message reports access to a global variable not listed in the function's global list. The second error illustrates the propagation of global usage through the specification. Since `g` specified to use `glob2`, the call to `g` in constitutes a use of `glob2`. Note that the implementation of `g` is irrelevant — the error is reported based solely on information in the specification of `g` and no inter-procedural analysis is required. The final error reports a global listed in the specification that does not appear in the implementation.

2.3 Modifies Checking

It is often a problem when a called procedure modifies something visible to its caller without the caller's knowledge. LCLint attempts to check that no externally visible value not listed in the function's modifies clause is modified by the body of the function.

In general, determining if something can be modified by a code fragment is undecidable. Given the time constraints on both LCLint's execution and its development, a simplistic view of modification is taken: an object is deemed modified whenever it appears on the left hand side of an assignment statement, is an operand to the increment or decrement operator, or may be modified by a called function according to the called function's modifies clause.

Some typical modification errors are shown in Figure 2-3. The first error reports modification of state visible to the caller through a parameter pointer. The second

Specification: (`globals.lcl`)

```
int glob1;
int glob2;

int f () int glob1; { }

int g (int a) int glob2; { }
```

Implementation: (`globals.c`)

```
1  # include "globals.h"
2
3  int f ()
4  {
5    int a = glob2;
6
7    return (g(a));
8  }
9  ...
```

LCLint execution:

```
globals.c:5,11: Unauthorized use of global glob2
globals.c:7,11: Called procedure g may access global glob2
globals.lcl:4,1: Global glob1 listed but not used
```

Figure 2-2: Global usage errors

Specification: (incInsert.lcl)

```
imports intSet;
int nins;

void incInsert (intSet m, int *a) int nins; { }
```

Implementation: (incInsert.c)

```
 1  # include "intSet.h"
 2
 3  int nins = 0;
 4
 5  void incInsert (intSet s, int *a)
 6  {
 7    *a = *a + 1;
 8    if (intSet_insert(s, *a))
 9      nins++;
10  }
```

LCLint execution:

```
incInsert.c:7,4: Suspect modification of *a: *a = *a + 1
incInsert.c:8,7: Called procedure intSet_insert may modify s:
                   intSet_insert(s, *a)
incInsert.c:9,5: Suspect modification of nins: nins++
```

Figure 2-3: Modification errors

error results from the call to `intSet_insert`. In `intSet.lcl` (see Figure 1-1), `intSet_insert` is specified to modify its first argument, so the call may modify `s`. The final error reports modification of a global variable. Note that unlike parameters, even without a pointer indirection an assignment to a global modifies visible state. Since other functions have access to the global variable, any change in its value is a visible modification.

Unlike globals checking, modifies checking is only done in one direction. No errors are reported if the implementation of a function does not modify all state specified in its modifies clause. The semantics of modifies places no obligation on the implementation to modify anything, it only constrains what must not be modified. In practice, though, a missing modification often suggests a flaw in the specification or implementation. It would be useful to get errors if there is no execution path through a function which produces a specified modification, however, this is not done by the current version of LCLint, and it undecidable in general.

There are several difficulties involved in accurately detecting client-visible modifications. Because of this modifies checking is necessarily unsound and incomplete.

2.3.1 Unseen Modifications

The semantics of LCL is that a modification only violates the specification if the modification is visible in the state of the caller after the function returns. LCLint, however, reports errors whenever a client visible value is modified in the body of a function, without analyzing the modifications to determine if it is visible to the client when the function returns. As a result, LCLint may report modification errors that are not present.

A function may modify a visible value during its execution, but restore the original value before returning. For instance, it may increment a global variable at the beginning of the function, and decrement it before returning on all possible execution paths. The value of the global variable is not modified by the function — its value in the post-state is identical to its value in the pre-state. Determining that a modification is reversed before a function returns, however, is well beyond the scope of a simple static checker.

In other cases, an abstract operation may modify the concrete representation of a type without causing a client-visible modification. There can be several possible concrete representations for an abstract value. Sometimes, switching between different representations can be useful for improving efficiency. For instance we could represent a set using an array. The order of elements in the array is invisible to the client, but it may improve the efficiency of certain operations if we re-order the elements of the array. Since the re-ordered array maps to the same abstract set, there is no modification visible to a client. A modification that switches between concrete representations of the same abstract type is known as a *benevolent side-effect*. Such modifications should not be listed in the modifies clause, since they do not produce changes visible to the client. However, LCLint will not be able to determine that the modification is invisible to the client. Determining if a modification is a benevolent

side-effect would require specifiers of abstract types to provide abstraction functions giving the mapping from concrete representations to abstract values, and require LCLint to analyze changes at a semantic level. As with reversed modifications, this is well beyond what can be done by simple static analysis. LCLint provides control comments for suppressing modification errors when a programmer is aware that an apparent modification will not be visible to the caller.

2.3.2 Aliasing

In C, variables may be pointers that reference memory locations. Since a pointer may reference the location of another variable or a location referenced by another variable, it is possible to modify externally visible state through a local variable. There are also instances where an apparent modification to a parameter variable does not modify caller visible state. The parameter variable may have been assigned locally, so that it no longer references the actual parameter.

It is impossible to statically determine aliases exactly. Even in programs where execution paths can be easily determined, C pointer arithmetic makes static alias detection impossible. C allows arbitrary arithmetic using pointers, so programs can be written that depend unpredictably on the state of the memory system.

Fortunately, in real programs most aliases can be detected. LCLint attempts to analyze aliases in order to minimize the number of unreported modification errors without generating messages regarding modifications that are not present because of incorrect aliases. Rarely, incorrect assumptions are made leading to LCLint recording aliases that are not present. If these aliases refer to client-visible state, spurious modification errors may be issued. It is more common that LCLint will fail to detect an alias which is present. This may lead to modifications to client-visible state in the code that are not reported by LCLint.

LCLint analyzes aliases according to some simplifying assumptions:

- the result of pointer arithmetic does not alias anything
- the return value of a function call does not alias anything, and function calls do not create new aliases
- the possible aliases at the end of a while or for loop are the union of the aliases before the loop and the aliases derived from tracing the body of the loop once
- all paths through conditional branches are possible

The first two assumptions reflect limitations on what can be derived statically. The minimum assumptions have been chosen to prevent spurious aliases. Alternatively, the maximal assumptions could be used to eliminate undetected aliases — the results of pointer arithmetic and function calls may alias anything. These would lead to incorrect aliases being assumed and may generate spurious messages. Another possibility would be to use additional information in the specification to determine

Specification: (alias.lcl)

```
imports intSet ;

int glob;
intSet globSet;

int f(int *a, int **c, intSet s1, intSet s2)
   int nsets; int glob; intSet globSet;
{ modifies nsets, globSet, *c; }
```

Implementation: (alias.c)

```
 1  # include "alias.h"
 2
 3  intSet globSet;
 4  int glob;
 5
 6  int f(int *a, int **c,
 7        intSet s1, intSet s2)
 8  {
 9    int *x;
10
11    x = a;                    x aliases a
12    *x = 4;                   modifies *a
13    *c = &glob;               *c aliases &glob
14    **c = 4;                  modifies glob
15
16    globSet = s1;             globSet aliases s1
17    s1 = s2;                  s1 aliases s2
18    intSet_insert (s1, 4);    modifies s2
19    s2 = intSet_create();
20    intSet_insert (s2, 5);    no visible modification
21    return 3;                 returns with *c aliasing &glob
22  }                                     and globSet aliasing s
```

LCLint execution:

```
alias.c:12,4: Suspect modification of *a through alias *x: *x = 4
alias.c:14,5: Suspect modification of glob through alias **c: **c = 4
alias.c:18,3: Called procedure intSet_insert may modify s2 through
                 alias s1: intSet_insert(s1, 4)
alias.c:21,12: Function returns with parameter *c aliasing global &glob
alias.c:21,12: Function returns with global variable globSet aliasing s1
```

Figure 2-4: Modifications in the presence of aliasing

what the value returned by a function may alias. Then, checking could be done to ensure that functions do not return values that alias variables inconsistently. This would provide the best results, but requires deeper analysis of specifications than is done by the current version of LCLint.

The remaining assumptions are necessary to make alias analysis computationally tractable. Usually it is valid to assume that the possible aliases after many iterations of the loop are identical to those after a single loop execution. This is not true when an alias propagates through a loop through several iterations. For example, consider the loop,

```
while (i < 3) { a = b; b = c; i++; }
```

Suppose before the loop, no variables are aliased and `a`, `b` and `c` are mutable function parameters. After one execution of the loop body, `a` aliases parameter `b`, and `b` aliases parameter `c`. After a second execution, `a` aliases parameter `c` instead.

To further simplify alias analysis, LCLint assumes either branch of any `if` statement may be taken. For conditions involving constants it can sometimes be proven that one branch is always taken. More common, are programs where one condition depends on another one — that is, only some paths through a chain of conditionals are possible. By assuming any may be taken, LCLint may report errors through impossible aliases.

In addition to the problems caused by aliases within a function, checking is jeopardized if a function returns with function parameters aliasing global variables or globals aliasing other globals. Since the body of the caller was checked using the assumption that function calls do not create new aliases, a function that introduces new aliases to global state in its parameters may lead to undetected modifications. LCLint checks that no execution of a function returns with a global variable being aliased by a parameter or another global variable. If a function returns with a parameter aliasing a global variable, the caller now has unrestricted access to the global variable.

Although simple heuristic-based alias analysis can only approximate run-time aliases, it can be done efficiently with a single pass of the source code, and is effective in detecting most aliases in real programs. Modification and global errors detected through aliases are shown in Figure 2-4.

2.3.3 Specification Aliasing

A parallel problem to source code aliasing occurs when the underlying representation in a specification may contain other objects. The LSL sorts used in LCL specifications may contain objects. LCLint cannot determine if objects contained in the underlying representations are caller-visible.

Consider the `hideSet` abstraction specified in Figure 2-5. This uses the LSL trait, `hide`, defined by:

```
hide (S, T) : trait
   T tuple of real: S
```

```
imports intSet;
mutable type hideSet;

uses hide (obj intSet, hideSet);

hideSet hideSet_create (intSet s) {
   ensures result'= [s];
}

bool hideSet_insert (hideSet s, int e) {
   modifies s^.real;
   ensures result = e ∈ s^.real^ ∧ s!real'= insert (e, s^.real^);
}
```

Figure 2-5: `hideSet.lcl`

A `T` in the `hide` trait is a one field tuple (akin to a `C struct`). The uses clause in `hideSet.lcl` makes `hideSet` a `T` where the `real` field is an `obj intSet`. The `obj` before `intSet` means it refers to `intSet` objects, as opposed to their values. So, the `real` field of a `hideSet` may refer to a client visible object, even if the `hideSet` itself does not. This corresponds to having a pointer to a global variable inside a locally declared structure, except that here we are dealing with objects at the specification level.

According to its specification, `hideSet_create` takes an `intSet` and returns a `hideSet` whose value is a tuple containing the `intSet` object. Thus, the `real` field of the returned value is the object `s`. Future modifications to `s` will modify the `real` field of the returned `hideSet`. Likewise, modifications to the `real` field of the `hideSet` will modify `s`. Note that this would not be the case if the uses clause do not use `obj` before `intSet`. Then, the `real` field would contain the value of `s` in a particular state instead of sharing the object.

LCLint cannot keep track of these objects being shared, and it is in general impossible to do this statically. As a result, certain modification errors go undetected.

2.3.4 Missing Specifications

LCLint is designed to be used effectively without having to write specifications for all functions. This causes problems for globals and modifies checking. By default, no globals or modifies errors are reported in unspecified functions. For most applications this is reasonable, since there is no indication of which globals may be used and what state may be modified for an unspecified function. It does mean, however, that some modifications and global uses may go undetected. Flags are provided to override these defaults, so that globals and modifies errors are reported in unspecified functions following the assumption that an unspecified function should not access any globals or modify any visible state.

A more fundamental problem occurs when a specified function makes a call to an unspecified one. Since there is no globals list or modifies clause for the unspecified function, we need to make some assumptions regarding its global usage and caller-visible modifications. One option is to assume an unspecified function uses all global variables and modifies all its parameters and all global variables. This assumption would prevent unreported errors involving the use of unspecified functions, but would produce many spurious messages reporting global usage and inconsistent modifications where unspecified functions are used.

The other extreme is to assume an unspecified function modifies nothing and uses no global variables. This is the approach taken by LCLint. This means global uses and modifications through calls to unspecified functions will be undetected, so some inconsistencies will not be reported. The only spurious messages that will be generated are those when a global is listed in the specification, but the only use in the implementation is through a call to an unspecified function. If we were building a tool intended to verify program correctness missing these errors would clearly be unacceptable. However, given LCLint's goal of finding as many inconsistencies as possible with as few spurious messages, this incompleteness is reasonable.

Another approach would be to do the required analysis to determine possible modifications for unspecified functions. This could not work in general, since the implementation of the function is not necessarily given to LCLint. In the case where the relevant function is in a file checked by LCLint, it would be possible to detect and record modifications in unspecified functions. This may require several passes of the source code as modifications are detected and propagated before actual modification errors can be reported. It did not seem worthwhile to implement such a scheme in LCLint, since writing specifications eliminates the problem.

There is a problem, however, when a module contains hidden (declared `static`) functions. Since these functions are not exported to clients, it would be wrong to write specifications for them in a client-level LCL specification. However, they may be called by exported functions in the module. Modifications and global uses in the hidden function will not propagate to the caller. One solution would be to write specifications for the hidden functions in an alternate specification file that is not intended to be seen by clients.

2.4 Use before Definition Checking

Like many static checkers, LCLint detects instances where the value of a location is used before it is defined. This analysis is done at the procedural level. If there is a path through a procedure that uses a local variable before it is defined, a use before definition error is reported.

LCLint can do more checking than standard checkers, because LCL specifications denote if the values associated with parameters are defined. Normally, if a parameter to a function is a pointer, it is assumed that the value it points to is defined and may be used in the body of the function. This can be a dangerous assumption if the caller

Specification: (outparam.lcl)

```
int f (out int *h, int *w) { }

int g () { }
```

Implementation: (outparam.c)

```
 1  # include "outparam.h"
 2
 3  int f (int *h, int *w)
 4  {
 5    return (*h + *w);
 6  }
 7
 8  int g ()
 9  {
10    int *x, *y;
11
12    return (f(x, y));
13  }
```

LCLint execution:

```
outparam.c:5,19: Variable h used before set
outparam.c:12,18: Variable y used before set
```

Figure 2-6: Use before definition errors involving `out` parameters

expects that the function will use this parameter only to return a value.

In LCL specifications, pointer parameters may be declared with an `out` type qualifier to denote a parameter that is intended only as an address for a return value. The value pointed to by an `out` parameter is undefined when the function is entered. LCLint will report an error if this value is used before it is defined. All other parameters are assumed to be defined when the function is entered. LCLint will report an error if a function is called with an argument that is not defined unless that argument is specified to be an `out` parameter. Calling a function defines the actual arguments associated with `out` parameters.

Typical errors detected involving `out` parameters are shown in Figure 2-6. The specification of `f` declares `h` to be an `out` parameter, so the use of `*h` in line 5 constitutes a use of an undefined variable. The second argument to `f` is not an `out` parameter, so the caller must pass in a defined value. Since `y` is undefined, an error message is generated.

2.5 Macro Checking

C preprocessors provide parameterized text substitution macros. Macros are often used as symbolic constants or as inlined implementations of functions, although more complicated macros that do not emulate functions are sometimes used. Constants and functions specified in LCL can be implemented using macros. There are several pitfalls associated with implementing functions as macros, and faulty macros are a common cause of subtle bugs in C programs.

A client of a module should not be able to tell when a specified function is implemented as a macro. When a specified function implemented as a macro is used, it is checked just like a function call. LCLint checks macros implementing specified functions to ensure that they act like functions from the perspective of the client:

- Each parameter to a macro must be used exactly once in all possible executions of the macro, so that side-effecting arguments behave as expected. The order of evaluation of function arguments is undetermined in C, so it is not an error to use the macro parameters in the wrong order. To be completely correct, all the macro parameters should be evaluated before the macro has any side-effects. Since checking this would require extensive side effects analysis for occasional modest gain, it was not implemented.
- A parameter to a macro may not be used as the left hand side of an assignment or as the operand of an increment or decrement operator in the macro text, since this produces non-functional behavior.
- Macro parameters must be enclosed in parentheses when they are used in potentially ambiguous contexts.
- The type of the macro body must match the return type of the specified function.

