74

Introduction to LCL, A Larch/C Interface Language

J.V. Guttag and J.J. Horning

July 24, 1991

Systems Research Center

DEC's business and technology objectives require a strong research program. The Systems Research Center (SRC) and three other research laboratories are committed to filling that need.

SRC began recruiting its first research scientists in l984—their charter, to advance the state of knowledge in all aspects of computer systems research. Our current work includes exploring high-performance personal computing, distributed computing, programming environments, system modelling techniques, specification technology, and tightly-coupled multiprocessors.

Our approach to both hardware and software research is to create and use real systems so that we can investigate their properties fully. Complex systems cannot be evaluated solely in the abstract. Based on this belief, our strategy is to demonstrate the technical and practical feasibility of our ideas by building prototypes and using them as daily tools. The experience we gain is useful in the short term in enabling us to refine our designs, and invaluable in the long term in helping us to advance the state of knowledge about those systems. Most of the major advances in information systems have come through this strategy, including time-sharing, the ArpaNet, and distributed personal computing.

SRC also performs work of a more mathematical flavor which complements our systems research. Some of this work is in established fields of theoretical computer science, such as the analysis of algorithms, computational geometry, and logics of programming. The rest of this work explores new ground motivated by problems that arise in our systems research.

DEC has a strong commitment to communicating the results and experience gained through pursuing these activities. The Company values the improved understanding that comes with exposing and testing our ideas within the research community. SRC will therefore report results in conferences, in professional journals, and in our research report series. We will seek users for our prototype systems among those with whom we have common research interests, and we will encourage collaboration with university researchers.

Robert W. Taylor, Director

Introduction to LCL,
A Larch/C Interface Language

J.V. Guttag and J.J. Horning

July 24, 1991

Abstract

This report is aimed primarily at the C programmer who wishes to begin to integrate formal specifications into the program development cycle. We present a specification language targeted specifically at C and discuss how it can be used to support a style of C programming in which abstraction plays a vital role.

The report begins with a quick overview of the use of the Larch family of languages for program specification. It continues with an overview of LCL, a Larch interface language for (ANSI) standard C. It then describes LCL by means of an extended example. Parts of an implementation of the specified interfaces are provided in the body of the report. The remaining parts of the implementation are presented in an appendix. Another appendix contains a brief introduction to the Larch Shared Language.

Contents

List of Figures

1 An overview of Larch

The most vexing problems in building systems concern overall system organization and the integration of components. Modularity is the key to controlling them, and specifications are essential for achieving program modularity. Abstraction boundaries make it possible to understand programs one component at a time. However, an abstraction is intangible. Without a precise description, there is no way to know what it really is, and it is easy to confuse an abstraction with one of its implementations.

Specifications can be written in natural languages, in semi-formal notations (with a restricted syntax but no formal semantics), or in truly formal notations. The potential advantages of formal specifications are that they have unambiguous meanings and are subject to manipulation by programs. The latter advantage can be fully realized only by using tools that support constructing and reasoning about them. The Larch Project is developing languages, tools, and techniques to aid in the productive application of formal specifications to software design, implementation, integration, and maintenance. [Guttag, et al. 85, 90, Garland et al. 90]

A Larch *interface specification* describes the interface that a program component provides to *clients* (programs that use it). Each interface specification is written in a programming-language-dependent *Larch interface language*. It relies on definitions from an *auxiliary specification*, written in a programming-language-independent specification language, the *Larch Shared Language* (*LSL*).

The Larch family of specification languages support:

- *Specification reuse.* Many language-independent abstractions are useful in a wide variety of specifications, for example, integers, lists, sets, queues, arrays, relations, mappings, and orders. Larch encourages the accumulation of open-ended collections of reusable specification components in LSL handbooks.
- *Abstraction.* Larch supports a style of program design in which data and functional abstractions play a prominent role.
- *Development tools.* The Larch languages are designed for use with tools that support the construction and checking of specifications, implementations, and clients.

Many informal specifications have a structure similar to Larch's. They rely on auxiliary specifications, but leave them implicit. They describe an interface in terms of concepts—such as sets, lists, or files—with which readers are assumed to be familiar. But they don't define them. Readers may misunderstand such specifications unless their intuitive understanding precisely matches the specifier's. And there's no way to be sure that such intuitions match. LSL specifications solve this problem by mathematically defining the terms that appear in interface specifications. Appendix B provides a brief introduction to LSL. A complete definition is available in a separate report [SRC-58].

An interface specification provides information that is needed both to write client programs and to write acceptable implementations. A critical part of a component's interface is its communication with its environment. Communication mechanisms differ from one programming language to another, sometimes in subtle ways. It is easier to be precise about communication when the specification language reflects the programming language. Such specifications are generally shorter than those written in any "universal" interface language. They are also clearer to programmers who implement interfaces and to programmers who use them.

Each Larch interface language deals with what can be observed about the behavior of program components written in a particular programming language. It provides a way to write assertions about program states. It incorporates notations that are specific to its programming language for constructs such as side effects, exception handling, concurrency, and iterators. Its simplicity or complexity depends largely on the simplicity or complexity of its programming language.

Each Larch interface language has a mechanism for specifying abstract data types. If its programming language doesn't provide direct support for them (as C does not), the mechanism is designed to be compatible with the general style of the programming language.

LCL is a Larch interface language designed to specify program components written in, or called from, the standard C programming language [ANSI]. For comparison, LM3, a Larch interface language for Modula-3 is described in [Jones 91].

2 LCL preliminaries

This report describes most of LCL (version 1.0) and gives an informal description of its semantics. It discusses some LCL tools, but it is not a user's guide for any of them.

LCL is not a C dialect or preprocessor. Programs specified and developed with LCL are C programs, accepted by ordinary C compilers. The use of LCL will tend to encourage some styles of development, but it does not change the programming language.

Before presenting any interface specifications, we discuss the intended relation between LCL specifications and C programs, the structure of LCL function specifications, and the relation of names appearing in LCL specifications to values in C states.

2.1 LCL specifications and C implementations

C is a general and flexible language that is used in many different ways. A common style for organizing programs is to construct them as a set of program units, often called *modules*. A module consists of an *interface* and an *implementation*. The interface is a collection of types, functions, variables, and constants for use in other modules, called its *clients*.

A C module M is typically represented by three files:

- M.c contains most of its implementation, including function definitions and private data declarations.
- M.h contains a description of its interface, plus parts of its implementation. Comments provide an informal specification of the module for the guidance of client programmers. Type declarations, function prototypes, constant definitions, declarations of external variables, and macro definitions provide all the information about M that is needed to compile its clients.
- M.o contains its compiled form. Such files are linked together to create executable files.

C modules specified using LCL have two additional files:

- M.lcl contains its LCL interface specification, a formal description of the types, functions, variables, and constants provided for clients, together with comments providing informal documentation. It replaces M.h as documentation for client programmers. The extra information it provides will also be exploited by a planned *LCLint* tool to perform more extensive checking than an ordinary C lint.
- M.lh is a header file derived automatically from M.lcl to be included in M.h. Mechanical generation of .lh-files file saves the user from having to repeat information in the .h-file. This reduces the bulk of the implementation and avoids an opportunity for error. The implementation portion of M.h must still be provided by the implementor.