Static checkers that do not use specifications have also attempted to check macro usage to detect likely problems. Check [Spu90] detects uses of macros involving side-effecting parameters and operator precedence errors. Errors are reported when a macro is used with a side-effecting parameter regardless of whether the particular macro used could behave unexpectedly. Without specifications, however, there is no way to check the types of the parameters or the return value of the macro. Further, since there is no way for the checker to know which macros are intended to implement functions, the functional behavior of the macro cannot be checked. The checker is limited to syntactic checks depending on presumptions about standard macro usage where the macro is defined and used.

By using specifications, LCLint has information about the intended functionality of a macro and the types of its parameters and result. For most macros, the body can be parsed as though it were a function body or expression and all regular checking can be done. Rarely, programmers may write macros that cannot be parsed normally. LCLint will check as much as possible, and ignore the remainder of the macro.

An example showing macro checking is shown in Figure 2-7.

Specification: (macros.lcl)

```
int first (char a, int b) { }
int choose (int a, int b) { }
int assign (int a) { }
```

Implementation: (macros.h)

```
1  # define first(a, b) (a)
2  # define choose(a, b) (((a) == 2) ? (a) : b + (b))
3  # define assign(a) ((a) = 3)
```

LCLint execution:

```
macros.h:1,1: Macro parameter not used: b
macros.h:1,1: Function first specified to return int, implemented as
                 macro having type char: (a)
macros.h:2,43: Macro parameter used without parentheses: b
macros.h:2,1: Macro parameter used more than once on some path: a
macros.h:2,1: Macro parameter not used on some path, used more than once
                 on different path: b
macros.h:3,21: Assignment to macro parameter: a
```

Figure 2-7: Macro checking

2.6 Other Checks

Some other checks are done using specifications at a global level. Every type, function, variable and constant that is specified should be implemented. Optional warnings can be issued when a variable, function or type is exported but not specified. Since LCLint is designed to work with partial specifications and incomplete source code, this check is not done by default, but can be used to check that the specification is a completely describes the interface to a module.

In addition to checks that depend on specifications, LCLint also incorporates many checks that only depend on the source code. Although our primary research interest is in how specifications can make checking more useful, some useful checks can be done without specifications. Many of these duplicate checks done by traditional lint, such as checking for variables that are declared but never used and return values that are ignored.

More interesting, are those checks that can be used to enforce a particular coding style. These checks do not detect errors in traditional C style, so they do not correspond to anything reported by traditional lint. In certain programming styles, however, they may reveal a problem. By setting flags, programmers can use these checks to verify that a program conforms to an intended coding style, and often catch bugs in the process.

Some of the extended checks derive from stricter type checking of primitive C types.

In standard C, `char` and `int` may be used interchangeably. Programmers can direct LCLint to treat them as distinct types. C does not include a primitive boolean type, but LCL provides a primitive `bool` type. LCLint can treat `bool` as a distinct type (either abstract or exposed) to detect type errors and check that conditional tests are booleans. Primitive C comparison operators (e.g., ==) return `bool`s, and logical operators (e.g., `&&`) take `bool` operands and return `bool`s.

Additional checking is enabled by the specification of C standard library functions. LCLint loads a standard library, which is derived from specifications based on the headers of standard include files. These specifications contain more information than could be derived from the library header files, such as declaring a return value to be type `bool` or a modifies clause. The standard library also declares some abstract types, such as `FILE`. Clients of a standard library are afforded the same checking as clients of a user-specified type.

2.7 LCLint Messages

When LCLint detects an error it prints out a message, consisting of the file name, line and column numbers where the error was found, and a description of the suspected problem. The description attempts to provide sufficient context information without being excessively long. There is no distinction between a *warning* and an *error*. In the text, an error refers to any message produced by LCLint. Often, these are not errors which cause program failures, but violations of stylistic guidelines or inconsistencies between the source code and its specification.

Methods are available for suppressing unwanted messages. Command line options can be used to turn on or off certain classes of checks and make two types indistinguishable. For example, `+boolint` will make `bool` and `int` indistinguishable types, and `-modifies` will turn off all messages related to modifications. Mode flags can be used to set many flags at once. The `-weak` mode sets flags to check according to common C conventions. It turns off all the macro checking, treats `bool`, `char` and `int` as equivalent types, and suppresses many other classes of messages. This is useful for running LCLint on typical C code, but gives up some possibilities for error detection.

The `limit` option is useful for preventing avalanches of messages. It is followed by an integer argument, *n*, and means that if there is a sequence of more than *n*+1 consecutive similar messages, only the first *n* are printed followed by a message telling the number of unprinted similar messages.

In addition, messages can be suppressed locally by stylized comments in the source code. No errors will be reported in code between `/*@ignore*/` and `/*@end*/`. The `ignore` and `end` control comments must be matched — a warning is printed if the file ends in an ignore region. Finer control is provided by stylized comments that allow or disallow access to the representation of particular abstract types, or set command line flags for a section of code and restore them to their original values.

A comprehensive list of flags is given in Appendix A.5.

Chapter 3

Checking Specified Programs

LCLint has been used in a number of different ways, on programs varying from small test examples to real programs (including, of course, LCLint itself). Running LCLint often motivated changes to LCLint. Most of these changes involved adding options to suppress certain messages or adding checks to catch additional problems.

The original purpose of LCLint was to use specifications to check source code as a new system was being developed. Typically, partial specifications would be written, and LCLint would be used to check source code as it was completed and the specifications were refined. In the course of this research, other uses of LCLint became apparent. The most significant of these is using LCLint to aid in software maintenance, which is described in the next chapter.

Since LCL was in use before the development of LCLint, there existed programs with complete specifications which had never been checked. One such program is the `dbase` example from the Larch book [GH93, Chapter 5]. This example comprises seven specified modules and an unspecified test driver comprising over 300 lines of specifications and 800 lines of code. The program is a database of employee records. The modules are:

- `employee` — an exposed type for representing employee records
- `empset` — a mutable abstract type for representing sets of employees
- `eref` — an immutable abstract type for referencing an employee (similar to a pointer)
- `erc` — a collection of `erefs`
- `ereftab` — a table of `erefs`
- `dbase` — the top-level interface, including operations for hiring, firing, promoting employees and querying the database.

The specifications had been checked by the LCL checker (whose functionality is now subsumed by LCLint), and the source code had been compiled and tested extensively.

```
% lclint drive.c dbase.c ereftab.c erc.c eref.c empset.c employee.c
LCLint 1.2 --- 05 May 94

drive.c:41,6: Return value (type bool) ignored: employee_setName(&e, na)
drive.c:42,6: Return value (type bool) ignored: empset_insert(em1, e)
drive.c:51,6: Return value (type bool) ignored: employee_setName(&e, na)
drive.c:52,6: Return value (type bool) ignored: empset_delete(em1, e)
drive.c:62,6: Return value (type bool) ignored: employee_setName(&e, na)
drive.c:63,6: Return value (type bool) ignored: empset_insert(em2, e)
drive.c:74,6: Return value (type bool) ignored: empset_delete(em3, e)
drive.c:86,6: Return value (type bool) ignored: employee_setName(&e, na)
drive.c:87,25: Return value (type db_status) ignored: hire(e)
drive.c:93,4: Return value (type bool) ignored: fire(17)
drive.c:113,4: Return value (type bool) ignored:
                   fire(((eref_Pool.conts[((em3->vals)->val)])).ssNum)
drive.c:11,26: Parameter not used: argv
dbase.c:55,7: Return value (type bool) ignored: empset_insert(s, e)
dbase.c:93,9: Return value (type bool) ignored: erc_delete(db[i], er)
dbase.c:114,6: Return value (type bool) ignored:
                   erc_delete(db[mNON], er)
dbase.c:118,6: Return value (type bool) ignored:
                   erc_delete(db[fNON], er)
dbase.c:137,8: Variable declared but not used: er
dbase.c:138,12: Variable declared but not used: e
ereftab.c:21,3: Return value (type bool) ignored: erc_delete(t, er)
empset.c:22,8: Variable declared but not used: er
empset.c:90,5: Return value (type bool) ignored: erc_delete(s1, er)
empset.c:95,12: Variable declared but not used: e

Finished LCLint checking --- 22 code errors found
```

Figure 3-1: Checking `dbase` without using specifications

Since the code and specifications were written by experts, and checked copiously by hand prior to publication, it was expected that not many bugs would be found.

By running LCLint on this type of system, we hoped to find the types of bugs that can be detected automatically, but are often overlooked by humans. This would give some idea of LCLint's usefulness as a supplement to human checking, but would not provide insights into LCLint's effectiveness during the development process.

3.1 Code Checks

First, we consider the messages which could be detected without using specifications. To begin, LCLint is run on all source files without using any specifications (Figure 3-1). These errors are less interesting than those detected when specifications are

used, since they reveal problems that could also be detected by a standard lint. They illustrate how LCLint can be used to enforce style conventions.

Four of the messages (e.g., `dbase.c:137,8`) concern unused variables. These are harmless errors, but we can clean up the code by removing the unnecessary declarations. Alternately, we can use the `-varuse` option to suppress the messages. Unless we are particularly worried about making changes to the code, it is best to just remove the unnecessary declarations.

A similar error (`drive.c:11,26`) reports a parameter not being used. Unlike local variables, there are often good reasons why a function may not use some of its parameters. For example, we may want to make it type-compatible with some other function so that they may both be passed as function pointers with the same type. Here, the parameters to `main` are fixed, so we cannot remove the unused argument from the parameter list. Instead, we suppress the message using `-paramuse`.

The remaining errors all concern ignored return values. Often, ignoring a return value reveals a missing test of the error status of a function call. Since C provides no exception mechanisms, it is a common programming idiom to return a value denoting the success or failure of a call. Here, the function `employee_setName` returns `FALSE` if its second argument, an array of characters, is too long to be stored in the employee record. Otherwise, it sets the employee name and returns `TRUE`. Unless we know the name we are passing to `employee_setName` is small enough, failing to test the return code may lead to unexpected problems.

In other situations, it is not dangerous to ignore a return value. The `empset_insert` and `empset_delete` operations return a `bool` indicating if the employee we inserted was already in the set. This information may sometimes be useful to the caller, but ignoring it is harmless.

There are several approaches to eliminating the return value errors. The most robust approach is to examine each return value error — if it reveals a missing error test, add code to check the return value; otherwise, add a `(void)` cast before the call to denote that the result is safely ignored. This has two disadvantages — it requires checking every error (here there are only 17 return value errors, so this is not unreasonable), and it clutters the code with `(void)` casts. Another possibility is to use the `-returnval` flag to suppress all messages regarding ignored return values. This is quick and simple, but it abandons the possibility of detecting cases where a return value is incorrectly ignored. Short of this, we could use `-returnvalbool` to suppress only those messages concerning ignoring return values of type `bool`. For some coding styles, this may be the best approach if `bool` return values tend to be used to return non-essential information to the caller. But this would miss cases like the call to `employee_setName`, where ignoring a `bool` return value may be a bug in the program.

Perhaps a better approach is to introduce an abstract type, `status`, for representing return status codes. A simple implementation of `status` could use a boolean representation and provide the operations for creating success and failure status codes and checking if a `status` value is success or failure. We can adopt a conven-

tion that operations returning success or failure status codes are declared to return `status`. Because this has been specified as an abstract type, it is distinguished from `bool` by LCLint. The difference between the return types of `employee_setName` and `empset_insert` is now made apparent in their declarations: `employee_setName` returns a `status` representing an exit code while `empset_insert` returns a `bool` which may be safely ignored.

Now we can use the `-returnvalbool` flag to suppress errors relating to unimportant ignored return values, without losing relevant messages. This generated five messages regarding ignored return values. Each reveals an untested status return code, so we add code to check the return status. LCLint has provided the means to a more robust coding style — we now explicitly distinguish between functions returning booleans that may be ignored, and those returning status codes that must be checked. Although standard lint can detect ignored return values, the added flexibility of LCLint combined with abstract types allows us to eliminate spurious errors and focuses our attention on relevant ignored return values.

By running LCLint on the program without using any specifications, we detected some simple problems in the code. None of these were actual bugs, except perhaps some of the ignored return value errors. We also learned about the style conventions followed by the program. Here, more was discovered by the errors that LCLint did not report than those it did. Since no errors were uncovered regarding `bool`s, we can infer that the program treats `bool` as a distinct type.

3.2 Specification Derived Checks

The result of running LCLint on the revised code using its specification is shown in Figure 3-2. Note that we have added the `+showfunc` flag so that each message reported is preceded by the name of the function in which it appears, and the `-paramuse` and `-returnvalbool` flags as discussed in the previous section.

The first three messages and a later message for `erc.h:15,25` concern the use of macro parameters without parentheses. Since the macros were specified as functions, LCLint now checks that they always behave as functions. A static checker could do this check without specifications (as Check [Spu90] does), but has no way of preventing the same errors from being reported in macros which are not intended to implement functions. Since macros are just text substitutions, programmers may want to write macros which do not use parentheses around their parameters. This is acceptable as long as the macro is not intended to implement a function. We add parentheses where suggested to eliminate the macro errors.

None of the remaining errors could be detected without using specifications, so the added benefits of specifications are apparent. They are errors only because of information given in the specification. This leaves open the question whether the error is in the implementation or the specification. Often, such as when an abstraction barrier is broken, it is clear the problem is in the implementation. In other cases, such as when the wrong global is listed in the specification, the problem is in the

```
% lclint -paramuse -returnvalbool +showfunc drive.c dbase eref
         erc ereftab empset employee status
LCLint 1.2 --- 05 May 94

eref.h: (in macro eref_assign)
eref.h:21,51: Macro parameter used without parentheses: e
eref.h: (in macro eref_equal)
eref.h:23,31: Macro parameter used without parentheses: er1
eref.h:23,38: Macro parameter used without parentheses: er2
erc.c: (in function erc_member)
erc.c:34,9: Operands of == are abstract type (eref): tmpc->val == er
erc.c: (in function erc_delete)
erc.c:58,9: Operands of == are abstract type (eref): elem->val == er
erc.c: (in function erc_sprint)
erc.c:110,28: Called procedure erc_iterStart may modify c:
                 erc_iterStart(c)
erc.h: (in macro erc_choose)
erc.h:15,25: Macro parameter used without parentheses: c
ereftab.c: (in function ereftab_lookup)
ereftab.c:30,28: Called procedure erc_iterStart may modify t:
                   erc_iterStart(t)
empset.c: (in function empset_disjointUnion)
empset.c:58,28: Called procedure erc_iterStart may modify s2:
                  erc_iterStart(s2)
empset.c:58,28: Called procedure erc_iterStart may modify s1 through
                  alias s2: erc_iterStart(s2)
empset.c: (in function empset_union)
empset.c:75,28: Called procedure erc_iterStart may modify s1:
                  erc_iterStart(s1)
empset.c:75,28: Called procedure erc_iterStart may modify s2 through
                  alias s1: erc_iterStart(s1)
empset.c: (in function empset_subset)
empset.c:99,28: Called procedure erc_iterStart may modify s1:
                  erc_iterStart(s1)

Finished LCLint checking --- 13 code errors found
```

Figure 3-2: Checking `dbase` using specifications

specification. Many times, as we shall see, it is less clear — there is a problem somewhere since the specification and implementation are inconsistent, but it is not obvious where the fault lies and how it should be fixed.

The next two messages report the use of a C operator with abstract types. Line 34 for `erc.c` is,

```
if (tmpc->val == er) return TRUE;
```

where `er` has type `eref` and `tmpc` has type `ercList` which is a pointer to a structure whose `val` field has type `eref`. In `eref.lcl`, `eref` was declared to be an immutable abstract type. So the expression, `tmpc->val == er`, is an equality test on abstract types. Without knowing the representation of `eref` we cannot be sure what this statement means (see Section 2.1). If they are pointers, C will do a pointer equality test (i.e., are they the same object), and we are depending on `eref` being implemented in such a way that equal `eref`s always use the same pointer. In the given implementation, `eref` is represented by an `int` handle and the test produces the desired semantics. However, there is a dangerous hidden assumption about how `eref` is implemented.

This can be fixed by adding an `eref_equal` operation to `eref` which does equality checking. If we are concerned with efficiency, `eref_equal` could be implemented as a macro with no efficiency loss. Then line 34 can be rewritten as,

```
if (eref_equal(tmpc->val, er)) return TRUE;
```

The message for line 58 also reports an equality comparison involving `eref`s, which can be rewritten in a similar manner.

The remaining seven errors reported concern modifications. They illustrate two of the difficulties in accurately detecting client-visible modifications — modifications that are hidden through a specification alias to an object (Section 2.3.3), and modifications that are unseen by clients since they are reversed before the function returns (Section 2.3.1).

Only one message uncovers a bug, although the other errors draw our attention to potentially confusing code. All the errors concern a macro for iterating through the elements in an `erc`. Because C provides no mechanisms for iteration abstraction, there is no easy way to cycle through each element of an abstract type. The approach taken by this program is to define an `ercIter` type that can be used to return a different element of an `erc` until every element has been returned.

The message for `erc.c:100,28` reports a modification of `c` in the implementation of `erc_sprint`. The specification of `erc_sprint` is:

```
char *erc_sprint(erc c) {
   ensures isSprint(result[]', c^) ∧ fresh(result[]);
   }
```

There is no modifies clause, so no externally visible values should be modified, including the parameter which is a mutable abstract type. The relevant lines in the

implementation of erc_sprint are shown below:

```
110    for_ercElems (er, it, c) {
111      employee_sprint(&(result[len]), eref_get(er));
112      len += employeePrintSize;
113      result[len++] = '\n';
114    }
```

The modification error was reported for line 110, which instantiates the following macro:

```
#define for_ercElems(er, it, c)\
   for(er = erc_yield(it = erc_iterStart(c));\
       !eref_equal(er, erefNIL);\
       er = erc_yield(it))
```

The function erc_iterStart returns an ercIter, a mutable abstract type specified in erc.lcl. An ercIter iterates through the erefs in an erc. Since it is the result of erc_iterStart(c), then each call to erc_yield(it) returns an element of c which has not already been returned, or erefNIL if all elements of c have been returned.

The specification of erc_iterStart is:

```
ercIter erc_iterStart(erc c) {
   modifies c;
   ensures fresh(result) ∧ result'= [c^.val, c]
    ∧ c'= startIter(c^);
   }
```

In the implementation of erc_sprint, erc_iterStart(c) is called, where c is the parameter to erc_sprint. Because of the modifies clause in erc_iterStart, this constitutes a modification of c as reported in the message.

At this point, we may be tempted to add a cursory modifies clause to erc_sprint to indicate that c may be modified. This would eliminate the inconsistency between the implementation and specification, so LCLint would no longer report an error. However, the specification of erc_sprint would be wrong — as far as the client is concerned, it does not make sense for it to modify c. We need to look more closely at the erc module to understand the real problem.

The interface specification of erc uses the sorts erc and ercIter defined by an LSL trait. An erc is a tuple of two fields: val, a collection of erefs, and activeIters, an int. The activeIters field maintains a count of the number of active iterators associated with this erc. An ercIter is a tuple consisting of toYield, the elements of the erc that have not yet been returned, and eObj, the erc object that was used to create this ercIter. This exhibits the problem described in Section 2.3.3, where an object inside an underlying representation hides a client-visible modification.

The operator startIter produces an erc with the same elements as its argument, but one more active iterator. The inverse operator, endIter, decrements the number

of active iterators. The result of erc_iterStart is specified in its ensures clause by the first two conjuncts. It returns a fresh ercIter whose toYield field contains the elements of the pre-state value of c and whose eObj field is the object c. This means future modifications to the eObj field of the result will constitute indirect modifications of the parameter used to create the ercIter.

The specification of erc_iterStart also reflects the creation of a new active iterator for c. The final conjunct in the ensures clause is c' = startIter(c^). The post-state value of the parameter is the result of applying the startIter operator to its pre-state value. Hence, in the post-state, c has one more active iterator.

The code for for_ercElems loop calls erc_yield(it) until an erefNIL is returned. The specification of erc_yield is:

```
eref erc_yield(ercIter it) {
   modifies it, it^.eObj;
   ensures if it^.toYield ≠ { }
     then yielded(result, it^, it') ∧ (it^.eObj)'= (it^.eObj)^
     else result = erefNIL ∧ trashed(it)
            ∧ (it^.eObj)'= endIter((it^.eObj)^);
   }
```

The ensures clause has two branches. If there are more elements to yield, a new value is yielded and the value of the eObj field of the ercIter does not change. Otherwise, erefNIL is returned and the post-state value of it^.eObj is the result of applying endIter to its pre-state value. That is, the object which was used to create this ercIter now has one fewer active iterator. In the for loop, the argument to erc_yield was returned by erc_iterStart(c), so we know it^.eObj refers to the same object as c. Thus, when erc_yield returns erefNIL, it decrements the number of active iterators associated with c.

Note that this happens only once, since when erc_yield returns erefNIL, the loop terminates. Thus, after completing the loop there have been two modifications to c — the number of active iterators was incremented by the call to erc_iterStart and decremented by the final call to erc_yield. As long as the loop runs to completion, there is no modification visible to the client since the modification caused by erc_iterStart is reversed before the function returns (see Section 2.3.1).

This analysis rests on two assumptions: no element in c^.val is erefNIL and the loop runs to completion. This first assumption is necessary so we know the only time erc_yield returns an erefNIL (and hence, terminates the loop) is after every element of the erc has been yielded. This is guaranteed by the requires clause for erc_insert which prohibits inserting erefNIL into an erc.

The second assumption may be violated when there is a control flow statement in the loop body. If the loop may exit without completing, we need to make sure the code ends the iterator, otherwise there is indeed a client visible modification.

The body of the loop in erc_sprint which generated the first error message has no control statements. Hence, we are guaranteed that c is not modified by the for_ercElems loop. We can use stylized comments to document this in the code

and suppress the LCLint messages. Line 110 is surrounded by control comments to temporarily turn off modifies checking:

```
/*@-modifies*/ for_ercElems (er, it, c) /*@=modifies*/ {
```

Now, we consider the other modification errors reported, to check if the loops run to completion. The next error concerns the following code in `ereftab_lookup`:

```
30    for_ercElems(er, it, t) {
31        e1 = eref_get(er);
32        if (employee_equal(&e, &e1)) return er;
33        }
```

Here, our second assumption is violated — the return on line 32 may prevent the loop from running to completion and the initial modification to `c` will not be reversed. We fix this by adding a call to `ercIter_final` before the return. Line 32 is rewritten,

```
if (employee_equal(&e, &e1)) { erc_iterFinal(it); return(er); }
```

Now we can guarantee that the loop does not modify `c` since it either runs to completion or calls `erc_iterFinal` and returns.

Note that this unspecified modification reveals a real bug in the code — without the call to `erc_iterFinal`, there is a storage leak. The `ercIter` returned by the call to `erc_iterStart` would never be freed in the original code if the loop does not run to completion.

The next two errors concern `empset_disjointUnion`:

```
empset.c:58,28: Called procedure erc_iterStart may modify s2:
                    erc_iterStart(s2)
empset.c:58,28: Called procedure erc_iterStart may modify s1
                    through alias s2: erc_iterStart(s2)
```

The same call appears to modify both `s1` and `s2`. Looking at the code preceding the `for_ercElems` instantiation, we see that it could indeed modify either `s1` or `s2`:

```
if (erc_size(s1) > erc_size(s2)) {
  tmp = s1;
  s1 = s2;
  s2 = tmp;
}
```

The code swaps `s1` and `s2` if the size of `s1` is greater than the size of `s2`. So if the test is true, `s2` becomes an alias for the parameter `s1`. If it is false, `s2` still refers to the original argument. As with `erc_sprint` there is no control flow within the loop, so it is guaranteed to complete and not modify the `erc`.

The next two messages, which concern `empset_union`, are similar. The final message concerns `empset_subset`:

```
 99      for_ercElems(er, it, s1)
100        if (!empset_member(eref_get(er), s2))
101           erc_iterReturn(it, FALSE);
```

Here, `erc_iterReturn` is a macro defined to call `erc_iterFinal` on its first argument, and return its second (the earlier fix to `ereftab_lookup` could use this macro). This correctly reverses the modification, so no client-visible modification is apparent.

It is interesting to note that although the modification bug reported by LCLint is only visible at the specification level, the modifies checking has incidentally uncovered a real bug in `ereftab_lookup`. Compared to `empset_subset`, it is clear the return in the loop body should have been an instantiation of `erc_iterReturn` instead.

It may seem accidental that this bug was found since it is not directly related to LCLint checks. However, the modification error detected by LCLint directed our attention to a segment of code containing the bug. Finding bugs not directly related to LCLint checks is not as uncommon as one might suspect. It is often the case that simple modification, globals or type abstraction errors reported by LCLint reveal more fundamental problems in specifications or code.

As expected, running LCLint on the `dbase` example did not find many significant problems. It did uncover two abstraction violations, and one legitimate modification error. Because modifies checking is unsound, it reported several incorrect modification errors. These illustrate some of the difficulties involved in accurate modifies checking — determining that the modification was reversed involved reasoning about the underlying semantics of the specification and the control flow of the code. Clearly, these types of analyses are well beyond the scope of a simple static checker. However, the places where LCLint incorrectly reports modifications are also likely to be difficult for programmers to understand. By drawing our attention to these points, LCLint helps us understand the code, and may lead us to discover subtle errors.

Chapter 4

Maintaining Programs

The previous chapter illustrated how LCLint can be used to check specified programs. This chapter shows how LCLint can be used to understand and maintain existing C programs that have no formal specifications. It is common for a programmer to have to maintain and make changes to a program written by someone else, often without adequate documentation. When confronted with a large program for the first time, it is helpful to identify the abstract types. Without a tool like LCLint, we can only guess if a type is abstract. By using LCLint we can verify type abstractions and often find a few instances where an intended abstract type is exposed. By specifying the abstract types and their interfaces, we will gain a high-level understanding of the program and produce a program that is better documented and easier to maintain. Further, in the process of writing the specifications and checking the source code, we may uncover bugs.

This is done in three phases. First, we run LCLint on the source code with no specifications. Next, minimal specifications are written to make types abstract and LCLint is used to check source code against the new specifications. Finally the specifications are developed to include more information together with more LCLint checking.

This example uses `quake`, a 19-file, 5000-line program for automating system builds in Modula-3. The code for `quake` was provided by Steve Harrison at DEC SRC.

4.1 No Specifications

Before writing specifications for an unfamiliar program, it is helpful to get a feel for the coding style used. We can use LCLint's flags to discover what coding conventions are followed.

Running LCLint with no flags on all the source files of `quake` yields 169 messages. An inexperienced LCLint user would now be encouraged to use the `-weak` flag. This sets several flags to reflect a common loosely-typed C style. Running LCLint with the `-weak` flag on all the source files yields a manageable 13 messages. We could accept the coarseness of the `-weak` flag at this point, and continue by checking the reported

errors. Instead, it is instructive to use the fine-grain control flags instead. This way we can determine the actual conventions followed by this code.

Many of the errors yielded when we run LCLint with no flags relate to the `bool` type. Since `quake` was developed with no knowledge of LCL it had used `Boolean`, `typedef`ed to `int`, instead of the conventional `bool`, to represent booleans. We could use the `+boolint` flag to make `bool` and `int` (hence, `Boolean`) interchangeable. This, however, would sacrifice all the boolean checking. Instead, we can change the `typedef` of `Boolean` from its original type of `int` to the `bool` type and include the standard header implementing `bool`. Assuming the original `Boolean` type was used in the conventional way, this has no effect on the execution, but allows LCLint to check booleans as a distinct type. This change eliminates 79 of the original 169 errors.

Half of the remaining errors involve ignored return values of type `int`. Traditionally, C programmers declare functions with no return value to return `int` since functions with no declared return type are implicitly assumed to return `int`. To support this, LCLint provides the `-returnvalint` flag (analogous to `-returnvalbool` used in Section 3.1) for suppressing just those ignored return value messages where the return value is an `int`. The large number of error messages regarding ignored `int` return values may lead us to conclude that the coding style employed does not considered ignoring an `int` return value to be an error. Looking a little further, we see that all the return value errors regard calls to `yyerror`, a function declared by the `yacc` parser generator. If this were a user-defined function we could change the declaration to return `void`. Since it is not, we use a macro to call `yyerror` and cast the result to `void`. By fixing it this way, instead of using the `-returnvalint` flag, we preserve a coding style where all return values are relevant, and do not lose checking for other functions that return `int`s.

Running LCLint following these changes produces 45 messages. Eight of these involve type errors between `char` and `int`. If there were only one or two, we might decide these were mistakes and fix them. Instead, we decide that the style employed uses `char` and `int` interchangeably. The `+charint` flag reflects this convention.

Seven messages report type matching and casting errors between pointers to the abstract type `FILE` and `void` pointers. `FILE` is a built-in abstract type, defined in the standard library. Strictly, this is an abstraction violation — if we can cast an abstract pointer to a `void` pointer then when it is referenced there is no longer any type safety. In practice, however, it is often necessary to do this — for example, many generic data structures use `void` pointers to get around C's lack of polymorphism. Many of the instances reported are initializations of `FILE` pointers to `NULL`. The others are casts from `void` pointers in a generalized list data structure. In this case, we are willing to accept the abstraction violation as a matter of convenience. It would be just too awkward in some styles of C coding to disallow casts between pointers to abstract types and `void` pointers. The `+voidabstract` flag reflects this convention, allowing `void` pointers to match pointers to abstract types.

Running LCLint with the `+charint` and `+voidabstract` flags yields the 30 errors shown in Figure 4-1. Although the initial 169 messages may have seemed overwhelming, we have eliminated 139 of them by making a few simple changes to the code and

```
% lclint  -limit 1 +charint +voidabstract Array.c Atom.c builtin.c \
    code.c dict.c Execute.c file.c Hash.c iostack.c lexical.c list.c \
    Name.c operator.c path.c quake.c Set.c stack.c String.c utils.c
LCLint 1.2 --- 07 May 94

builtin.c:111,35: Parameter not used: argc
builtin.c: (11 more similar errors unprinted)
builtin.c:436,2: Return value (type struct Atom * *) ignored:
   Dict_Install(name, Atom_Builtin(name, f->is_function, f->argc,
   f->operator), (1 << (1)))
code.c:22,2: Return value (type Array) ignored:
                Array_AppendAtom(code->array, atom)
operator.c:99,2: Return value (type struct Atom * *) ignored:
                    Dict_Install(designator->u.name, value, dict_flags)
operator.c:204,2: Return value (type Set) ignored:
                     Set_InsertAtom(set, key, value)
operator.c:331,2: Return value (type struct Atom * *) ignored:
                     Dict_Install(subject, atom, (1 << (0)))
operator.c:400,6: Return value (type struct Atom * *) ignored:
         Dict_Install(procedure->arg_names[arg], Pop_Any(), (1 << (0)))
path.c:88,11: if predicate not bool, type int: sp
path.c:77,10: while predicate not bool, type char: *src
quake.c:33,5: Return value (type struct Atom * *) ignored:
  Dict_Install(Name_Register(string), Atom_String(String_New(sep)), (0))
quake.c:49,3: Return value (type ExitCode) ignored:
                 Execute_Stream(stdin, String_New("* stdin *"))
quake.c:53,3: Return value (type ExitCode) ignored:
   Execute_File(Path_ExtractPath(temp), Path_ExtractFile(temp))
Set.c:169,5: Return value (type Array) ignored:
   Array_AppendAtom(ToArrayTarget, Atom_String(String_New(bucket->key)))
stack.c:33,2: Return value (type Atom) ignored:
                 Atom_CheckType(Stack[StackPtr], tag)
String.c:38,26: initialized initialized to type bool, expects int: FALSE
String.c:40,10: Operand of ! is non-boolean (int): initialized
String.c:44,2: Assignment of bool to int: initialized = TRUE
utils.c:101,5: Return value (type void *) ignored:
                  memcpy(to, from, bytes)
utils.c:106,5: Return value (type void *) ignored:
                  memset(dest, 0, bytes)

Finished LCLint checking --- 30 code errors found
```

Figure 4-1: Checking `quake`

by adding two command line flags.