M.lcl may also refer to another kind of file:

- .lsl-files contain auxiliary specifications in the form of LSL *traits*. A trait precisely defines operators used in .lcl-files.

Traits are the principal reusable units in Larch specifications. An interface specification (.lcl-file) may refer to more than one trait and a trait may be referred to by more than one interface. Commonly useful traits are collected into handbooks.

2.2 Function specifications

A C function may communicate with its callers by returning a result, by accessing objects accessible to the caller, or by modifying such objects. The specification of each function in an interface can be studied, understood, and used without reference to the specifications of other functions. A specification consists of a function prototype followed by a body of the form:

requires $reqP$;
modifies $modList$;
ensures $ensP$;

A specification places constraints on both clients and implementations of the function. The *requires clause* states restrictions on the arguments with which the client is allowed to call it. The *modifies* and *ensures clauses* place constraints on its behavior when it is called properly. They relate two states, the state when the function is called, which we call *pre*, and the state when it terminates, which we call *post*. A requires clause refers only to values in *pre*. An ensures clause may also refer to values in *post*, including the value returned by the function, written as *result*.[1]

A modifies clause says what a function is allowed to change. It says that the function must not change the value of any objects visible to the caller except for a specified list. Any other object must have the same value in *pre* and *post*. If there is no modifies clause, then nothing may be changed. Of course, it would be an error to include a const parameter in a modifies clause.

For each call, it is the responsibility of the client to make the requires clause true in the pre state. Having done that, the client may presume that: the function will terminate, the ensures clause will be true on termination, and changes will be limited to the objects indicated in the modifies clause. The client need not be concerned with how this happens.

The implementor of a function is entitled to presume that the requires clause holds on entry, and is not responsible for the function's behavior if it does not. Since a function's behavior is totally unconstrained unless its requires clause is satisfied, it is good style to use the weakest feasible requires clause. An omitted requires is equivalent to the weakest possible requirement, *requires true*.

In summary, a specification as a whole is a predicate on the pre and post states, interpreted as

$reqP$(pre) =>
    ($terminates$ /\ $modP$(pre, post) /\ $ensP$(pre, post))

where => stands for logical implication, and /\ stands for conjunction (logical and).

[1] Part of the post state is the point to which control will be transferred. For most invocations, this is the return address of the pre state; constructs like exit, abort, and longjump can be specified as modifications of the pseudo-variable *control*.

2.3 States and names

Simplifying slightly, *states* are mappings from *locs* (locations) to *objects*. Each variable identifier names a loc. The major kinds of objects are:

- *basic values*. These are mathematical abstractions, like the integer 3 and the letter A. Such values are independent of the state of any computation. As discussed in Appendix B, LSL is used to give meaning to basic values.
- *locs*. These store objects; for example, intLocs store objects of type int. The *value stored in a loc* in a state is the object to which the state maps the loc.
- *structs*. These are collections of locs, each denoted by a member name. For example, given the variable declaration
  ```
  struct {int first; char second;} s;
  ```
  s.first denotes an intLoc and s.second a charLoc.
- *unions*. These are similar to structs, except that their locs overlap.
- *arrays*. These are bounded vectors of adjacent locs, indexed from 0. If a is an array, maxIndex(a) is its upper bound.[2]
- *pointers*. These are references to collections of one or more adjacent locs, each denoted by an offset from a base address. They can be thought of as triples consisting of a loc and two bounding indexes. For example, given the code
  ```
  int a[100];
  int *p;
  p = &(a[1]);
  ```
  *p denotes an intLoc in the region allocated to a, and minIndex(p) and maxIndex(p) denote the maximum number of intLocs before and after *p, respectively (1 and 98). These locs are accessible using arithmetic on p.

[2] C does not make the values of maxIndex and minIndex available at runtime, but they are useful for specifying and reasoning about programs.

The following LCL primitives are available for accessing the pre and post states:

- `^` can be applied to locs, arrays and structs. It is used to extract their values from the pre state. It cannot be applied directly to unions, but can be applied to the loc yielded by applying a field selector to a union.
  - When applied to a loc, it yields the value stored in that loc in the pre state).
  - When applied to an array, it yields a vector of the same length containing the values stored in the array's locs in the pre state.
  - When applied to a struct, it yields a tuple containing the values stored in the struct's locs in the pre state.
- `'` is like `^`, but extracts values from the post state.
- `*` is used, as in C, to dereference a pointer,[3] producing its loc with offset 0.
- `->`, as in C, is a syntactic shorthand to dereference a pointer to a struct and then select one of its members. For example, a->b is equivalent to (*a).b .
- `[i]` is used, as in C, to index into an array, producing a loc.
- `[]` is applied to a pointer to cast it into an array. For example, `p[]` is an array whose first loc is `*p` and whose upper bound is maxIndex(p), and `p[]^` is a vector.

LCL is strongly typed. Each identifier's type defines the kind of objects to which it can map in any state. Similarly, each LSL value has a unique *sort*. To connect the two languages, there is a mapping from C types (and LCL abstract types) to LSL sorts. Each built-in type of C, each type built from C type constructors (e.g., int `*`), and each abstract type defined in LCL is *based on* an LSL sort. LCL specifications are written using types and values. The properties of these values are defined in LSL, using operators on the sorts on which those types are based.

A standard LSL trait defines operators of the sorts upon which C builtin types and type constructors are based. Users familiar with C will already know what these operators mean.

Consider the specification fragment:

```
void f(int i, int a[], const int *p) {
  requires i >= 0 /\ i <= maxIndex(a);
  modifies a;
  ensures a[i]' = (*p)^ + 1;
  }
```

[3]It is also used to dereference an abstract ref, as described in Section 3.6.

Since ints are passed by value in C, i denotes not an intLoc but an int value.[4] The expression `a[i]` denotes the i*th* loc of the array `a`. Applying `'` to this loc yields the int it stores in the post state. Applying `*` to the pointer p yields an intLoc. Applying `^` yields its int value in the pre state. The other operators are defined by the standard trait for int.

3 A guided tour through a specification

To illustrate the use of most of LCL's features, we present and discuss a small specification. This example is only superficially realistic; it was structured to use language constructs in the order we want to discuss them. It is not really a typical specification, or an especially wonderful program design. As you study this report, you will probably find it instructive to consider alternative designs and how they would be specified.

The example in this section uses various conventions for names, formatting, comments, etc. These are not mandated by LCL; specifications should be written using the conventions of the organization for which they are intended. Because the example is being used to document LCL features, rather than a real interface, the density of comments embedded within the formal text is low, and most of the comments are in the accompanying prose.

This example has been machine-checked (just as the prose has been mechanically spell-checked). The .lcl- and .lsl-files have been checked by the LCL and LSL Checkers, respectively. The .lh-files were automatically generated by the LCL Checker. The .lh-, .h-, and .c-files were compiled by gcc (this took somewhat longer than all the Larch checking). Finally, the compiled code was exercised by a test driver. Although we tried to be very careful at each stage of development, each of the mechanical checks caught some errors that we had not. Based on this experience, we expect that when LCLint is available, it will find a few more errors (just as we expect that careful readers will find a few typos in the prose). These will probably be errors that would manifest themselves only in very unusual circumstances, and would therefore be difficult to root out by testing.