The first twelve errors reported (11 of which are not printed because of the `-limit 1` flag) concern unused parameters in `builtin.c`. They all concern an unused integer parameter declared `int argc`. Since function passing in C is limited to functions having the same type, the unused parameter is needed so that these functions may match other functions that need this parameter. We could use the `-paramuse` flag to suppress these messages. However, since all the relevant errors are in one file, it may be better to use a local control comment instead. The `/*@-paramuse*/` control comment is inserted at the beginning of `builtin.c`. This way, we eliminate the messages for `builtin.c` without losing the checking in other files. If we are even more concerned about ignored parameters, we could place `/*@-paramuse*/` and `/*@=paramuse*/` control comments around the particular functions that do not use a parameter.

The next six messages, and seven others, concern ignored return values. Earlier, we removed the ignored `int` return value errors generated by calls to `yyerror`. The remaining errors concern a number of different return types and functions. We could use the `-returnval` flag to suppress all ignored return values messages. However, an ignored return value is often evidence of a real bug, such as failing to check the return status code of a function call. So, it is worth checking each one individually. Those that may be ignored are cast to `void` to make it clear when a return value is being legitimately ignored. In two cases, the ignored return value is an `ExitCode`, and ignoring it is a bug in the code. We add code to check the return value, and exit appropriately if it is invalid.

This leaves five messages. The first two deal with non-boolean predicates:

```
path.c:88,11: if predicate not bool, type int: sp
path.c:77,10: while predicate not bool, type char: *src
```

In C, a predicate may be any non-structure type — this is a common cause of bugs that are not detected statically. LCLint checks that predicates have type `bool`. This check can be turned off using `-pred`. However, since there are only two instances of this it is likely that the coding style uses `bool` predicates. We replace implicit tests causing these messages with explicit inequality comparisons with `0` and `'\0'` respectively to eliminate the messages.

The final three messages,

```
String.c:38,26: initialized initialized to type bool, expects int: FALSE
String.c:40,10: Operand of ! is non-boolean (int): initialized
String.c:44,2: Assignment of bool to int: initialized = TRUE
```

concern this code fragment:

```
38      static initialized = FALSE;
39
40       if (!initialized) {
41          String_False = String_New("");
42          String_True  = String_New("true");
43
44          initialized = TRUE;
```

The variable `initialized` is implicitly declared to be an `int`, when it should be a `bool`. Changing the declaration in line 38 to `Boolean` fixes this problem, and eliminates the messages. Since `Boolean` is represented by an `int`, this inconsistency does not lead to program faults. However, using the `Boolean` declaration makes the code more readable.

Now, LCLint runs on all the source files with no errors using two flags. We have learned a lot more about the code than we would have by just using the `-weak` flag. For instance we know that the code treats booleans and integers differently, but uses chars and integers interchangeably. In some cases, we have made rather arbitrary decisions about the intended coding style. When a particular check floods us with errors, we decide that the check does not apply to the coding style and turn the check off. In other cases, a check leads to just a few errors. We could justify turning the check off and presuming it is also accepted in the coding style. It is usually better, though, to get as much checking as possible by fixing these few instances that do not conform to the checked programming convention.

We have used LCLint's flags to analyze an unknown coding style, without making any major changes to the code. In the process a few bugs were found and some potentially confusing code was made more readable. The flexible flag settings and local control comments of provided by LCLint make it easy to customize checking for a particular coding style. A similar approach could be taken to enforce a desired coding style instead. We could have begun with desired flag settings in mind, and adapted the code to conform to them.

4.2 Adding Minimal Specifications

In the previous section we saw how LCLint can be used without any specifications to get a handle on an unknown coding style and detect certain classes of bugs, many of which could also be found by a regular lint. The main benefits of LCLint are realized after we write specifications. In this section, we show how minimal specifications can be added to get significant checking benefits. A one-line specification declares a type to be abstract. This leads to greatly enhanced checking, and further increases our understanding of the code as well as our confidence in its correctness.

To begin, we need to decide which of the types are intended to be abstract. When confronted with an unspecified program, there are many clues about which modules are meant to represent abstract types. The most superficial clue is the name of the file — good programmers will give their abstract data structures recognizable names. We can now use LCLint to confirm that they are used as abstract types, and

```
% lclint  +charint +voidabstract Array.c Atom.c ... utils.c Set.lcl
LCLint 1.2 --- 06 May 94

builtin.c:131,13: Arrow access field of abstract type (Set): set->body
builtin.c:132,17: Arrow access field of abstract type (Set): set->body
operator.c:282,14: Arrow access field of abstract type (Set): set->body
operator.c:283,14: Arrow access field of abstract type (Set): set->body
operator.c:283,35: Arrow access field of abstract type (Set): set->body
operator.c:529,36: Arrow access field of abstract type (Set): set->body
operator.c:581,33: Arrow access field of abstract type (Set): set->body

Finished LCLint checking --- 7 code errors found
```

Figure 4-2: Checking `Set` is abstract

typically uncover abstraction violations and other bugs in the process. This section describes the process of making types abstract, and illustrates typical errors detected and how they can be fixed. It focuses on two modules of `quake` — `Set` and `Hash`. Each reveals some of the benefits achieved by making types abstract, as well as some of the difficulties involved in programming in C without violating abstraction boundaries.

Set

The module name `Set` suggests that the module is intended to be an abstract type representing the standard notion of a mathematical set. As it happens, what `Set` implements is actually a key-value table. We can check if `Set` is an abstract type by writing a specification file, `Set.lcl`, containing the single line:

```
mutable type Set;
```

This declares `Set` to be a mutable abstract type. To decide that it should be mutable we looked to see if it provides any operations that may change the value of a `Set`. Since the operation `Set_InsertAtom` inserts a new atom into its `Set` argument we declare `Set` to be a mutable type.

We add `Set.lcl` to the command line, and run LCLint as before. LCLint reports seven places where `Set` is exposed, shown in Figure 4-2. Now we must decide if `Set` was intended to be an abstract type. If no errors had been reported, then `Set` is consistently used abstractly and no further work is necessary. If many errors had been reported, we might question our hypothesis that `Set` is intended to represent an abstract type and declare it instead as an exposed type.

In this case seven errors are reported, all involving accessing the `body` field of a `Set`. This suggests that `Set` is intended to be an abstract type, and these places are abstraction violations that should be examined and recoded. Although it would be less work to just say `Set` is an exposed type, this would give up the benefits of type

abstraction — we could no longer change the `Set` implementation without worrying about introducing problems in other parts of the code. Fixing the abstraction violations will not only allow us to treat `Set` as an abstract type, but will make the client code shorter and more readable. An abstraction violation typically suggests a flaw in either the abstraction or the client, and occasionally a more serious design problem. Unless the client simply overlooked a provided abstract operation, the abstraction violation was necessary because the abstract type did not provide an operation to do what the client needed.

The quickest fix would be to introduce a `Set_getBody` operation that returns the `body` field of a `Set`. Although this would eliminate the error messages, it is not a satisfactory solution. For one thing, `Set_getBody` does not correspond to anything in our abstract notion of what a `Set` is. There is no convenient way to describe it using this particular `Set` implementation. Worse, if the `Set_getBody` operation simply returned the `body` field of its argument it would expose the representation of `Set`. The body of `Set` is a `Hash_Table`, which can be mutated. A client of `Set` could use the `Set_getBody` operation to get the `Hash_Table` associated with a key in a `Set` and then use `Hash_Table` operations directly to manipulate the `Set`. This violates a fundamental principle of type abstraction — we are changing the concrete value of the abstract type without using its defined operations. We can no longer reason about properties of the abstract type, since there is no guarantee that its representation will not be changed arbitrarily from outside. Hence, the implementation of `Set_getBody` must return a fresh copy of the `Hash_Table`. This would avoid exposing the representation, but is too inefficient for most applications. While we can use this approach to eliminate the error messages, it does not address the underlying problem of why the abstraction was violated in the first place.

Abandoning the simplistic approach, we need to look at the code fragments where errors are reported to see if more acceptable solutions can be found. Understanding why the client needed to access the type representation reveals the inadequacies of the `Set` abstraction.

The first two messages relate to this code fragment from `builtin.c`:

```
130        is_empty =
131            set->body == NULL ||
132                set->body->nEntries == 0;
```

This checks if the set is empty. Note that this code uses an awkward level of detail for a client of `Set`. We should not have to understand how `Sets` are represented to understand the `builtin` module. The need to access the representation directly results from the `Set` abstraction missing the needed operations. We can preserve our abstract `Set` type, and clarify the code by adding a `Set_isEmpty` operation to the `Set` module and replacing this code with a call to `Set_isEmpty`. If we are concerned about efficiency, `Set_isEmpty` can be implemented as a macro. Thus, we have removed an abstraction violation and made the code more readable without any efficiency penalty.

```
277 static ExitCode ForeachSet(Set set, Name subject, Code code)
278 {
279     List *l;
280     Atom *atom = Dict_Install(subject, NULL, DICTFLAGS_LOCAL);
281
282     for (l = set->body->buckets;
283          l < set->body->buckets + set->body->nBuckets;
284          l++) {
285         if (*l != NULL) {
286             List b;
287
288             for (b = *l; b != NULL; b = b->tail) {
289                 Hash_Bucket hash_bucket = (Hash_Bucket) b->first;
290                 SetData set_data = (SetData) hash_bucket->data;
291                 ExitCode e;
292
293                 *atom = set_data->key_atom;
294
295                 if ((e = Execute_Code(code)) != Exit_OK)
296                     return e;
297             }
298         }
299     }
300
301     return Exit_OK;
302 }
```

Figure 4-3: Original implementation of `ForeachSet`

The other five errors reported are in `operator.c`. The first three are found in the body of `ForeachSet` (Figure 4-3).

This unwieldy code fragment obscures the essence of the code. It is iterating through the elements of `set`, and executing lines 293–296 for the data associated with each element in the set. The body of the loop is somewhat more complex than it appears, since the result of `Execute_Code` depends on the value of `*atom` which references global storage in the dictionary.

The surrounding code iterates through the elements of `set`. Because it is written using low-level details of the `Set` implementation, it is hard to read and understand. `Set` is implemented using a `Hash_Table`, so the code for iterating through its elements involves nested loops using the representations of both `Set` and `Hash_Table`.

It is no surprise that `Set` provides no operation that directly implements `ForeachSet`. This operation is not natural for a `Set` abstraction and depends on much external code not related to the `Set` type. Although it would eliminate the abstraction violations, moving this code to the `Set` module is unacceptable if we desire a well-organized modular program. However, the ability to iterate through the elements of a `Set` corresponds to an abstract notion that is often needed by clients. Since C provides no mechanisms for iteration abstraction, we have to resort to some other means for

```
277  static ExitCode ForeachSet(Set set, Name subject, Code code)
278  {
279    Atom *atom = Dict_Install(subject, NULL, DICTFLAGS_LOCAL);
280
281    Set_ElementValues(set, val)
282      {
283        ExitCode e;
284        *atom = val;
285
286        if ((e = Execute_Code(code)) != Exit_OK)
287           return e;
288      } end_Set_ElementValues
289
290    return Exit_OK;
291  }
```

Figure 4-4: Revised implementation of `ForeachSet`

abstracting iteration. In Section 3.2, an abstract iterator type was used to iterate through the elements of an `erc`. There are other means for emulating iteration abstraction in C including non-functional macros or providing an abstract operation that takes a `Set` and a function as arguments and applies the function to each element in the `Set`.

Thus, we write an abstract iterator, `Set_ElementValues` for iterating through each data value of a `Set`. `Set_ElementValues` is not a function — it instantiates a macro that iterates through the elements of its first argument, assigning its second argument to the current data value in the body of the loop. Here, `Set_ElementValues` is implemented so that the end of the loop is balanced with `end_Set_ElementValues`. Figure 4-4 shows how `ForeachSet` can be rewritten using the abstract iterator.

In most senses, this code is significantly easier to read and understand than the original code. No longer does someone reading `operator.c` have to guess or look at the implementation of `Set` to figure out what is going on here. Further, a programmer may reimplement `Set` without making any non-local changes — everything is localized to `Set.c` and `Set.h`. It does, however, conceal the looping nature of the construct and may confuse C programmers unfamiliar with stylized iterators.

Because the macro used to implement `Set_ElementValues` is not a function, it cannot be specified in LCL. It is expanded in-line and checked like a regular C macro. As a result, we need to use `/*@access Set*/` comments to allow access to the `Set` representation in the macro definition, and `/*@noaccess Set*/` to disallow access in the body of the loop. Section 5.2.3 discusses adding methods of specifying iterators to LCL to provide a better alternative.

This final two messages report problems similar to the first. Both are instances where an abstraction violation was needed because an abstract operation was not provided. We consider the messages in reverse order.

The last error reported concerns the code fragment:

```
578    String key = Pop_String();
579    Set set = Pop_Set();
580
581    Push(Atom_Boolean(Hash_Find(set->body,
582                                key->body,
583                                Hash_String(key->body)) != NULL));
```

It is not clear from the code that the argument to `Atom_Boolean` is testing if `key` is a member of `set`. The provided `Set` abstraction did not include any operation for testing membership. This is an operation one would expect an abstract `Set` type to provide, since it is part of our abstract notion of a `Set`. So, we add a `Set_isMember` operation, which takes a set and a key as arguments and returns a `Boolean`. Using `Set_isMember`, lines 581–583 can be rewritten in a more readable and appropriate way:

```
Push(Atom_Boolean(Set_isMember(set, key)))
```

The remaining message concerns a similar problem, except here we may have to compromise between efficiency and data abstraction. The code is,

```
529   Hash_Bucket bucket = Hash_Find(set->body,
530                                  key->body,
531                                  Hash_String(key->body));
532
533   if (bucket == NULL)
534     ymerror("Set does not contain an entry for \"%s\"", key->body);
535   Push(((SetData) bucket->data)->value_atom);
```

The code is like the previous excerpt, except that if a key is found it pushes the data associated with the key. It finds the `Hash_Bucket` in the body of `set` that is associated with the body of `key` using a hash value obtained from the body of `key` (lines 529–531). If the returned `Hash_Bucket` is a `NULL` pointer, it reports an error (lines 533–534), otherwise, the data field of the returned bucket is cast to a `SetData` type, and its `value_atom` field is pushed (line 535). This is an unpleasant fragment of code to have to deal with inside the `operator` module — it depends heavily on the representation of `Set`, as well as the underlying representation of `Hash_Bucket`.

We can replace the first part (lines 529-534) using the `Set_isMember` operation. We add a `Set_getData` operation to the `Set` abstraction that returns the data associated with a key in a `Set`. Then, the excerpt is rewritten as,

```
if (!(Set_isMember(set, key)))
   ymerror("Set does not contain an entry for \"%s\"", key->body);
Push(Set_getData(set, key));
```

Unlike all the earlier changes, this one results in an efficiency penalty. All the earlier abstraction violations were fixed with no more efficiency loss than the overhead of a function call. We hope (perhaps unrealistically) the compiler will be able to optimize this. If we are particularly concerned with performance, we can implement the function using a macro and eliminate any performance penalty.

In the last example, however, some performance is sacrificed for improved readability and maintainability. Before, only one search in the `Hash_Table` was necessary, since we could use the returned `Hash_Bucket` data to both test membership and add the appropriate value when it exists. Now, the calls to `Set_isMember` and `Set_getValue` each duplicate the same search. In most circumstances, this minor performance penalty is well worth the improvement in code readability and preservation of abstraction. If this were a particularly performance-critical section of code, however, it may be unacceptable. One possible solution would be to provide an abstract operation to test a set for membership and produce the appropriate error if the key is not in the set, or push the associated data. This would not belong in the `Set` abstraction, since it does not correspond to an abstract operation in a general context. Alternately, we could add a `Set_getBucket` operation to `Set` that returns the `Hash_Bucket` associated with a key in the set. If `Set_getBucket` were implemented by returning the result of the `Hash_Find` call, this would allow us to write the code excerpt abstractly with no efficiency loss. However, it would expose the representation of `Set`. We could prevent this exposure by copying the `Hash_Bucket`, but this would not be an efficiency improvement. Finally, we could to decide accept the abstraction violation. The dangers could be minimized, as long as it is well-documented and localized.

We can now run LCLint with no errors, ensuring the `Set` type is consistently abstract. Through the process, we have gained an understanding of the code, and replaced several awkward and difficult to read fragments with clearer alternatives. Since `Set` is now verified to be consistently used as an abstract type, a maintainer of the code knows that making changes to the `Set` implementation will not introduce problems elsewhere.

Hash

Many other modules in `quake` are also intended to implement abstract types. For most modules, the process is similar to that described for `Set`. We will look at converting the types in the `Hash` module since they illustrate some other benefits of LCLint, as well as difficulties involved in maintaining strict abstraction boundaries in a language like C.

There are three types associated with hash tables defined in `Hash.h`. We declare the first two to be mutable abstract types, and `HashValue` to be an immutable abstract type in `Hash.lcl`:

```
mutable type Hash_Table;
mutable type Hash_Bucket;
immutable type HashValue;
```

An initial run generates 44 messages. This is too many to tackle at one time. We could conclude that `Hash_Table` is not intended to be an abstract type, and make everything exposed. However, it seems like `Hash_Table` should be abstract, so instead

of rejecting this possibility, we try making each type abstract in turn to see where the problems lie.

Running LCLint with just Hash_Table declared as an abstract type yields twelve messages. Two of the messages concern the use of the primitive constant NULL where an abstract Hash_Table is expected:

```
dict.c:19,38: GlobalDictionary initialized to type void *, expects
                  Hash_Table: NULL
Name.c:13,27: Names initialized to type void *, expects Hash_Table: NULL
```

NULL is declared to have type void *, so it cannot be used where an abstract type is expected. This is a reasonable prohibition since allowing NULL to be used as an abstract type assumes that the abstract type is represented by a pointer and that NULL has some defined meaning. This is a dangerous, and often incorrect assumption. We can remedy this by creating a new constant, Null_Hash_Table. Its specification is added to Hash.lcl:

```
constant Hash_Table Null_Hash_Table;
```

It is implemented by a macro defining it to be NULL. We replace the NULLs that were used as Hash_Tables, with Null_Hash_Table.

Some of the remaining errors are the result of missing Hash_Table operations. Like Set, the Hash_Table module does not provide adequate operations for clients to use it abstractly. We add Hash_isEmpty, Hash_equal, and a Hash_entries iterator to the Hash_Table abstraction, and rewrite offending client code to use them.

This leaves four errors, all in dict.c:

```
dict.c:50,34: Cast to abstract type Hash_Table: (Dictionary)f->first
dict.c:70,30: Cast to abstract type Hash_Table: (Dictionary)f->first
dict.c:108,18: Function Hash_Destroy expects arg 1 to be Hash_Table gets
                   void *: List_Pop(&DictionaryStack)
dict.c:157,33: Cast to abstract type Hash_Table: (Dictionary)f->first
```

Three of the messages report casting errors, although they seem peculiar since the cast expression is (Dictionary), which is not declared as an abstract type. But Dictionary is typedefed to Hash_Table, so a cast involving Dictionary is as much of an abstraction violation as one involving Hash_Table.

Correcting the abstraction violation, however, is not easy. All the problems we have seen so far could be readily fixed by adding additional abstract operations or minor restructuring of the code; here we are faced with a more fundamental problem. Each of the errors involves using the List type. For the cast errors, f is a List, and we needed the cast to coerce the first field of the List, which is a void pointer, into a Dictionary. The other error involves a parameter type mismatch, as a result of a List operation returning a void pointer.

Because C does not support polymorphic or parameterized types, it is common practice to use `void` pointers to implement generic data structures. So, internal `List` operations accept and return `void` pointers. Clients of `List` are expected to keep track of the actual types of the `List` elements, and cast them to the appropriate type. I know of no elegant way to avoid the type abstraction violation. One solution is to make a separate list module for every abstract type that needs list operations — e.g., `DictionaryList`, for keeping a list of dictionaries. This is cumbersome and may involve writing substantial additional code, but it does preserve type abstractions. The best solution may be to accept the abstraction violation and leave the code as is. Although this is indeed a type violation, it is likely to be a harmless one. If `List` is indeed a faithful implementation of our notion of lists, it does not do anything to manipulate the actual elements. The burden of maintaining the type of the List is placed on the client, but it is hoped that well-named variables should keep errors to a minimum. To eliminate the error messages, we surround the offending code with `/*@access Hash_Table*/` and `/*@noaccess Hash_Table*/` control comments.

Now that `Hash_Table` is abstract, we can move on to trying to make `Hash_Bucket` abstract. Running LCLint with `Hash_Bucket` declared as a mutable abstract type yields the 26 messages. Some of these involve abstraction violations similar to those seen for `Hash_Table` including the use of `NULL` to initialize an abstract type and missing abstract operations.

After fixing the simple problems, seventeen errors remain. All report accessing the field of an abstract `Hash_Bucket`. Looking at the code generating these errors, most do not appear to be implementing abstract `Hash_Bucket` operations, but need access to the underlying data structure at a fundamental level. While each instance where the data field of a `Hash_Bucket` is manipulated directly could conceivably be coded to avoid this, in many instances it would involve considerable work and significant performance penalties to do so. At this point, we have to reconsider whether `Hash_Bucket` should be made an abstract type at all. Perhaps we could coerce it into an abstract type, but it seems clear this is not what the programmer originally intended. Instead, we declare it to be an exposed type.

It remains to make `HashValue` abstract. Four errors are reported after we add the declaration of `HashValue` as an immutable abstract type to the specification:

```
dict.c:52,7: Function Hash_Find expects arg 3 to be HashValue gets int:
                name->hash_value
dict.c:78,60: Function Hash_Find expects arg 3 to be HashValue gets int:
                 name->hash_value
dict.c:83,45: Function Hash_Insert expects arg 3 to be HashValue gets
                 int: name->hash_value
Name.c:30,2: Assignment of HashValue to int:
                name->hash_value = Hash_String(text)
```

At first, these errors seem puzzling — each involves a type mismatch with the `hash_value` field of a variable `name`. Further inspection reveals that all the `name` variables are declared to be type `Name`, defined by:

```
typedef struct Name {
    char *text;
    unsigned int hash_value;
} *Name;
```

The type of the `hash_value` field is an `unsigned int`, not a `HashValue`, as we would expect. In this implementation, the type of `HashValue` is also `unsigned int`. However, this naming inconsistency is dangerous, and could lead to bugs if the representation type of `HashValue` were changed. Here, LCLint has found a problem that would probably not be found by other means.

We are left with a `Hash_Table` abstraction that is not completely abstract — it includes an exposed `Hash_Bucket` type. There are operations provided which return `Hash_Buckets` that are contained in a `Hash_Table`, thereby exposing the representation of `Hash_Table`. Ideally, the only type exported by the `Hash_Table` would be an abstract `Hash_Table` type providing all necessary operations directly. If we were willing to invest in a major coding effort, we could replace the `Hash_Table` implementation with a truly abstract type. However, since our goal here is to understand the code and make it easier to maintain, learning that the type is not abstract is probably sufficient.

Additional Minimal Specifications

Several more types were made abstract in the same way, with similar results.

Making `Array` abstract uncovered 23 errors, one involving an initialization to `NULL` and the rest involving accessing fields of an `Array`. The initialization error was fixed by adding a `Null_Array` constant, as before with `Null_Hash_Table`. Closer inspection of the other errors revealed many omissions from the `Array` module — no `Array_Fetch`, `Array_Set`, or `Array_Size` operations were provided. Adding these operations to `Array` facilitated easy fixes for most of the errors reported. The remaining errors are fixed by adding an `Array_Elements` iterator. Replacing the direct Array manipulations outside the Array module with calls to the new abstract operations not only produces smaller and more readable client code, but makes it easier to consistently do appropriate run-time checks to ensure that array indexes are not out of bounds. This is easier and more reliable than having to scatter the checks throughout client code as was done previously.

Declaring the `Atom` type to be abstract revealed 86 abstraction violations, in many different source files. Many of these involved explicit tests of the tag field of the `Atom`, as well as direct manipulation of its contained data. Given the large number of errors and their nature, the declaration of `Atom` changed to make it an exposed type. Perhaps it would have been better to design the program with an abstract `Atom` type; however, it is clear the programmer did not intend for `Atom` to be treated as an abstract type.

Making `Dictionary` an abstract type served as a pleasant contrast — no errors were reported. Although this does not lead us to change any code, it does provide useful

information that could not be easily obtained without using LCLint. By running LCLint with `Dictionary` declared as an abstract type, we have certified that it is indeed abstract. This is helpful for understanding and maintaining the program.

Finally, `Code` was declared to be abstract. LCLint reported five errors, all regarding arrow accesses in the body of `Execute_Code` in `Execute.c`. We could add abstract functions to avoid the abstraction violations; however, it seems `Execute_Code` could be part of the `Code` module instead of the `Execute` module. So, we use control comments to allow it to access the representation of `Code`.

In this section, we have seen how LCLint can be used to declare a type to be abstract and detect and eliminate abstraction violations. Although it may seem that this is a critique of the code, the effectiveness of this approach depends on the code being well-designed and implemented in a style employing abstract types. Because the code was written with abstract types in mind, declaring types to be abstract typically uncovered only a few abstraction violations. The flaws, then, are not with the code, but with the methods and tools available when the code was developed. Since C does not provide abstract types, there is no way for a programmer to denote that a type is intended to be abstract and check that the abstraction is not violated. LCLint provides this ability, and hence the benefits associated with type encapsulation.

In the process of making types abstract, no implementation bugs producing incorrect behavior were found. This may be because `quake` had already been extensively tested and used, so it is likely not many bugs remain. However, running LCLint on `quake` and adding minimal specifications was still a worthwhile process. We gained an understanding of the code, and also made it easier to maintain in the future. By having LCL specifications declaring types to be abstract we now know which types can be safely modified in isolation and have a better idea what level of detail is needed to analyze the code. Further, by eliminating the abstraction violations discovered by LCLint, we made the client code shorter and easier to read, understand, and maintain.

4.3 Developing the Specifications

So far, our specifications have been limited to simple abstract type declarations. Much can be gained from these minimal specifications relating to error detection and code maintainability. However, without additional specifications we cannot benefit from certain other checks, including globals and modifies checking, and our specifications do not serve well as documentation.

In this section, the specifications for `Set`, `Hash_Table` and `Execute` are augmented by adding the prototype information. The process is incremental — as we add more specifications, we often uncover additional problems in earlier ones.

Set

As a first step to augmenting the specifications, the prototypes in `Set.h` are moved into `Set.lcl` and their terminating semicolons are replaced with empty specification

```
imports Array, Atom, Boolean, String, <stdio>;

mutable type Set;
immutable type SetData;

Set Set_New() { }
Set Set_InsertAtom(Set set, Atom key, Atom value) { }
void Set_Put(Set set, Atom key, Atom value) { }
SetData Set_Get(Set set, Atom key) { }
Atom Set_getValue (Set set, String key) { }

Boolean Set_isMember (Set set, String key) { }
Boolean Set_isEmpty (Set set) { }

Set Set_Convert(Atom atom) { }
Array Set_ToArray(Set set) { }
void Set_Dump(FILE *stream, Set set) { }
```

Figure 4-5: `Set.lcl` after including prototypes

bodies. `Set.h` is changed to include `Set.lh` where the prototypes were removed. `Set.lh` is generated automatically by LCLint from the prototypes now in `Set.lcl`.

Figure 4-5 shows the revised `Set.lcl`. The first line imports five other modules. The first four modules are specified as part of this system. We need to use the types defined in the `Array`, `Atom`, `Boolean` and `String` specifications in the prototypes for our functions. The fifth import, `<stdio>`, refers to a standard library. LCL libraries are provided to mirror the standard ANSI C libraries. Importing `<stdio>` provides the declaration of the `FILE` abstract type. The next two lines of the specification are the familiar type declarations from the original specification. The remainder are the function prototypes taken from `Set.h`. For now, each function header lists no globals and each body specification is empty. This implies that no global variables may be used, and no client-visible state may be modified. By using LCLint, we find where function implementations violate this specification, and amend the specifications accordingly.

Running LCLint with the new specification reports two errors:

```
Set.c:173,5: Called procedure fprintf may modify *stream:
                fprintf(stream, "{")
Set.c:175,5: Called procedure fprintf may modify *stream:
                fprintf(stream, "}")
```

Both are in the function `Set_Dump`. The function `fprint` is specified in the standard library by:

```
int fprintf (FILE *stream, char *format, ...) {
   modifies *stream;
}
```

Since `fprintf` may modify the `FILE` pointed to by its first argument, the calls in `Set_Dump` may modify the `FILE` pointed to by `stream`. Since `stream` is an argument

```
imports Boolean, <stdio>;

mutable type Hash_Table;
constant Hash_Table Null_Hash_Table;

immutable type HashValue;
HashValue Hash_String(char *string) { }

typedef struct Hash_Bucket {
  char *key;
  HashValue hash_value;
  void *data;
} *Hash_Bucket;

typedef void (*Hash_WalkProc)(Hash_Bucket bucket);

Hash_Table Hash_InitializeTable(Hash_Table t) { }
Hash_Table Hash_NewTable(int initial_buckets) { }
Boolean Hash_isEmpty (Hash_Table t) { }
Boolean Hash_equal (Hash_Table t1, Hash_Table t2) { }

Hash_Bucket Hash_Find(Hash_Table t, char *key, HashValue val) { }
Hash_Bucket Hash_Delete(Hash_Table t, char *key, HashValue val) { }
Hash_Bucket Hash_Insert(Hash_Table t, char *key, HashValue val) { }

void Hash_Walk(Hash_Table t, Hash_WalkProc proc) { }
void Hash_Destroy(Hash_Table t) { }
void Hash_DumpTable(FILE *stream, Hash_Table t) { }
```

Figure 4-6: `Hash.lcl` after including prototypes

to `Set_Dump`, this modification is visible to a client. Hence, it should be reflected by adding a modifies clause to the specification of `Set_Dump`:

```
void Set_Dump(FILE *stream, Set set) { modifies *stream; }
```

Now, no errors are reported by LCLint. We may wonder why no modification errors were reported for `Set_InsertAtom` or `Set_Put`. These sound like operations that modify their `Set` argument. In the implementation, though, the modifications are hidden in calls to `Hash_Insert`. Since we have not yet written prototype specifications for the `Hash_Table` module, LCLint does not propagate the modification in `Hash_Insert`. Hence, we see an instance where modifies checking is incomplete because of missing specifications (see Section 2.3.4).

Hash

As with `Set`, we augment the specification of `Hash` by moving prototypes and exposed type declarations from `Hash.h` to `Hash.lcl` (shown in Figure 4-6). The only unusual line is the `typedef` for `Hash_WalkProc`. This defines an exposed type `Hash_WalkProc`, that is a function with no return value taking a single `Hash_Bucket` argument. `Hash_WalkProc` is used as the type of a function argument to `Hash_Walk`.

Figure 4-7 shows the result of running LCLint using the `Hash.lcl` specification. All errors reported are for functions which we expect to modify their arguments. True to their names, `Hash_InitializeTable`, `Hash_Delete` and `Hash_Insert` modify their `Hash_Table` argument. As in `Set_Dump`, calls to `fprintf` in `Hash_DumpTable` generate modification errors. We can add the appropriate modifies clauses to reflect these modifications.

Surprisingly, no modification errors were reported for `Hash_Destroy`. If the implementation of `Hash_Destroy` used a standard memory free routine, a modification would be detected since the standard library specifies `free` to modify the value or its argument. Instead, it used `Utils_FreeMemory` which is not yet specified.

The case with `Hash_Walk` is less clear. Since it applies a higher-order parameter it may contain undetected modifications. Conceivably, the function argument may modify any argument to `Hash_Walk` and any global variable. Currently, LCLint is not designed to deal with higher-order functions effectively, so we disregard this possibility. (Section 5.2.3 discusses extending LCLint to handle higher-order functions.)