3.1 Gender

The interface specified in Figure 1, gender, exports a type, a constant, and two functions to its clients.

The first line defines an *exposed type*, also named gender, using a C typedef. Clients of this interface are being told exactly how gender is represented as a C type. They

[4] Within the implementation of f, a loc will be associated with the formal, but since that loc does not exist in the environment of the caller of f, it is not relevant to the specification.

```
/* Exports one type, a function to convert gen-
ders to          */
/* strings and a function to (trivially) initialize the mod-
ule. */

typedef enum {MALE, FEMALE, gender_ANY} gender;

constant int gender_maxPrintSize = lenStr("unknown gender");

uses sprint(gender, char[]);

int gender_sprint(char s[], gender g) {
   requires maxIndex(s) >= gender_maxPrintSize;
   modifies s;
   ensures isSprint(s', g)
           /\ result = lenStr(s')
           /\ result <= gender_maxPrintSize;
   }
void gender_initMod(void) { ensures true; }
```

Figure 1: gender.lcl

may deal with gender values in any way allowed by standard C. However, LCL's type checking is stricter than standard C's. LCL uses name equality for type checking, and LCLint will warn programmers about type violations that C lint will not catch.[5]

The theory of C's types and type constructors is built into LCL. C's enum types are axiomatized using LSL's *enumeration of* shorthands, and struct types using *tuple of* shorthands.

The *constant declaration* gives a symbolic name for an important property of the interface: the size of the longest string gender_sprint is allowed to return. LCL interface constants may be implemented either by macro definitions or by C const variables.

A *uses* clause invokes and auxiliary specification—an LSL trait that defines operators used in the LCL specification. Users familiar with the operators involved may not need to examine such traits closely, but most users are expected to read them. The uses clause here incorporates an LSL specification that gives the meaning of operators such as isSprint and lenStr. It also says the sort T of sprint.lsl is to be replaced by the sort gender (on which the type gender is based) and the sort String by whatever sort the type char [] is based on (its name isn't important).

The function gender_sprint is typical of a kind found in many interfaces. It converts

[5]But, in deference to long tradition, LCLint will use structural type checking on calls to standard library functions.

gender values into a string form suitable for printing, and returns the length of that string. Its specification begins with its *function prototype*. LCL prototypes are more restricted than C's. For example, LCL requires that each of the formal parameters be named, although names are optional in C. This guarantees that the specification can refer to any parameter by name. Since all functions in an interface are exported, the keyword extern will be added automatically when gender.lh is generated.

LCL distinguishes between pointers and arrays in prototypes. In a C prototype, char `*s` and char `s[]` are essentially equivalent. In an LCL prototype, however, char `*s` allows access to all of the characters from $*(s - \text{minIndex}(s))$ to $*(s + \text{maxIndex}(s))$, while char `s[]` allows access only to the characters from `s[0]` to s[maxIndex(s)].

The body of the specification consists of three clauses. The requires clause says that the array s must be big enough to hold the longest string that will ever be returned. The modifies clause says that only the contents of the array s can be changed. The ensures clause constrains the new value of s and the function's result.[6]

Arrays are passed by reference in C, so the formal s refers to the array, rather than its contents. The term s' denotes the vector of characters contained by the locs in s upon return from gender_sprint. Since parameters of enumeration types are passed by value, g denotes a value of type gender. The meanings of isSprint and lenStr are given in Figures 2 and 3, which are discussed below.

This specification does not say what string will be generated for each gender value—only that it will have certain properties. We might want such freedom, for example, in a module that will have different implementations for different countries or languages. This specification doesn't even require an implementation to be *deterministic*; for example, it doesn't require gender_sprint(s, MALE) to always put the same chars in s, or to always return the same int value. Although our implementation of gender doesn't take advantage of this freedom, later interfaces will have implementations that do.

The trait sprint.lsl was written for specifying functions that convert values to strings. It includes the library trait string, which specifies the operators nullTerminated and lenStr. Note that string trait, like C, defines the value of lenStr only when it is applied to a null-terminated string.

The trait in Figure 3 is intentionally weak. It doesn't say much about the meanings of its operators. This allows considerable flexibility in implementing the interface functions.[7] The first two assertions guarantee that different T values will have different string forms, without specifying what those forms are. The second equation gives two important properties of acceptable string forms. We could repeat these properties in the interface specification of each such function, but it is better to get them right once, and then reuse the trait.

[6] A good rule of thumb is that each object in the modifies clause should appear in primed form at least once in the ensures clause.

[7] It is hard to write a specification that leaves the implementation so much flexibility, but still imposes the necessary constraints. sprint is the most subtle trait in this report.

```
% Define the relation between C's vectors of chars and
% C's conventions for null-terminated character strings.

string: trait
includes integer
introduces
    null: -> char
    empty: -> String
    append: String, char -> String
    len: String -> int
    nullTerminated: String -> bool
    throughNull: String -> String
    sameStr: String, String -> bool
    lenStr: String -> int
    % and many other operators not used here ...

asserts
    String generated by empty, append
    forall s, s1, s2: String, c: char
        len(empty) == 0;
        len(append(s, c)) == len(s) + 1;

        not(nullTerminated(empty));
        nullTerminated(append(s, c)) ==
                         c = null \/ nullTerminated(s);

        nullTerminated(s) =>
            throughNull(append(s, c)) = throughNull(s);
        not(nullTerminated(s)) =>
            throughNull(append(s, null)) = append(s, null);

        sameStr(s1, s2) == throughNull(s1) = throughNull(s2);

        lenStr(s) == len(throughNull(s)) - 1
        % and many other axioms not needed here ...
```

Figure 2: string.lsl fragment

```
% Defines minimum requirements for an unparse function that
% converts from a T to a String without losing information.

sprint(T, String): trait

includes string

introduces
   parse: String -> T
   unparse: T -> String
   isSprint: String, T -> bool

asserts
   T partitioned by unparse
   forall t: T, s: String
       parse(unparse(t)) == t;
       isSprint(s, t) == parse(s) = t /\ nullTerminated(s)
```

Figure 3: sprint.lsl

```
typedef enum {MALE, FEMALE, gender_ANY} gender;

extern int gender_sprint(char s[], gender g);
extern void gender_initMod(void);
```

Figure 4: gender.lh

In this example, we include an initMod function as part of every interface. Later we will discuss the way in which we use these functions. The function gender_initMod is required by its specification to have no visible effect, since it modifies nothing and returns no value. The absence of a requires clause (equivalent to *requires true*) says that it must always terminate.

From gender.lcl the LCL Checker generates the file gender.lh, Figure 4. This is used in the implementation of gender.h, Figure 5, and hence, gender.c, Figure 6.

By convention, we start our .h-files with a #if that makes sure that including them more than once into the same module will not cause a problem. Both gender.c and all clients of gender will include gender.hİn turn, gender.h includes gender.lh, which provides prototypes. The implementation of the function gender_initMod is also in gender.h.

```
#if !defined(gender_h_expanded)
#define gender_h_expanded
#define gender_maxPrintSize (sizeof("unknown gender"))

#include "gender.lh"

#define gender_initMod()
#endif
```

Figure 5: gender.h

```
#include <string.h>
#include "gender.h"

int gender_sprint (char s[], gender g) {
    static char *resultstr[] ={"male", "female", "unknown gender"};

    s[0] = '\0';
    (void) strncat(s, resultstr[g], gender_maxPrintSize-1);

    return strlen(s);
}
```

Figure 6: gender.c

3.2 Employee

The employee interface, Figure 7, directly exports to its clients two constants, three exposed types, and three functions.