Adding the modifies clauses to the specifications of functions which reported modification errors eliminates all of the original errors reported. The modifies clauses propagate to clients of these functions, revealing more unspecified modifications:

```
Set.c: (in function Set_InsertAtom)
Set.c:51,2: Called procedure Hash_Insert may modify set->body:
  Hash_Insert(set->body, key_string->body, Hash_String(key_string->body))
Set.c:59,2: Called procedure Hash_Insert may modify set->body:
  Hash_Insert(set->body, key_string->body, Hash_String(key_string->body))
Set.c: (in function Set_Put)
Set.c:137,2: Called procedure Hash_Insert may modify set->body:
  Hash_Insert(set->body, key_string->body, Hash_String(key_string->body))
Set.c:146,2: Called procedure Hash_Insert may modify set->body:
  Hash_Insert(set->body, key_string->body, Hash_String(key_string->body))
```

The expected modifications for `Set_InsertAtom` and `Set_Put` are now detected, through the specification of `Hash_Insert`. Adding the appropriate modifies clauses to the specifications of `Set_InsertAtom` and `Set_Put` corrects the inconsistencies.

Execute

For a final example, we write prototype specifications for the `Execute` module. Unlike the other modules we have considered, `Execute` does not implement an abstract type. It does, however, export global variables. The specification for `Execute` is shown in Figure 4-8. The lines after the imports clause declare five global variables.

Running LCLint produces 46 messages, all regarding the use and modification of the global variables declared in `Execute.lcl`. To reduce the number of messages, the declarations of all globals except `Execute_CurrentStream` are moved back to `Execute.h`. Now, we can focus on messages relating to `Execute_CurrentStream`.

```
% lclint  +showfunc +charint +voidabstract Array.c ... utils.c \
   Array.lcl Atom.lcl ... Set.lcl Hash.lcl
LCLint 1.2 --- 07 May 94

Hash.c: (in function Hash_InitializeTable)
Hash.c:18,5: Suspect modification of table->nEntries:
                table->nEntries = 0
Hash.c: (in function Hash_Delete)
Hash.c:126,3: Suspect modification of table->buckets[?]:
                  table->buckets[bucket_index] = temp->tail
Hash.c:128,6: Suspect modification of table->nEntries: table->nEntries--
Hash.c: (in function Hash_Insert)
Hash.c:188,7: Suspect modification of table->buckets[?]:
                  table->buckets[i] = table->buckets[i]->tail
Hash.c:189,7: Suspect modification of table->buckets[?]->tail through
    alias new_buckets[new_index]->tail: new_buckets[new_index]->tail =
    new_head
Hash.c:193,2: Suspect modification of table->nBuckets:
                  table->nBuckets = new_length
Hash.c:196,2: Suspect modification of table->buckets:
                  table->buckets = new_buckets
Hash.c:204,2: Suspect modification of table->nEntries: table->nEntries++
Hash.c: (in function Hash_DumpTable)
Hash.c:252,5: Called procedure fprintf may modify *stream:
                  fprintf(stream, "%d buckets\n", table->nBuckets)
Hash.c:253,5: Called procedure fprintf may modify *stream:
                  fprintf(stream, "%d entries\n", table->nEntries)
Hash.c:266,6: Called procedure fprintf may modify *stream:
                  fprintf(stream, "\t[%d]:", i)
Hash.c:271,3: Called procedure fprintf may modify *stream:
                  fprintf(stream, " \"%s\"", b->key)
Hash.c:274,6: Called procedure fprintf may modify *stream:
                  fprintf(stream, "\n")

Finished LCLint checking --- 13 code errors found
```

Figure 4-7: Modification errors reported using `Hash.lcl`

```
imports basic, Name, String, < stdio > ;

int Execute_CurrentLineNumber;
FILE *Execute_CurrentStream;
Name Execute_LastName;
String Execute_CurrentFileName, Execute_CurrentPathPrefix;

void Execute_Initialize(void) { }
void Execute_PushContext(void) { }
void Execute_PopContext(void) { }

ExitCode Execute_Atom(Atom atom) { }
ExitCode Execute_Code(Code code) { }
ExitCode Execute_Stream(FILE *stream, String stream_name) { }
ExitCode Execute_File(String path_prefix, String file_name) { }
```

Figure 4-8: `Execute.lcl`

Figure 4-9 shows the output of LCLint. Although the number of errors is large, all of them are in `yylex` in `lexical.c` or in `Execute.c`. We eliminate the error messages and improve the interface documentation by adding `Execute_CurrentStream` to the globals lists (and, where appropriate, to the modifies clauses) of these functions. The only message not due to `Execute_CurrentStream` is the use of `errno` reported in `Execute_File`. The global variable `errno` is declared in the standard library, and used by library functions to return error conditions to the caller. It is checked like a user-specified global, and the inconsistency is eliminated by adding `errno` to the globals list of `Execute_File`.

As with modifies clauses, listing new globals leads to detection of additional errors through propagation of global usage and modification information. Three errors are detected in `Execute_Code` because of calls to `Execute_PushContext` and `Execute_PopContext`.

Note that no globals or modifies checking is done in functions that are not specified (see Section 2.3.4). Adding the globals lists to function specifications improves the documentation of the interface, but does not provide a complete description of where a global variable is used. Running LCLint with the `-globunspec` flag detects eighteen instances where `Execute_CurrentStream` is used or modified in a function that is not specified. Most of these functions are not exported, so it would be a mistake to simply add specifications for them to document the global use. At present, there is no good solution for this problem.

4.4 Summary

In this chapter, we have seen how LCLint can be used to understand a program that had no specifications, and in the process document its interfaces, make it easier to maintain, and detect bugs. Instead of starting with a specification and implemen-

```
lclint +showfunc +charint +voidabstract Array.c ... utils.c
   Array.lcl ... Execute.lcl
LCLint 1.2 --- 09 May 94

Execute.c: (in function Execute_PushContext)
Execute.c:42,43: Unauthorized use of global Execute_CurrentStream
Execute.c: (in function Execute_PopContext)
Execute.c:54,5: Unauthorized use of global Execute_CurrentStream
Execute.c:54,5: Suspect modification of Execute_CurrentStream:
                  Execute_CurrentStream = context->stream
Execute.c: (in function Execute_File)
Execute.c:181,82: Unauthorized use of global errno
Execute.c: (in function Execute_Initialize)
Execute.c:193,5: Unauthorized use of global Execute_CurrentStream
Execute.c:193,5: Suspect modification of Execute_CurrentStream:
                   Execute_CurrentStream = NULL
lexical.c: (in function yylex)
lexical.c:369,26: Unauthorized use of global Execute_CurrentStream
lexical.c:369,21: Called procedure getc may modify
    *Execute_CurrentStream: getc(Execute_CurrentStream)
lexical.c:380,27: Unauthorized use of global Execute_CurrentStream
lexical.c:380,22: Called procedure getc may modify
    *Execute_CurrentStream: getc(Execute_CurrentStream)
lexical.c:385,30: Unauthorized use of global Execute_CurrentStream
lexical.c:385,25: Called procedure getc may modify
    *Execute_CurrentStream: getc(Execute_CurrentStream)
lexical.c:391,28: Unauthorized use of global Execute_CurrentStream
lexical.c:391,23: Called procedure getc may modify
    *Execute_CurrentStream: getc(Execute_CurrentStream)
lexical.c:394,31: Unauthorized use of global Execute_CurrentStream
lexical.c:394,26: Called procedure getc may modify
    *Execute_CurrentStream: getc(Execute_CurrentStream)
lexical.c:399,134: Unauthorized use of global Execute_CurrentStream
lexical.c:399,114: Called procedure ungetc may modify
    *Execute_CurrentStream: ungetc(CurrentChar, Execute_CurrentStream)
lexical.c:406,136: Unauthorized use of global Execute_CurrentStream
lexical.c:406,116: Called procedure ungetc may modify
    *Execute_CurrentStream: ungetc(CurrentChar, Execute_CurrentStream)

Finished LCLint checking --- 20 code errors found
```

Figure 4-9: Checking globals using `Execute.lcl`

tation as was done in Chapter 3, we started with an undocumented implementation and derived a specification. The automated checks between our hypothetical specifications and the actual source code verify that the specification is reasonable, and may discover problems in the source code. In practice, a combination of the two approaches is useful when using LCLint to develop new systems. Some modules may be well-understood and specified before they are implemented. Then, LCLint is used to check the source code against the specification. Other modules may be less fully specified, leading to a more deductive process. The result is the same: a specified program where the source code has been checked against the specification. This method of software development increases our confidence in both the code and specification, without the heavy burden associated with program verification.

None of the specifications written in this chapter are adequate client-level interface documentation. To complete the specifications, we need to add requires and ensures clauses and write LSL traits for the underlying types. Since LCLint does not use any of this information to perform checks on the source code, it is outside the scope of this thesis.

Chapter 5

Conclusions

LCLint confirmed the claim that specifications can be effectively used to do simple static checks on source code. Experience has shown LCLint to be a useful pragmatic tool, although it only touches the surface of what can be done using specifications to check source code.

The primary motivation for developing LCLint was as a tool for detecting bugs. Not enough experience has been had using LCLint as code is being developed to establish its effectiveness in finding bugs. Since most of the experience with LCLint has been with well-tested systems, it is not surprising that most of the messages report violations of data abstractions and style conventions rather than bugs. Our experience has shown LCLint to be useful in improving code quality, supporting a programming methodology employing data abstraction and detecting flaws in specifications. It found a few code bugs that could not be detected without using specifications in programs that had already been tested, but so far has been more useful in validating abstraction barriers and detecting violations of style conventions.

Towards the end of LCLint development, I began using LCLint on its own code and specifications. Many errors were caught, most involving type abstraction violations. Most of the actual bugs detected by LCLint were not related to specification checking, although the flexibility and strict type-checking provided by LCLint discovered errors that would not have found with traditional lint. There was one instance where a function was declared to return the wrong abstract type that was detected by LCLint. Many problems were also detected regarding the misuse of macro parameters. Only minimal specifications were written, so no modifies or globals checking was done. Since most of the code had already been tested extensively before LCLint was robust and stable enough to be used on its own source code, many bugs that could have been detected by LCLint had already been found through testing and corrected.

I derived the most benefit from LCLint when, relatively late in the development, I decided that the underlying implementation for representing types was too inefficient. Without LCLint, I would have been reluctant to reimplement such a pervasive type for fear that unexpected dependencies on the previous implementation would lead to difficult to detect bugs. By using LCLint, however, I could verify that the type

was truly abstract, and change its implementation without concerns that it might introduce bugs elsewhere.

5.1 Design Goals

The design goals for LCLint were efficiency, ease of learning, incremental gain and flexibility (see Section 1.1). Most of these were met with at least partial success.

5.1.1 Efficiency

Although early versions of LCLint did not perform as well as a compiler for code sizes over a thousand lines, reimplementing some abstractions led to major efficiency improvements. The current version runs in time comparable to a typical compiler.

To enable running LCLint on large systems with long specifications, a library facility is provided. Specifications can be processed once and stored in a condensed format. The library can be quickly loaded, allowing an individual source file to be processed efficiently. By controlling system builds with `make` (see Appendix A.3), we can ensure that only files that have changed are rechecked. If external interfaces are specified, we can depend on changes in a source file having no effect on other files. When a specification is changed, we need to rebuild the library and check all source files that use this module. Care was taken in designing the library encoding so that large libraries can be loaded quickly.

Some statistics for running LCLint on three systems of varying size are shown in Figure 5-1. As expected, the execution time of LCLint is approximately linear in the size of the code. Time to process specifications for a fully specified program is long compared to the time to do the C checks, but for large systems we can build a library file encoding their specifications. Since we expect the specifications will not change as frequently as the code during development, it is more essential that source code processing is quick when a specification library is loaded. For the `dbase` example, using a library reduces checking time by more than fourfold.

The difference between the lines of source code and the actual number of lines processed by LCLint is due to the nature of file inclusion in C. Since many header files are included by more than one source file, the actual number of lines processed greatly exceeds the number of source lines. The `+singleinclude` flag optimizes file inclusion by eliminating some reprocessing for header files that are included more than once. This is not the default, since it may cause incorrect checking if the same identifier is declared in two different header files with different types.

Although the statistics in Figure 5-1 show that LCLint successfully runs faster than a typical compiler in processing complete programs, what is more important is how quickly we can check a single file that is part of a larger system. When code is developed, typically a few source files will be edited at a time, so the time needed to recompile changed source files and link a new executable is of prime importance. If LCLint is to be used in the development process, it should not increase this time

Program	lines spec	source	processed	compile (sec)	LCLint (sec)
`dbase` (Chapter 3)					
complete system	307	997	3 113	3.9	8.7
making spec library	307				7.3
checking source using library		997	3 113		2.0
`quake` (Chapter 4)					
no specifications		6404	75 995	19.7	19.8
final version	149	6812	78 329	21.4	28.7
making spec library	149				6.0
checking source using library		6812	78 329		21.4
using `+singleinclude`		6812	26 675		7.8
LCLint sources					
making spec library	362				3.6
sources using `+singleinclude`		65 695	217 257	236.2	139.2

Times give are sum of user and system time, measured as the median of five trials on a DEC 3000 AXP 500. Source line counts include `.c`, `.h` and `.lh` files. Machine generated source files are included in the line counts and checked for completeness. Compilation is done using `gcc` with no optimization or warning flags.

Figure 5-1: Statistics for running LCLint on entire programs

file	lines	lines processed	plain build	LCLint build	increase
`globals.c`	3	7	3.8	4.3	12%
`udnode.c`	147	16 693	6.6	15.9	141%
`llsymtab.c`	534	17 714	6.8	15.8	132%
`ctbase.c`	1783	20 221	8.2	19.1	133%
`abstract.c`	4784	26 752	11.8	27.6	134%

As in a realistic development environment, the compilations were run using a makefile. Build time is total time taken to rebuild the system (including linking) when the given source file is changed. LCLint build time is the total time taken to run LCLint on the changed file using the specification, compile the file and link the program. All times are in seconds.

Figure 5-2: Statistics for running LCLint on single source files

unacceptably. Figure 5-2 shows statistics comparing the time to compile one source file from the LCLint source and relink the binary, with the time taken running LCLint on the source file using the specification library. On average, LCLint more than doubles the build time. This is high, primarily because of the number of lines included through header files. Because of file inclusion, to check a small source file may involve processing tens of thousands of lines. This could be somewhat improved by structuring header files differently. While more than doubling the build time is certainly not a negligible cost, I feel that the checking benefits are enough to merit this time penalty. These results could be substantially improved by recoding portions of LCLint.

5.1.2 Flexibility

Balancing the needs for flexibility with usability often leads to compromises. As systems become more flexible, they tend to become harder to use since users need to know more to customize the tool for their purposes. LCLint provides forty flags for controlling which checks are done and which types are considered distinct, and additional flags for controlling the cosmetic appearance of messages and high-level behavior. There are too many flags for anyone to remember, although mnemonic names mean a few commonly used flags are usually remembered. Despite this, certain classes of messages cannot be suppressed without suppressing desirable messages also. While there are flags for turning off checking of ignored return values of any type, and ignored return values that are `int` or `bool` only, there is no way to suppress these messages for some other type without suppressing them for all other types. We can turn on checking to disallow pointer arithmetic, but cannot allow it for certain types of pointers and disallow it for others. It would be useful to have more flexibility than the current flags provide — however, doing so would add considerably to the complexity of LCLint, both for the user and the implementor.

Originally, we were reluctant to introduce stylized comments to control checking at a local level. This has the major drawback, that someone running LCLint from the command line may not see errors they are interested in since control comments in the source code suppress the messages. This is especially worrisome in the case where a type abstraction violation is concealed by a control comment, and a programmer is checking to see if the abstraction can be changed safely. However, being able to control checking at a local level allows for added flexibility in suppressing specific messages at specific code points. It might be worth adding an option to override control comments from the command line, but at present it is up to the programmer to use them judiciously.

5.1.3 Incremental Effort and Gain

Much is gained by using LCLint without any specifications. In addition to the traditional lint checks done by LCLint, many bugs were detected as a result of introducing a true `bool` type. Other C checks which proved particularly useful are those report-

ing cases in `switch` statements which fall-through to the next case and undefined variable usage errors.

The most significant benefits, though, result from using LCLint to check abstract types. Only a trivial LCL specification is necessary to detect abstraction violations and make the code easier to understand and maintain.