The *imports* clause says that the specification of the employee interface depends on the specification of the gender interface; it gives employee and its clients access to the type gender and the function gender_sprint. It also makes the trait associated with the gender interface available for use in the specification of the employee interface. Such specification dependencies should not be confused with implementation dependencies, where one module is used within the implementation of another; clients should not be concerned with what modules the implementation uses.

The constant clause equates the C constant maxEmployeeName and the LSL constant MaxEmployeeName. Looking in employeeName.lsl, Figure 8, we see that the implementation has a great deal of freedom in implementing this constant; any int greater than zero is allowed.

The exposed types in this interface are conventional. We will later ensure that (in any database) each Social Security Number (ssNum) identifies a unique employee, so we can use it as a key into the database. Cf. Figure 17 and the discussion on page 3.4.

```
imports gender;

constant int maxEmployeeName = MaxEmployeeName;
constant int employee_maxPrintSize =
             maxEmployeeName + gender_maxPrintSize + 30;

typedef enum {MGR, NONMGR, job_ANY} job;
typedef char employeeName[maxEmployeeName];
typedef struct {int ssNum;
                employeeName name;
                int salary;
                gender gen;
                job j;} employee;

uses employeeName, sprint(employee, char[]);

bool employee_setName(employee *e, employeeName na) {
   requires nullTerminated(na^);
   modifies e->name;
   ensures result = lenStr(na^) < maxEmployeeName
           /\ (if result
               then sameStr(e->name', na^)
                    /\ nullTerminated(e->name')
               else unchanged(e->name));
   }
int employee_sprint(char s[], employee e) {
    requires maxIndex(s) >= employee_maxPrintSize;
    modifies s;
    ensures isSprint(s', e)
            /\ result = lenStr(s')
            /\ result <= employee_maxPrintSize;
    }
void employee_initMod(void) {
    ensures true;
    }
```

Figure 7: employee.lcl

Like gender.lcl, employee.lcl uses the sprint trait. This means that employee incorporates sprint.lsl twice, with different renamings: directly with employee for T, and indirectly with gender for T. It also uses employeeName, which was written specifically for use in employee.lcl, and needs no renamings.

In addition to employee_sprint and employee_initMod functions, this interface exports the function employee_setName. This function returns a value of type bool, the one builtin type of LCL that is missing from C. When LCL specifications are checked, bool is treated as a distinct type. If the type identifier bool appears in an LCL specification, the Checker places #include "bool.h" in the corresponding .lh-file. A typical bool.h is shown in Figure 9.

The requires clause in employee_setName says that it should be called only with null-terminated strings. The implementation is entitled to rely on this. Indeed, it often must. It is not generally possible to determine at runtime the maxIndex of an array. Yet without a guarantee that a string is null-terminated, it is not safe to search for its terminating null. The search might run past the end of the allocated storage and generate references to nonexistent memory. Completely defensive programming just isn't possible in C.

The modifies clause says that employee_setName may change the name field, e−>name, of its first argument, but nothing else. This is a finer-grained constraint on modification than is possible using only C's const qualifier. Unlike requires and ensures clauses, a

```
employeeName: trait

includes string(employeeName for String), integer

introduces MaxEmployeeName: -> int

asserts equations
   MaxEmployeeName > 0
```

Figure 8: employeeName.lsl

```
#if !defined(bool_h_expanded)
#define bool_h_expanded
#define FALSE 0
#define TRUE (!FALSE)
typedef int bool;
#define bool_initMod()
#endif
```

Figure 9: bool.h

```
#include "bool.h"
#include "gender.h"

typedef enum {MGR, NONMGR, job_ANY} job;
typedef char employeeName[maxEmployeeName];
typedef struct {int ssNum;
                employeeName name;
                int salary;
                gender gen;
                job j;} employee;

extern bool employee_setName(employee *e, char na[]);

extern int employee_sprint(char s[], employee e);

extern void employee_initMod(void);
```

Figure 10: employee.lh

modifies clause constrains everything it doesn't mention.

The ensures clause says that employee_setName will have one of two outcomes. It will either:

- Make the name field of its first argument the same as its second argument (when both are interpreted as strings), make the new value of the name field be null-terminated, and return TRUE, or
- Change nothing and return FALSE.

Furthermore, the first outcome will occur exactly when the new name fits, (i.e., lenStr(na^) < maxEmployeeName). The use of *result* in several subterms of an ensures clause is a frequent idiom. Since the predicate in the ensures clause is just a logical formula, it makes no semantic difference whether the equation for *result* is written first or last. We are free to choose an order that helps the exposition or emphasizes some particular aspect of the specification.

A number of design decisions are recorded in employee.lcl. It says which functions must be implemented, and for each function it indicates both the conditions that must hold at the point of call and the conditions that must hold upon return. This constitutes a contract between the implementation and the clients of employee that establishes a "logical firewall," allowing their programmers to proceed independently of each other, relying only on the interface specification.

The file employee.lh, Figure 10, is automatically constructed from employee.lcl. In addition to the appropriate typedefs and function prototypes, it #includes the .h-files of the explicitly imported interface gender and the implicitly imported interface bool.

```
#if !defined(employee_h_expanded)
#define employee_h_expanded

#define maxEmployeeName 20
#define employee_maxPrintSize (maxEmployeeName + gen-
der_maxPrintSize + 30)
#include "employee.lh"

#define employee_initMod()\
            do {bool_initMod(); gender_initMod();} while (0)
#endif
```

Figure 11: employee.h

The file employee.h, Figure 11, defines the constant maxEmployeeName using a macro. Because of a restriction imposed by C, this definition must precede the inclusion of employee.lh, since the constant is used in the typedef of employee_name contained in employee.lh. The #define cannot be automatically generated because the LCL processor has no way of knowing what value the constant is to have; the specification leaves that decision to the implementation.

The file employee.h also implements employee_initMod. Our convention is for each module to initialize any modules it explicitly imports. Thus employee_initMod calls gender_initMod. Since the specification of this function guarantees that it modifies nothing, calling it multiple times cannot have effects visible to clients.

In general, M.h contains, in order:

- A test of whether M_h_expanded is defined in the current context. This makes sure, for example, that a client of employee can safely include both employee.h and gender.h without getting an error caused by a second occurrence of the type definition for gender.
- A definition of M_h_expanded.
- Definitions of all constants declared in M.lcl, either as macros or as C const variables.
- Concrete representations (typedefs) for any abstract types declared in M.lcl. Abstract data types are discussed in the next section.
- An include of M.lh.
- Macros, if any, for inline implementations of functions with prototypes in M.lh.

The implementation of employee_setName in employee.c, Figure 12, relies on the requires clause in its specification. It may crash if na^ isn't null-terminated.

```
#include <string.h>
#include "employee.h"

bool employee_setName(employee *e, employeeName na) {
   int i;

   for (i = 0; na[i] != '\0'; i++)
      if (i == maxEmployeeName) return FALSE;
   strcpy(e->name, na);
   return TRUE;
}
int employee_sprint(char s[], employee e) {
   char gstring[gender_maxPrintSize];
   static char *jobs[] = {"manager", "non-manager", "unknown job"};

   gender_sprint(gstring, e.gen);

   (void) sprintf(s, "%d,       %s,      %s,       %s,         $%d",
                     e.ssNum, e.name, gstring, jobs[e.j], e.salary);
   return strlen(s);
}
```

Figure 12: employee.c

3.3 Empset

The interface empset.lcl, Figure 13, exports a set of functions and an *abstract data type*. Types specified in LCL can be either exposed or abstract. As we have seen, exposed types are specified using C typedefs. Abstract types are specified by specifying a collection of functions that create, examine, and manipulate their values, leaving their representation as a "secret" of the implementation.

Although C provides no direct support for abstract types, there is a style of C programming in which they play a prominent role. The programmer relies on conventions to ensure that the implementation of an abstract type can be changed without affecting the correctness of clients. The key restriction is that clients never directly access the representation of an abstract value. All access is through the functions provided in its interface.

To ensure that client programs are independent of the way abstract types are represented, several restrictions on their use are necessary. Values of abstract types must not be assigned with = or compared with ==.[8] Without these restrictions, the choice of representations would be severely limited; for example, if comparison using == were allowed, structs could not be used at the top-level of a representation. More importantly, these operators would likely have surprising semantics in client programs. Consider, for example, two empsets, $s1$ and $s2$. Suppose each empset was implemented by a

[8]Ref abstract types, discussed below, are an exception to this rule.

```
/* empset is a set of employees             */
/* set.lsl can be found in an LSL handbook */

imports employee;
abstract type empset;
uses set(employee for Elem, empset for Set),
     sprint(empset, char[]);

void empset_init(empset *s) {
   modifies *s;
   ensures (*s)' = { };
   }
void empset_final(empset *s) {
   modifies *s;
   ensures trashed(*s);
   }
void empset_clear(empset *s) {
   modifies *s;
   ensures (*s)' = { };
   }
bool empset_insert(empset *s, employee e) {
   modifies *s;
   ensures result = not(e \in (*s)^) /\ (*s)' = in-
sert(e, (*s)^);
   }
void empset_insertUnique(empset *s, employee e) {
   requires not(e \in (*s)^);
   modifies *s;
   ensures (*s)' = insert(e, (*s)^);
   }
bool empset_delete(empset *s, employee e) {
   modifies *s;
   ensures result = e \in (*s)^ /\ (*s)' = delete(e, (*s)^);
   }
empset *empset_union(empset *s1, empset *s2) {
   ensures (*result)' = (*s1)^ \union (*s2)^ /\ fresh(*result);
   }
```

Figure 13: empset.lcl, part 1

```
empset *empset_disjointUnion(empset *s1, empset *s2) {
   requires (*s1)^ \intersect (*s2)^ = { };
   ensures (*result)' = (*s1)^ \union (*s2)^ /\ fresh(*result);
   }
void empset_intersect(empset *s1, empset *s2) {
   modifies *s1;
   ensures (*s1)' = (*s1)^ \intersect (*s2)^;
   }
int empset_size(empset *s) {
   ensures result = size((*s)^);
   }
bool empset_member(employee e, empset *s) {
   ensures result = e \in (*s)^;
   }
bool empset_subset(empset *s1, empset *s2) {
   ensures result = (*s1)^ \subset (*s2)^;
   }
employee empset_choose(empset *s) {
   requires (*s)^ != { };
   ensures result \in (*s)^;
   }
int empset_sprint(char s[], empset *es) {
    requires maxIn-
dex(s) >= (size((*es)^) * employee_maxPrintSize);
    modifies s;
    ensures isSprint(s', (*es)^)
            /\ result = lenStr(s')
            /\ re-
sult <= (size((*es)^) * employee_maxPrintSize);
    }
void empset_initMod(void) {
    ensures true;
    }
```

Figure 13: empset.lcl, part 2

pointer to some data structure, with NIL representing the empty set. The expression $s1 == s2$ would return true whenever two empty sets were compared, but otherwise would return false whenever two distinct objects were compared, even if they had the same values as sets. The statement $s1 = s2$ would make $s1$ and $s2$ point into the same data structure; modifications to either set would then change both.

For the same reasons that assignment of abstract types is not allowed, using values of abstract types as parameters or as results is forbidden. References are passed and returned, instead. For example, empset_union takes and returns values of type empset *, rather than empset.

Type checking for abstract types (like that for exposed types) in both the LCL Checker and LCLint is based on type names, not on their representations. However there are two differences in the way LCLint will check the use of abstract types. First, for exposed types, calls to functions from the standard C library will be checked using the representation of the type. For abstract types, names will be used for all type checking. Second, within the implementation of the module exporting an abstract type, the type's representation will be used. This allows the implementation to access the internal structure that is hidden from clients.

The first two functions, empset_init and empset_final are typical of functions found in interfaces exporting abstract types. Since an abstract type cannot be assigned outside its implementation, its variables must be initialized by calling a function in its interface. By convention, an object of an abstract type T is initialized by the T_init function before any other use. LCLint will check for this in the same way it checks for uninitialized variables of exposed types. Once it has been initialized, no reference to it should be passed to T_init again.

A client of empset should call empset_final when it knows that an empset object will never be referenced again. The clause *ensures trashed(*s)* says that upon return from empset_final nothing can be assumed about the storage pointed to by s in the pre state. References to that loc could even cause the client program to crash. A good implementation of empset_final will free storage that is no longer needed, although this specification does not require it to. Since a client has no information about how an empset is represented, it cannot directly free one. For example, if empset is implemented as a pointer to a data structure, the call free(&s1) would free only the pointer, not the data structure.

The third function in the interface, empset_clear, appears to have the same specification as empset_init. However, empset_clear is provided for reinitializing an existing empset, rather than initializing a new one, and LCLint will treat it differently, because it is not the mandatory initialization function. As we will see later, empset_init and empset_clear implemented very differently.

The functions empset_insert and empset_insertUnique both add an employee to an empset. The chief difference is that empset_insertUnique requires that the employee to be added is not already present. This makes it possible to implement the function more efficiently. However, if the requirement is violated, the behavior of empset_insertUnique is totally unconstrained by the specification. The implementation we give later does not check the requirement. If it is violated the implementation returns without complaint, but it breaks a representation invariant—thus leading to unpredictable behavior on subsequent uses of the empset.

The functions empset_union and empset_disjointUnion both return the union of two empsets. Once again, the requires clause makes it possible to implement one more efficiently than the other. Notice that even though `*s1` and `*s2` are not modified, the specifications refer to `(*s1)^` and `(*s2)^`. The `^` is needed because `*s1` and `*s2` refer to locs containing empsets. These must be evaluated in some state to get an empset. Here `*s1` and `*s2` contain the same value in the pre and post states. We use `^` rather than `'` for objects that are guaranteed to have the same values in both states.

Both functions are required (by *fresh(*result)*) to return sets that are not aliased to any objects visible in the pre state. Thus the sets that they return can be modified without affecting the values of other sets. One way of implementing this is to allocate new storage.