Writing more complete specifications can occasionally uncover additional problems, but it is hard to justify solely for this reason. At present, the primary benefit of extending specifications beyond type declarations is improved client-level documentation. The modifies and globals checking done by LCLint are useful in improving this documentation, and may occasionally find bugs. For most programs, though, the chances of finding significant code bugs are not high enough to justify the extra effort solely for this reason.

5.1.4 Easy to Learn and Use

Not enough experience has been gained introducing new users to LCLint to make definitive statements about how easy it is to learn. My intuition is that for programmers already accustomed to using abstract types only a little extra effort is needed to write minimal LCL specifications and gain significant benefits from using LCLint. Programmers who are not accustomed to using abstract types, may receive some benefits from using LCLint, but are unlikely to benefit considerably without adopting a programming style employing abstract types. We hope the availability of LCLint will encourage C programmers who did not program in a style employing data abstraction or formal specifications to adopt a more modular style and begin to use abstract types and write specifications. As yet, this is unsubstantiated.

Unfortunately, there are some significant barriers to learning to use LCLint effectively. Foremost is the difficulty of using many flags to customize checking to a particular coding style. Although the defaults have been chosen carefully to coincide with what we consider good programming style, individual programming styles vary widely, and most C programmers do not adopt some of the default conventions that are checked by LCLint. Without knowledge of LCLint's flags, new users are likely to be flooded with messages. This quickly leads to frustration if the problems reported are not considered important by the programmer, such as comparisons between `char` and `int` values. If the programmer does not realize there is a flag for suppressing these messages, it is likely that LCLint will be quickly abandoned.

Several approaches have been attempted to ameliorate this problem. On-line help provides some assistance — running LCLint with no flags, or with the `-help` flag will produce some general help information and a list of available flags. Further information on particular flags is given when they are listed after `-help` on the command line. This is useful to experienced LCLint users who forget the name of a flag or its precise definition. It is less useful to novice users who have no idea what types of flags are available or how particular checks can be controlled.

A more effective solution is provided by modes. Instead of needing to set each specific

flag, a mode can be used to set many flags to pre-defined values. For instance, using `-weak` turns off checks likely to irritate seasoned C programmers. This is useful for providing a quick way for new programmers to begin using LCLint, but it also reduces LCLint's effectiveness by turning off some checks that could detect bugs. The `-strict` mode is provided for doing much stricter checking than normal. In practice, this is rarely used. Programmers would have to be exceedingly careful to avoid being flooded with messages when `-strict` is used.

Perhaps the most important factor in making LCLint attractive to new users, is correct settings of the defaults. There are many tradeoffs involved — if the default does too much checking, novice users will be overwhelmed with messages when they first run LCLint; if it does too little, important errors may go unreported. Since programming styles vary widely, it is impossible to develop a default setting that works for all code. The existing default settings are based largely on my own experience and intuition, as well as some helpful feedback from a few users, none of whom could be described as typical C programmers. Clearly, much more experience is needed to develop satisfactory default settings.

5.2 Extensions

The current implementation of LCLint suggests many other possibilities for using specifications to check source code. The most interesting extensions involve extracting more information from the specification to improve checking, and augmenting the specification language with constructs that provide addition opportunities for checking as well as improving the documentation of the interface.

5.2.1 Improvements

The range of checks which can be done on C code is virtually unlimited. There are many common C errors which could be detected statically but are not detected by LCLint. Many of these have been investigated by other static checking tools since they do not rely on specifications.

A more relevant improvement would be checking that abstract representations are not exposed indirectly. Exposing the representation is a common problem in implementations of abstract types, and it often leads to serious bugs that are hard to detect. LCLint could attempt to check that no abstract operation returns a value which references a mutable part of an abstract representation or returns with parameters aliasing mutable parts of the abstract representation. Using the alias analysis already done by LCLint, it would not involve much additional code to check for simple cases of representation exposure.

Several changes could be made to the interface to LCLint to make it easier to use. Most of the difficulty in using LCLint is a result of large numbers of messages generated. While the existing command-line options and control comments provide mechanisms for suppressing messages, they are not sophisticated enough to be satisfactory. The

`-limit` flag provides a coarse means for suppressing similar messages, but it only works for consecutive messages that are similar in a simple textual way. Better ways to suppress multiple messages would greatly improve LCLint's usability. Ideally, we would like to be able to detect and suppress messages that are similar in content to some threshold number of previous messages.

Another improvement would be providing optional hints to the user on controlling messages. If many errors are detected involving incompatible `bool` and `int` usage, it would be helpful to a novice user if LCLint prints a message suggesting the `+boolint` option be used.

5.2.2 Using More of the Specification

The only parts of a function specification presently used by LCLint are the header and the modifies clause. No checking is done relating the source code to other parts of the specification.

The ensures clause contains the most specific information constraining the implementation of a function. LCLint foregoes many opportunities for checking by not detecting problems related to the ensures clause. Ultimately, program verification could be done using the ensures clause for a fully specified function. This would contradict our goal of making LCLint a simple, inexpensive tool. Instead, many checks short of full program verification could be done within our efficiency guidelines.

Some of the ideas from Aspect [Jac92] could be used, with similar benefits in LCLint. Consider a typical specification, such as this one from Figure 1-1:

```
bool intSet_choose (intSet s, out int *choice) {
   modifies *choice;
   ensures if (result) then (*choice)'∈ s^
           else size(s^) = 0;
}
```

LCLint's checking is limited to type checking, checking the `out` parameter is not used before it is defined, and checking that no visible state other than `*choice` is modified. But the specification contains much more information that would be useful in detecting errors. It constrains `intSet_choose` to return `FALSE` only when the set is empty; and if it returns `TRUE`, the value of `*choice` in the post-state should be an element of the set. Checking this completely would be well outside the acceptable efficiency of LCLint, and would most likely require guidance from the programmer to direct a proof. But LCLint could check that if `TRUE` is returned, `*choice` has been set to a value that depends on `s`. Checking that the return value is correct would be more complicated. There is no direct way to relate the semantics of the `size` operator to the concrete representation of an `intSet`.

Information in the ensures clause could also be used to improve alias analysis involving function calls. Instead of assuming values returned by functions do not alias any other storage, we could check if the function ensures the result is `fresh`. If it does, we know it cannot alias any visible state. If it does not, we could assume it

may alias any state visible to the called function or attempt a deeper analysis of the ensures clause to determine possible aliases. Further checking can be done on the implementation to verify that the return value satisfies alias conditions in the specification. It would be easy enough to determine if the result references fresh storage, though other conditions may be harder to check.

LCLint could also attempt to verify that the implementation is consistent with the checks clause. The checks clause constrains the implementation to test a condition and issue an error message if it is not met. Although we cannot easily verify that the condition in the checks clause is tested, we can check that some condition dependent on values used in the checks clause is tested and some branch determined by the test produces an error message. LCLint could issue an error message when it is clear that an implementation does not do the test specified by its checks clause. These may be helpful in detecting problems where an implementation lacks a test of necessary assumptions.

It might also be interesting to investigate whether any simple checking can be done at point of call involving requires clauses. Perhaps some of the typestate [SH83] concepts could be used to detect possible violations of requires clauses. I suspect, however, that no useful checking of requires clauses could be done without abandoning the simplicity of LCLint.

5.2.3 Augmenting the Specification Language

In addition to using more information in specifications written in the existing specification language, additional bugs could be detected if LCL were augmented to allow more comprehensive specifications and provide more constraints on the implementation and use of a function.

Iterators

Chapters 3 and 4 each include instances where it would be helpful to have an abstract operation for iterating through the elements of an abstract collection type. LCL provides no way to specify an iterator. We could add syntax to LCL for specifying iterators. However, since C does not support user-defined iterators, we would also need to adopt conventions on how iterators are used and implemented. LCLint could enforce these conventions in checking the implementation of the iterator and its use.

Higher-Order Functions

The current version of LCL has no way to describe higher-order functions. We can declare parameters and global variables that are functions using C's type syntax, but cannot write specifications for higher-order functions. Ideally, specifiers should be able to specify a higher-order function using the specification of the argument function. In Section 4.3, we saw one instance where a higher-order specification would be useful. We would like to express that if the function passed to `Hash_Walk` modifies

its argument, then `Hash_Walk` modifies the `Hash_Table`. Further, any globals used or modified by the argument function are used or modified by the called function. Current LCL syntax has no way of expressing these constraints. The difficulty associated with handling higher-order functions may outweigh the benefits, especially given that most C programs rarely use function parameters.

Variable Classes

Currently, the only distinction between variables of the same type made by LCLint is that between `out` parameters and regular parameters. The effectiveness of `out` parameter checking suggests adding more variable subclasses. In particular, significant gains could be made if more descriptive pointer declarations were employed. Traditionally, pointer arguments to C functions serve several purposes — they may be indirections to improve efficiency of the function call; they may be used to simulate multiple return values, intended only as an address where the called function should place a returned value; they may be used to simulate pass by reference; or, they may be pointers into a block of storage. The `out` type qualifier distinguishes the case where the parameter is intended as an address for a return value, but LCL provides no syntax for distinguishing between other uses of pointers. Additional qualifiers would enhance the specification's role as documentation, in addition to providing opportunities for additional LCLint checking.

Post-Obligations

The idea of post-obligations as used in Inscape [Per89] provides some interesting possibilities for improved checking. A simple post-obligation could be expressed as a type qualifier on a return value. For instance, we could declare a function that returns a value that may not be modified by the caller, or a pointer to storage that must be freed at some point. To check post-obligations, we need to propagate the obligations through specifications in the same way globals usage and modifies clauses are propagated. This would add some complexity to LCLint, but is likely to have major benefits in improving error detection.

5.3 Summary

The future of using specifications to check source code seems promising. The power of checkers that do not use specifications is limited to detecting anomalies in the source code. Program verifiers require complete specifications and programmer directed proofs that are too expensive for nearly all applications. Experience with LCLint has shown that small partial specifications can be combined with an efficient and flexible tool to detect classes of bugs which could not be found without specifications. Further, the process of writing specifications and using them to check source code is helpful in understanding and maintaining programs.

Appendix A

User's Guide

This appendix is extracted from the *LCLint User's Guide*, Version 1.2. Only those sections of the user's guide containing information not presented in the body of this thesis are included here. The latest version of the complete user's guide is available electronically (see Section A.7 for directions).

A.1 Type Access

Where code may manipulate the representation of an abstract type, we say the code has *access* to that type. There are three ways to control access to abstract types in LCLint:

- Specifications — if function *f* is specified in the LCL file that declares abstract type *t*, then the implementation of *f* has access to *t*.
- File name conventions — if abstract type *t* is declared in *t*`.lcl`, then all functions in *t*`.c` have access to *t*. (The `-accessunspec` flag overrides this convention, so that only functions that are specified will have access to the abstract type.)
- Control comments — `/*@access` *t*`*/` in the source code allows succeeding code to access the representation of *t*. Similarly, `/*@noaccess` *t*`*/` makes *t* abstract. Both `access` and `noaccess` may be given a list of types separated by spaces. Type access applies from the point of the comment to the end of the file or the next access control comment for this type.

Functions implemented in *t*`.c` will have access to all types declared in *t*`.lcl`, so it is possible to have access to several abstract types in the same module.

A.2 Libraries

To run LCLint efficiently on large systems, mechanisms are provided for creating libraries containing necessary information. This means source files can be checked independently, after a library has been created. The command line option `-dump` *library* stores information in the file *library* (the default extension, `.lldmp`, is added). Then, `-load` *library* loads the library. (See Appendix A.3 for an example makefile for using libraries.)

The LCLint dump file, `stdlibs.lldmp` contains the symbolic state after processing all the standard libraries. Unless `-load` is used to load some other library, or `-nolib` is used to prevent any library from being loaded, the standard library is loaded every time LCLint is run.

A.3 Make

For large systems, LCLint can be used more effectively when driven from a makefile. To support the use of LCLint in makefiles, LCLint returns exit status codes. Checking is successful when the number of actual errors matches number expected (none, unless expected number was set using `-expect`.) If checking is successful the exit status is `0`. If it is unsuccessful, the exit status is `1`.

Figure A-1 shows an example makefile for driving LCLint. When a source file is changed, it is checked by LCLint. When a specification is changed, we need to remake the library, and recheck all the source files against this new specification.

A.4 Emacs

If you use emacs to edit your source code, it can be beneficial to run LCLint directly from emacs. The LCLint release includes `lclint.elc` which defined an emacs command, `M-x lclint`, for running lclint. The new emacs command is similar to `M-x compile`, except it jumps to the exact column location of the error message, instead of the beginning of the line.

After typing `M-x lclint`, you will be prompted for a compile command. Enter the command identically to the command that would be used to run LCLint from the command line. If errors are found, `M-x next-lclint-error` jumps to the point where the next error was found.

A.5 Control Flags

LCLint provides many flags, in the hopes of supporting various programming styles and degrees of checking. In addition, modes are provided for setting many flags at

```
SPECS = # list .lcl files
LHS   = # list derived .lh files
SRC   = # list .c files
OBJ   = # list derived .o files

LCLINT = lclint # command to invoke lclint
LCLINT_FLAGS = -paramuse -returnvalint +charint

all : $(OBJ)
        $(CC) -o test $(OBJ)

check: lib.lldmp $(SRC) $(LHS)
        $(LCLINT) $(LCLINT_FLAGS) $(SRC) -load lib

lib.lldmp : $(SPECS)
        $(LCLINT) $(LCLINT_FLAGS) $(SPECS) -dump lib
        $(MAKE) check

.c.o: lib.lldmp $(LHS)
        $(LCLINT) $(LCLINT_FLAGS) -load lib $*.c
        $(CC) -c $*.c

.lcl.lh:
        $(LCLINT) -quiet $*.lcl

clean:
        rm -r lib.lldmp *.o *.lcs $(LHS) test

### list dependencies between specs here, e.g. if
### spec1.lcl imports spec2, we would write:

spec1.lcs: spec2.lcs
```

Figure A-1: Sample makefile

once. Individual message flags override the setting in the mode. Flags listed before the mode have no effect.

Flags can be preceded by + or −. When a flag is preceded by + it is *on*; when it is preceded by − it is *off*. The precise meaning of on and off depends on the type of flag. The +/− flag settings are clear and concise, but it is easy to accidentally use the wrong one. For this reason, LCLint issues warnings when a user redundantly sets a flag to the value it already had (unless `-warnflags` is used to suppress these warnings).

Flags can be set at the command line, to apply to all files checked. Some flags can also be set locally, using stylized comments. At any point in a file, a control comment can set the flags locally to override the command line settings. The original flag settings are restored before processing the next file. The syntax for setting flags in control comments is the same as that of the command line, except that flags may also be preceded by = to restore their setting to the original command-line value. For instance,

```
/*@ +boolint -modifies =charint */
```

makes `bool` and `int` indistinguishable types, turns off modifies checking, and restores the equivalence of `char` and `int` to its command line or default setting.

Flags can be grouped into three major functional categories: general flags, for controlling high level behavior; type equivalence flags for denoting particular types as equivalent or distinct; and message control flags for selecting which messages appear. General flags are applicable only at the command line; all other flags may be used both at the command line and in control comments.

General Flags

These flags have the same meaning when used with either + or −. They control initializations, message printing, and other behavior not related to specific checks.

`help` — on-line help
`dump` *file* — dump state to *file* (default extension `.lldmp`)
`load` *file* — load state from *file* (instead of standard library file)
`nolib` — do not load standard library
`whichlib` — show pathname and creation information for standard library
`i` *file* — set LCL initialization file
`I`*directory* — add *directory* to C include path
`S`*directory* — add *directory* to search path for LCL specs
`tmpdir` *dir* — set directory for writing temporary files
`showfunc` — show name of function containing error (first error in function only)
`singleinclude` — optimize include files
`stats` — display information on number of lines processed and execution time
`nolh` — suppress generation of `.lh` files
`quiet` — suppress herald and error count

`expect` *n* — set expected number of code errors (default 0)
`limit` *n* — suppress consecutive similar messages over limit parameter
`linelen` *n* — set length of messages in characters (default 80)

The following flags have difference meaning if + or − is used. The default behavior is on, described below. Using −*flag* has the opposite effect.

`warnflags` — warn when command line sets flag to default value in mode
`showcolumn` — show column number where error is found
`accessunspec` — representations of abstract types are accessible in unspecified function in the `.c` file with the same name as the specification (see Section A.1)

Type Equivalence Flags

Using +*flag* makes the named types indistinguishable; using −*flag* makes them distinct.