The requires clause of empset_choose is necessary to guarantee that the ensures clause is satisfiable. If `(*s)^` is empty, it is not possible to return an employee that is a member of it. Should (*s)^ contain more than one element, the specification is silent as to which member empset_choose returns. The implementation we present later gains efficiency by being abstractly non-deterministic: A single empset value may have many different representations (depending on the order in which its elements were inserted), and the value returned by empset_choose is determined by the representation value passed in.

Although the remaining functions are a necessary part of this interface, they don't illustrate any new LCL features. Its implementation is given in Appendix A, Figures 35 and 36, after we have specified the subsidiary abstractions it uses.

The specifications presented to this point have been in the ASCII form in which they can be entered for checking by the tools. One of the planned tools is a prettyprinter that will take this raw form, and convert it to a more readable form using the capabilities of a modern formatting system and a laser printer or bitmapped display device. Figure 14 shows sample of what its output will look like. The analogous tool for LSL is already in use.

/* empset is a set of employees */
/* set.lsl can be found in an LSL handbook */

imports employee;
abstract type empset;
uses set(employee **for** Elem, empset **for** Set),
    sprint(empset, char[]);

void empset_init(empset $*s$) {
    modifies $*s$;
    ensures $(*s)' = \{\}$;
    }
void empset_final(empset $*s$) {
    modifies $*s$;
    ensures $trashed(*s)$;
    }
void empset_clear(empset $*s$) {
    modifies $*s$;
    ensures $(*s)' = \{\}$;
    }
bool empset_insert(empset $*s$, employee e) {
    modifies $*s$;
    ensures $result = \neg(e \in (*s)^\wedge) \wedge (*s)' = insert(e, (*s)^\wedge)$;
    }
void empset_insertUnique(empset $*s$, employee e) {
    requires $\neg(e \in (*s)^\wedge)$;
    modifies $*s$;
    ensures $(*s)' = insert(e, (*s)^\wedge)$;
    }
bool empset_delete(empset $*s$, employee e) {
    modifies $*s$;
    ensures $result = e \in (*s)^\wedge \wedge (*s)' = delete(e, (*s)^\wedge)$;
    }
empset $*$empset_union(empset $*s1$, empset $*s2$) {
    ensures $(*result)' = (*s1)^\wedge \cup (*s2)^\wedge \wedge fresh(*result)$;
    }

Figure 14: empset.lcl fragment, prettyprinted

3.4 DBase

Up to now we have presented modules by first giving an interface specification, then its auxiliary LSL specification, and finally, its implementation. This works well when the reader has good *a priori* intuition about the meaning of the abstractions used in the interface specification. When such intuition cannot be relied upon, it is often better to present the auxiliary specification first, as we do here.

The definitions in trait dbase, Figure 16 use operators defined by the traits associated with gender and employee. But LSL specifications are programming-language-independent, and hence aren't allowed to reference LCL specifications. We could copy the operator definitions into dbase.lsl, but this would be another opportunity for unchecked discrepancies between parts of the specification. Instead, dbase.lsl, Figure 16, documents them as assumptions. Figure 15, dbaseAssumptions, indicates what must be supplied by any environment in which trait dbase is used. These assumptions are discharged in dbase.lcl by the imports of gender and employee; someday the LCL Checker will make sure that all assumptions are discharged.

Figure 16 introduces operators to create, manipulate, and query dbase values and then provides axioms giving their meanings. Once these operators are understood, it is straightforward to understand the specifications of the functions exported by dbase.lcl, Figure 17 (just as an understanding of the conventional operators on finite sets is the basis for understanding the specifications of the functions in empset.lcl).

The dbase module encapsulates a database and a set of functions to query and manipulate it. It exports two exposed types, dbase_q and dbase_status, and a number of functions. It also contains our first use of global variables. LCL uses the same scope rules as C. However, LCL extends the function prototype by including a list of the *global*

```
dbaseAssumptions: trait

includes integer,
         set(employee for Elem, empset for Set)

gender enumeration of MALE, FEMALE, gender_ANY

job enumeration of MGR, NONMGR, job_ANY

employee tuple of ssNum: int,
                  name: employeeName,
                  salary: int,
                  gen: gender,
                  j: job
```

Figure 15: dbaseAssumptions.lsl

variables referenced by the function. For example, hire is allowed to reference d, but not initNeeded. LCLint will check that each global variable accessed by the function body appears in this list.

As it happens, dbase has only *private* variables, defined for use only in the specification itself. Client code can refer to the functions specified in dbase.lcl, but cannot refer to private types and variables. Furthermore, since they are not exported, the private types and variables need not be implemented. The type dbase is defined only to declare the private variable d. Neither the type dbase nor the variable d appears in our implementation.

Notice that there is no dbase_init function for the private type dbase. Any necessary initialization of the private variable d can be done in dbase_initMod, which has access to the private variables.

The variable initNeeded is used to ensure that dbase_initMod is idempotent. This guarantees that multiple clients can use the data base, and can each call dbase_initMod, to ensure that the data base is initialized, without worrying about interfering with one another.

The function hire is closely related to the operator hire of dbase.lsl. The difference is that it does some error checking and returns a result indicating the outcome of this checking.

The function uncheckedHire is even more similar to the LSL operator, since it does no

```
dbase: trait

assumes dbaseAssumptions

dbase_q tuple of g:gender, j: job, l: int, h: int
dbase_status enumeration of dbase_OK, salERR, genderERR,
                            jobERR, duplERR
introduces
   new: -> dbase
   hire: dbase, employee -> dbase
   fire, promote: dbase, int -> dbase
   setSal: dbase, int, int -> dbase
   find: dbase, int -> employee
   employed: dbase, int -> bool
   numEmployees: dbase -> int
   match: gender, gender -> bool
   match: job, job -> bool
   query: dbase, dbase_q -> empset
```

Figure 16: dbase.lsl, part 1

```
asserts
   dbase generated by new, hire
   dbase partitioned by query
   forall e: employee, k: int, g, gq: gender, j, jq: job,
         q: dbase_q, sal: int, d: dbase
     fire(new, k) == new;
     fire(hire(d, e), k) ==
        if e.ssNum = k then fire(d, k) else hire(fire(d, k), e);
     promote(new, k) == new;
     promote(hire(d, e), k) ==
         if e.ssNum = k
         then hire(promote(d, k), set_j(e, MGR))
         else hire(promote(d, k), e);
     setSal(new, k, sal) == new;
     setSal(hire(d, e), k, sal) ==
         if e.ssNum = k
         then hire(setSal(d, k, sal), set_salary(e, sal))
         else hire(setSal(d, k, sal), e);
     find(hire(d, e), k) == if e.ssNum = k then e else find(d, k);
     employed(new, k) == false;
     employed(hire(d, e), k) ==
          if e.ssNum = k then true else employed(d, k);
     numEmployees(new) == 0;
     numEmployees(hire(d, e)) == numEmployees(d)
           + (if employed(d, e.ssNum) then 0 else 1);
     match(gq, g) == gq = gender_ANY \/ g = gq;
     match(jq, j) == jq = job_ANY \/ j = jq;
     query(new, q) == { };
     query(hire(d, e), q) ==
        if match(q.g, e.gen) /\ match(q.j, e.j)
           /\ q.l <= e.salary /\ e.salary <= q.h
        then insert(e, query(d, q)) else query(d, q)
```

Figure 16: dbase.lsl, part 2

```
imports employee, gender, empset;

typedef struct{gender g; job j; int l; int h;} dbase_q;
typedef enum {dbase_OK, salERR, genderERR, jobERR,
              duplERR} dbase_status;
private abstract type dbase;
private dbase d;
private bool initNeeded = true;

uses dbase, sprint(dbase, char[]);

dbase_status hire(employee e) dbase d; {
   modifies d;
   ensures
     (if re-
sult = dbase_OK then d' = hire(d^, e) else unchanged(d))
      /\ result = (if e.gen = gender_ANY then genderERR
                   else if e.j = job_ANY then jobERR
                   else if e.salary < 0 then salERR
                   else if employed(d^, e.ssNum) then duplERR
                   else dbase_OK);
   }
void uncheckedHire(employee e) dbase d; {
   requires e.gen != gender_ANY /\ e.j != job_ANY
            /\ e.salary > 0 /\ not(employed(d^, e.ssNum));
   modifies d;
   ensures d' = hire(d^, e);
   }
bool fire(int ssNum) dbase d; {
   modifies d;
   ensures result = employed(d^, ssNum)
                    /\ (if result then d' = fire(d^, ssNum)
                        else unchanged(d));
   }
```

Figure 17: dbase.lcl, part 1

```
int query(dbase_q q, empset *s) dbase d; {
   modifies *s;
   ensures (*s)' = (*s)^ \union query(d^, q)
           /\ result = size((*s)' - (*s)^);
   }
bool promote(int ssNum) dbase d; {
   modifies d;
   ensures result = (employed(d^, ssNum)
           /\ find(d^, ssNum).j = NONMGR)
           /\ (if result then d' = promote(d^, ssNum)
               else unchanged(d));
   }
bool setSalary(int ssNum, int sal) dbase d; {
    modifies d;
    ensures result = employed(d^, ssNum)
            /\ (if result then d' = setSal(d^, ssNum, sal)
                else unchanged(d));
    }
int dbase_sprint(char s[]) dbase d; {
    requires
       maxIndex(s) >= (numEmploy-
ees(d^) * employee_maxPrintSize);
    modifies s;
    ensures isSprint(s', d^)
           /\ result = lenStr(s')
           /\ result <= (numEmploy-
ees(d^) * employee_maxPrintSize);
    }
void dbase_initMod(void) dbase d; bool initNeeded; {
   modifies d, initNeeded;
   ensures if initNeeded^
           then d' = new /\ not(initNeeded') else unchanged(all);
  }
```

Figure 17: dbase.lcl, part 2

```
#include "bool.h"
#include "gender.h"
#include "employee.h"
#include "empset.h"

typedef enum {dbase_OK,
              salERR,
              genderERR,
              jobERR,
              duplERR} dbase_status;
typedef struct{gender g; job j; int l; int h;} dbase_q;

extern dbase_status hire(employee e);
extern void uncheckedHire(employee e);
extern bool fire(int ssNum);
extern int query(dbase_q q, empset *s);
extern bool promote(int ssNum);
extern bool setSalary(int ssNum, int sal);
extern int dbase_sprint(char s[]);
extern void dbase_initMod(void);
```

Figure 18: dbase.lh

error checking. Of course, if it is called when its requires clause does not hold, it is likely to do something unfortunate that may not be detected for quite some time, for example, when the employee is fired. Both functions modify the private variable d. Since d is a global variable rather than a formal parameter, it can be accessed directly; there is no need to pass in a pointer to it.

The function query is also closely related to the LSL operator query. But the operator returns an empset and the function returns an int: the number of employees added to `*s` as the required *side effect* of calling it. This is a common C idiom.

Now we can show that dbase preserves the property that there is at most one employee in d with any given ssNum. The function dbase_initMod ensures that d starts out empty. The only functions that are allowed to add employees to d are hire and uncheckedHire. If hire is called with an employee whose ssNum is already in d, its specification says that it must return duplERR and leave d unchanged. And uncheckedHire's requires clause forbids calling it with an employee whose ssNum is already in d—any subsequent havoc is purely the responsibility of uncheckedHire's client.

The only thing of note about dbase.lh, Figure 18, is that the private variables and private type do not appear in it.

Once again, we defer the presentation of dbase's implementation to Appendix A, Figures 37 and 38, since it also relies on subsidiary abstractions to be specified later.

3.5 Driver

Before looking at the abstractions used in the implementation of dbase, we pause to take a look at some code that uses dbase. Figure 19 is part of a program we used to test our implementations of the modules specified earlier in this section.

The program begins with a series of #includes of the .h-files for the modules containing functions or types that it uses directly. It does not include any subsidiary modules that they may use. While the included .h-files are necessary to compile the driver, to understand the code one need look only at the corresponding .lcl-files. If the implementation of one of the used modules, such as empset, should change, the driver will have to be re-compiled, but the code will not have to be changed.

After declaring some variables, the driver initializes the included modules (except for stdio). LCLint will issue a warning if this initialization is not done immediately following the declarations of the function main. Since the author of main has no way of knowing what modules are used in the implementations of the included modules, the various initMod functions must themselves call the initMod functions of the modules they use. This could result in some initMod functions being called twice, which is why their specifications typically require them to be idempotent.

The driver then initializes the the variable es. Our conventions require this because empset is an abstract type. LCLint will issue a warning if a locally declared variable of an abstract type isn't initialized immediately following the module initializations.

Finally, the driver calls some of the specified functions. Effects that are fully constrained by specifications, such as the result returned by fire, are checked internally. Where the specification allows a variety of acceptable effects, output is printed so it can be checked by hand or by a test harness (against previous runs).

```
#include <stdio.h>
#include "bool.h"
#include "gender.h"
#include "employee.h"
#include "empset.h"
#include "dbase.h"

int main(int argc, char *argv[]) {

   employee e;
   empset es;
   empset *emptr;
   char na[10000];
   int i, j;
   dbase_status stat;
   dbase_q q;

/* Initialize the LCL-specified modules that were included */
   bool_initMod();
   gender_initMod();
   employee_initMod();
   empset_initMod();
   dbase_initMod();

/* Initialize all of the variables of abstract types */
   empset_init(&es);
```

Figure 19: drive.c fragment, part 1

```
/* Perform tests */
   for (i = 0; i < 20; i++) {
     e.ssNum = i;
     e.salary = 1000 * i;
     if (i < 10) e.gen = MALE; else e.gen = FEMALE;
     if (i < 15) e.j = NONMGR; else e.j = MGR;
     (void) sprintf(na, "J. Doe %d", i);
     employee_setName(&e, na);
     if ( (i/2)*2 == i) hire(e);
        else {uncheckedHire(e); stat = hire(e);}
     }
   if (stat == duplERR) printf("Error 1: Duplicate not found\n");

   (void) dbase_sprint(na);
   printf("Should print 20 employees:\n%s\n", na);

   dbase_initMod();  /* Should have no effect */

   if (!fire(17)) printf("Error 2: 17 not fired\n");

   q.g = FEMALE; q.j = job_ANY; q.l = 15800; q.h = 18500;
   if ((i = query(q, &es)) != 2)
      printf("Error 3: Wrong number found %d\n", i);
   (void) empset_sprint(na, &es);
   printf("Should print two employees: \n%s\n", na);

   /* ... */
}
```