`boolint` — `bool` and `int` are equivalent
`charindex` — `char` can be used to index arrays
`charint` — `char` and `int` are equivalent
`enumint` — `enum` and `int` are equivalent
`forwarddecl` — forward struct and union declarations of pointers to abstract representation match the abstract type
`numliteral` — `int` literals can be floats
`voidabstract` — `void *` matches pointers to abstract types (dangerous)
`zeroptr` — `0` can be treated as a pointer

Message Control Flags

Message control flags are preceded by a − to turn the message off, or a + to turn the message on. Each flag is described by the class of messages that are reported when it is on, and suppressed when it is off.

Globals and Modifies Checking

`globals` — unspecified use of global variable
`globunspec` — use of global in unspecified function
`globuse` — global listed for a function not used
`modifies` — unspecified modification of caller-visible state
`modunspec` — modification in unspecified function
`stdio` — use/modification of standard streams (`stdio`, `stdout`, `stderr`)

Declarations

`topuse` — declaration at top level not used
`paramuse` — function parameter not used
`varuse` — variable declared but not used
`fcnuse` — function declared but not used
`exportvar` — variable exported but not specified
`exportfcn` — function exported but not specified
`exporttype` — type definition exported but not specified
`overload` — library function overloaded
`incondefs` — function or variable redefined with inconsistent type

Type Checking

`bool` — representation of `bool` is exposed
`pred` — type of condition test (for `if`, `while` or `for`) not boolean
`predptr` — type of condition test not boolean or pointer
`ptrarith` — arithmetic involving pointer and integer
`ptrcompare` — comparison between pointer and number
`strictops` — primitive operation does not type check strictly

Return Values

`returnval` — return value ignored
`returnvalbool` — return value of type `bool` ignored
`returnvalint` — return value of type `int` ignored

Macros

`macroundef` — undefined identifier in macro
`macroparens` — macro parameter used without parentheses
`macroparams` — macro parameter not used exactly once

Others

`specundef` — function or variable specified but never defined
`infloops` — likely infinite loop is detected
`casebreak` — non-empty case in a switch without preceding break
`unreachable` — code detected that is never executed

Modes

Figure A-2 shows the flag settings for each mode. A • means the flag is on (+), otherwise the flag is off (−). The default mode is `std`. Turning type equivalence flags on makes checking weaker. Turning message control flags on makes checking stronger.

Type Equivalence Flags

	weak	std	strict		weak	std	strict
boolint	•			charindex	•		
charint	•			enumint	•	•	
forwarddecl	•			numliteral	•	•	•
voidabstract	•			zeroptr	•	•	•

Message Control Flags

	weak	std	strict		weak	std	strict
globals	•	•	•	globunspec			•
globuse	•	•	•	modifies		•	•
modunspec			•	stdio			
topuse			•	paramuse		•	•
varuse	•	•	•	fcnuse	•	•	•
exportvar			•	exportfcn			•
exporttype			•	overload			
incondefs		•	•	bool			•
pred	•	•	•	predptr	•	•	•
ptrarith			•	ptrcompare		•	•
strictops			•	returnval	•	•	•
returnvalbool		•	•	returnvalint		•	•
macroundef		•	•	macroparens		•	•
macroparams		•	•	specundef		•	•
infloops		•	•	casebreak		•	•
unreachable		•	•				

Figure A-2: Mode settings

A.6 Messages

This section lists the messages related to source code checking produced by LCLint. Most messages should be self-explanatory. Explanations are provided for messages that may not be clear. If a message can be suppressed, information on suppressing it is given. (The opposite flag setting can be used to get the message.)

Type Checking

Array fetches

- Array fetch from non-array (*type*): *expr*
- Array fetch using non-integer, *type*: *expr*

  `+charindex` allows `chars` to be used as array indices

Binary Operators

Abstraction violations

Apply to all operators

- *which* operand of *op* is abstract type (*type*): *expr*
- Operands of *op* are abstract type (*type*): *expr*
- Assignment of *type* to *type*: *expr*
- *var* initialized to type *type*, expects *type*: *expr*

Numeric operators (`*`, `*=`, `/`, `/=`, `+`, `+=`, `-`, `-=`)

Suppressed by `-strictops`

- Operands of *op* are non-numeric (*type*): *expr*
- *which* operand of *op* is non-numeric (*type*): *expr*
- Pointer arithmetic (*type*, *type*): *expr*

  Suppressed by `-ptrarith`

Integer operators (`%`, `<<`, `>>`, `|`, `|=`, `<<=`, `>>=`, `%=`)

Suppressed by `-strictops`

- Operands of *op* are non-integer (*type*): *expr*
- *which* operand of *op* is non-integer (*type*): *expr*
- Comparison of pointer and numeric (*type*, *type*): *expr*

  Suppressed by `-ptrcompare`

- Operands of *op* are non-boolean (*type*): *expr*
- *which* operand of *op* is non-boolean (*type*): *expr*

Casting

- Cast from abstract type *type*: *expr*
- Cast to abstract type *type*: *expr*
- Redundant cast involving abstract type *type*: *expr*

  Cast type is the same as type of the expression

- Cast to underlying abstract type *type*: *expr*

  Cast to an exposed type that is typedefined to an abstract type

- Cast from underlying abstract type *type*: *expr*

Function Calls

- Function *fcn* called with *num* args, declared void
- Function *fcn* called with *num* args, expects *num*
- Function *fcn* expects arg *num* to be *type* gets *type*: *expr*
- Pointer to abstract type (*type*) used as void pointer (arg *num* to *fcn*): *expr*

  Use `+voidabstract` to allow abstract pointers to be used as void pointers
- Call to non-function (*type*): *var*

Structure accesses

- Access field of abstract type (*type*): *expr*
- Access field of non-struct or union (*type*): *expr*
- Access non-existent field of *type*: *expr*
- Arrow access field of abstract type (*type*): *expr*
- Arrow access field of non-struct or union pointer (*type*): *expr*
- Arrow access of non-pointer (*type*): *expr−>field*

Unary Operators

- Operand of *op* is abstract type (*type*): *expr*

  Applies to all unary operators except `&`
- Operand of *op* is non-numeric (*type*): *expr*

  Applies to `+`, `−`
- Operand of *op* is non-integer (*type*): *expr*
- Operand of *op* is non-boolean (*type*): *expr*

  Applies to `!`
- Reference of non-pointer (*type*): *expr*

  Applies to `*` only

General

- *test* predicate not bool, type *type*: *expr*

  Reported for `for`, `if`, `while`, and conditional expressions. `−pred` suppresses all predicate type checking, `−predptr` suppresses messages where the predicate is a pointer, `−boolint` makes `int` equivalent to `bool` so `int` predicates to not generate errors
- Conditional clauses are not of same type: *expr* (*type*), *expr* (*type*)
- Empty return in function declared to return *type*
- Return value type *type* does not match declared type *type*: *expr*

Globals

Globals checking (see Section 2.2) is suppressed completely by `−globals`. Checking in unspecified functions is suppressed by `−globunspec`.

- Global *var* listed (*where*) but not used

  Suppressed by `−globuse`
- Called procedure *fcn* may access global *var*

Suppressed by `-globals`

- Unauthorized use of global *var*

  Suppressed by `-globals`

Global aliases (see Section 2.3.2)

- Function returns with global variable *var* aliasing *loc*
- Function returns with parameter *var* aliasing global *loc*

Modifies

Modifies checking (see Section 2.3) is suppressed completely by `-modifies`. Checking in unspecified functions is suppressed by `-modunspec`.

- Suspect modification of *loc*: *expr*
- Suspect modification of *loc* through alias *loc*: *expr*
- Called procedure *fcn* may modify *loc*: *expr*
- Called procedure *fcn* may modify *loc* through alias *loc*: *expr*

Macros (see Section 2.5)

- Assignment to macro parameter: *expr*
- Operand of *op* is macro parameter (non-functional): *expr*

  Non-functional macro behavior using ++ or --.

- Specified constant implemented as parameterized macro: *var*
- Specified variable implemented as parameterized macro: *var*
- Specified variable implemented as macro: *var*
- Macro *name* specified with *num* args, defined with *num*
- Macro *name* specified as function, declared without parameter lists
- Macro parameter used without parentheses: *var*

  Suppressed by `-macroparens`

- Macro parameter not used: *var*

  Suppressed by `-macroparams`

- Macro parameter not used on some conditional path: *var*

  Suppressed by `-macroparams`

- Macro parameter used more than once: *var*

  Suppressed by `-macroparams`

- Macro parameter used more than once on some path: *var*

  Suppressed by `-macroparams`

- Macro parameter not used on some path, used more than once on different path: *var*

  Suppressed by `-macroparams`

Declarations

Exported but not specified

A variable, function or type is exported in a `.h` file, but is not specified.

- Variable exported, but not specified: *var*
  Suppressed by `-exportvar`
- Function exported, but not specified: *fcn*
  Suppressed by `-exportfcn`
- Type exported, but not specified: *type*
  Suppressed by `-exporttype`

Inconsistent

A variable, function or constant is declared with different types in the specification and implementation.

- Function *fcn* return type *type* does not match specified type *type* (ignoring specification)
- Function *fcn* specified with *num* args, declared with *num* args
- Parameter *num* type mismatch: specified *type*, declared *type*
- Function *fcn* specified with at least *num* args, declared with *num* args
- Function *fcn* specified with *num* args, declared with *num* args
- *var* specified *type* but declared *type*
- Constant *name* specified as *type*, defined as *type*: *expr*
- Function *fcn* specified to return *type*, implemented as macro having type *type*: *expr*
- Overloading standard library function *fcn* with inconsistent definition
  Suppressed by `-overload`
- Redefinition of static variable *var* in same file with inconsistent type: *type*
  Suppressed by `-incondefs`
- Redefinition of *var* with inconsistent type: *type*
  Suppressed by `-incondefs`

Unused

A parameter, function or variable is declared but never used. For declarations in the global scope, checking is suppressed by `-topuse`.

- Parameter not used: *var*
  Suppressed by `-paramuse`
- Function declared but not used: *fcn*
  Suppressed by `-fcnuse`
- Variable declared but not used: *var*
  Suppressed by `-varuse`
- Return value (type *type*) ignored: *expr*
  Suppressed by `-returnval`. To suppress only messages where the return value type is an `int` or a `bool`, use `-returnvalint` or `-returnvalbool` respectively.

Others

- Fall through case (no preceding break)

  A non-empty case is followed by the next case with no intervening `break` or control flow statement. Suppressed by `-casebreak`
- Suspected infinite loop: no condition values modified

  The body of a loop does not modify any value in the loop condition and the loop condition is not constant. Assumes any function call may modify any global variable. Suppressed by `-infloops`
- Path with no return in function declared to return *type*
- Unreachable code

  Suppressed by `-unreachable`
- Variable *var* used before set
- Out parameter *var* used before set

  See Section 2.4
- Mutable abstract type *type* declared without pointer indirection: *type*

  An abstract mutable type is implemented in a way that may not exhibit sharing semantics. See Section 1.2.4

A.7 Availability

LCLint is available via anonymous ftp from `larch.lcs.mit.edu`.

The current release is in `pub/Larch/lclint`*version*`.`*platform*`.tar.Z`. The release includes executables, documentation including the user's guide, the LCL grammar, UNIX manual pages, and a few examples including the `dbase` example used in Chapter 3. The latest version of the user's guide is available as a separate postscript file in the file `lclint`*version*`.userguide.ps` in the same directory.

Uncompress this file and unpack the archive:

```
% uncompress filename
% tar xvf filename
```

The file `INSTALL` describes the installation procedure. A shell script, `lclintvars`, is provided to set the appropriate environment variables. To start using LCLint, run this script, and copy the output into one of your login files (such as `.environment`). The `imports` subdirectory contains specifications for the standard libraries. When an LCL specification imports a standard library (using `imports` <*libname*>), LCLint will look for the file *libname*`.lcs` in the imports directory. The information in these libraries is based on header files in ULTRIX V4.3. If you are using a different operating system, you may find some of the definitions are inconsistent and need to edit these files.

Bibliography

[Ada83] The Ada programming language reference manual. ANSI/MIL-STD 1815A, US Department of Defense, US Government Printing Office, February 1983.

[FO76] L. D. Fosdick and Leon J. Osterweil. Data flow analysis in software reliability. *ACM Computing Surveys*, 8(3), September 1976.

[FP91] Tim Freeman and Frank Pfenning. Refinement types for ML. In *ACM Conference on Principles of Programming Language Design and Implementation*, 1991.

[GH93] John V. Guttag and James J. Horning, editors. *Larch: Languages and Tools for Formal Specification*. Texts and Monographs in Computer Science. Springer-Verlag, 1993. With Stephen J. Garland, Kevin D. Jones, Andrés Modet, and Jeannette M. Wing.

[How90] William E. Howden. Comments analysis and programming errors. *IEEE Transactions on Software Engineering*, SE-16(1), January 1990.

[Jac92] Daniel Jackson. Aspect: A formal specification language for detecting bugs. MIT/LCS/TR 543, Laboratory for Computer Science, MIT, June 1992.

[Joh78] S. C. Johnson. Lint, a C program checker. Computer science technical report, Bell Laboratories, Murray Hill, NH, July 1978.

[Koe89] Andrew Koenig. *C Traps and Pitfalls*. Addison-Wesley, 1989.

[KR88] B. W. Kernighan and D. N. Ritchie. *The C Programming Language, Second Edition*, 1988.

[LAB+81] B. Liskov, R. Atkinson, T. Boom, E. Moss, J. Schaffert, R. Scheifler, and A. Snyder. *CLU Reference Manual*, 1981.

[LG86] Barbara Liskov and John V. Guttag. *Abstraction and Specification in Program Development*. The MIT Electrical Engineering and Computer Science Series. MIT Press, Cambridge, MA, 1986.

[Luc90] David Luckham. *Programming with Specifications: An Introduction to Anna, A Language for Specifying Ada Programs*. Springer-Verlag, 1990.

[Man93] Walter Mann. The Anna Package Specification Analyzer user's guide. CSL-TN 93-390, Computer Systems Laboratory, Stanford University, January 1993.

[MMS88] Keith W. Miller, Larry J. Morell, and Fred Stevens. Adding data abstraction to Fortran software. *IEEE Software*, November 1988.

[OO89] Kurt M. Olender and Leon J. Osterweil. Cesar: A static sequencing constraint analyzer. In *Proceedings of the ACM SIGSOFT'89 Third Symposium on Software Testing, Analysis, and Verification (TAV3)*, 1989.

[OO92] Kurt M. Olender and Leon J. Osterweil. Interprocedural static analysis of sequencing constraints. *ACM Transactions on Software Engineering and Methodology*, 1(1), January 1992.

[Per89] Dewayne E. Perry. The logic of propagation in the Inscape environment. In *Proceedings of the ACM SIGSOFT'89 Third Symposium on Software Testing, Analysis, and Verification (TAV3)*, 1989.

[San89] Sriram Sankar. Automatic runtime consistency checking and debugging of formally specified programs. CSL-TR 89-391, Computer Systems Laboratory, Stanford University, August 1989.

[SH83] Robert Strom and Nagui Halim. A new programming methodology for long-lived software systems. IBM-RC 9979, IBM T. J. Watson Research Center, March 1983.

[Spu90] David A. Spuler. Check: A better checker for C. Honours Thesis, Department of Computer Science, James Cook University of North Queensland, Austrailia, November 1990.

[Str86] Bjarne Stroustrup. *The C++ Programming Language*. Addison-Wesley, 1986.

[Tan94] Yang Meng Tan. Formal specification techniques for promoting software modularity, enhancing software documentation, and testing specifications. MIT/LCS/TR 619, Laboratory for Computer Science, MIT, June 1994.

[WO85] Cindy Wilson and Leon J. Osterweil. Omega — a data flow analysis tool for the C programming language. *IEEE Transactions on Software Engineering*, SE-11(9), September 1985.

[WSS91] Michal Walicki, Jens Ulrik Skakkebæk, , and Sriram Sankar. The Stanford Ada Style Checker: An application of the Anna tools and methodology. CSL-TR 91-488, Computer Systems Laboratory, Stanford University, August 1991.