Figure 19: drive.c fragment, part 2

```
imports employee;

constant int eref_maxPrintSize = employee_maxPrintSize + 27;

abstract type employee ref eref;

uses sprint(eref, char[]);

int eref_sprint(char s[], eref er) {
    requires maxIndex(s) >= eref_maxPrintSize;
    modifies s;
    ensures isSprint(s', er)
            /\  result = lenStr(s') /\ re-
sult <= eref_maxPrintSize;
    }
void eref_initMod(void) {
    ensures true;
    }
```

Figure 20: eref.lcl

3.6 Eref

Now we move down a level of abstraction, and specify some modules that are useful in implementing the modules defined above. The next example introduces a new kind of type constructor. The constructor *ref* is a more abstract version of the `*` used in exposed types. Like all abstract types, values of ref types can be accessed only through the functions exported from the interface in which they are declared. Unlike other abstract types, however, the interface implicitly exports a constant and four functions (type_alloc, type_free, type_set, and type_get) in addition to those explicitly specified in the .lcl-file. The functions correspond to builtin operations on C pointers. Since their meaning is determined by LCL, they do not appear explicitly in the .lcl-file, but they must be implemented.

Figure 20 exports a ref type, eref. Figure 21 specifies its four implicitly exported functions, using a subset of LCL's pointer operations. The functions eref_free, eref_set and eref_get have unconstrained behavior when they are called with erefNIL.

Unlike LCL's other abstract types, ref types can be assigned using =, passed as parameters, returned from functions, and compared using ==. Since they can be assigned, there is no need for the type initialization function that must be provided for other abstract types.

Refs can be used in much the same way as pointers: to create sharing in data structures, to assign large objects inexpensively and pass them into and out of functions, to check

```
imports eref;

constant eref erefNIL = NIL;

eref eref_alloc(void) {
   ensures fresh(*result);
   }
void eref_free(eref er) {
   requires er != erefNIL;
   modifies *er;
   ensures trashed(*er);
   }
void eref_set(eref er, employee e) {
   requires er != erefNIL;
   modifies *er;
   ensures (*er)' = e;
   }
employee eref_get(eref er) {
   requires er != erefNIL;
   ensures result = (*er)^;
   }
```

Figure 21: eref.lcl's implied interface

inexpensively whether two objects are the same, to handle data structures whose size varies dynamically, etc.

There are some operations on pointers that are not available for ref types. There is no arithmetic on ref types. Although LCL allows the use of `*` and `->` on ref types in specifications, LCLint won't allow their use on ref types in client code. Instead, clients must use the functions exported by the interface.

Though ref types are more limited than pointer types, using them has some advantages:

- It provides a level of abstraction. The implementor can change the implementation, e.g., from a pointer to an index into an array, without worrying about invalidating client code.
- It allows private storage management, even if the chosen representation is a pointer. For example, a compacting storage manager can be written, since all access must be via functions in the module.
- It is more general, allowing references to data that is in another address space, on another machine, on a disk, etc.

```
#include "bool.h"
#include "employee.h"

extern eref eref_alloc(void);
extern void eref_free(eref er);
extern void eref_set(eref er, employee e);
extern employee eref_get(eref er);
extern int eref_sprint(char s[], eref er);
extern void eref_initMod(void);
```

Figure 22: eref.lh

```
#if !defined(eref_h_expanded)
#define eref_h_expanded

#define eref_maxPrintSize (employee_maxPrintSize + 27)

#include "employee.h"

typedef int eref;

/* Private type defs used in macros.  */
typedef enum {used, avail} eref_status;
typedef struct {employee * conts;
                eref_status * status;
                int size;
                int index;} eref_ERP;

extern eref_ERP eref_Pool;

#include "eref.lh"

#define erefNIL (-1)
#define eref_free(er)  (eref_Pool.status[er] = avail)
#define eref_set(er, e) (eref_Pool.conts[er] = e)
#define eref_get(er) (eref_Pool.conts[er])
#endif
```

Figure 23: eref.h

Figure 22 contains the .lh-file generated by LCL from eref.lcl. Notice that it includes prototypes for the implicit functions.

Figures 23 and 24 contain an implementation of eref. It is not a particularly interesting implementation, but it does show that there is considerable freedom in implementing ref types. The only constraints are that the top-level representation (int here) is assignable and comparable (using ==), and that implementations of the exported functions meet their specifications. Because the implementation variable eref_Pool is used in three macro definitions, C requires it to be declared *extern* in eref.h, even though clients of eref are not supposed to reference it—or even know about its existence, since it doesn't appear in eref.lcl.

```
#include <stdio.h>
#include "eref.h"

eref_ERP eref_Pool;     /* private */
static bool needsInit = TRUE;  /* private */

eref eref_alloc(void) {
  int i, res;
  int * tmp;

  for (i=0;
       (eref_Pool.status[i] == used) && (i < eref_Pool.size);
       i++);
  res = i;
  eref_Pool.status[res] = used;
  if (res == eref_Pool.size - 1) {
    eref_Pool.conts =
     (employee *) realloc(eref_Pool.conts,
                          2*eref_Pool.size*sizeof(employee));
    eref_Pool.status =
     (eref_status *) realloc(eref_Pool.status,
                        2*eref_Pool.size*sizeof(eref_status));
    eref_Pool.size = 2*eref_Pool.size;
    for (i = res+1; i < eref_Pool.size; i++)
        eref_Pool.status[i] = avail;
  }
  return (eref) res;
}
int eref_sprint(char s[], eref er) {
  int len;
  (void) sprintf(s, "eref: %d. Employee: ", (int) er);
  len = strlen(s);
  return len + employee_sprint(&(s[len]), eref_get(er));
}
```

Figure 24: eref.c, part 1

```
void eref_initMod(void) {
  int i;
  const int size = 2;

 /* So that initMod will be idempotent  */
  if (needsInit == FALSE) return;
  needsInit = FALSE;

  bool_initMod();
  employee_initMod();
  eref_Pool.conts = (employee *) malloc(size*sizeof(employee));
  eref_Pool.status =
              (eref_status *) malloc(size*sizeof(eref_status));
  eref_Pool.size = size;
  eref_Pool.index = 0;
  for (i = 0; i < size; i++) eref_Pool.status[i] = avail;
}
```

Figure 24: eref.c, part 2

3.7 Erc

Figure 27 specifies functions operating on an abstract type, erc (for employee ref collection), that contains erefs. An erc is basically a bag with a pair of functions that make it possible to iterate over its elements. It is used in the implementation of empset and dbase.

The iteration functions add some complexity to the specification. This shows up most notably in erc.lsl, Figure 25. The partitioned by clause indicates that erc values can be viewed as pairs of bags; the relevant portions of bag.lsl are given in Figure 26, and the complete trait appears in Appendix B, Figure 43. The operator val maps an erc to the bag of erefs that have been inserted (and not deleted). The operator wereYielded maps an erc to the bag of values that have been marked as yielded (by the yield operator). The derived operator toYield maps an erc to the bag of values that remain to be yielded. These operators are used in the specification of the functions erc_iterStart and erc_yield.

```
erc: trait
  includes bag(eref, erefBag)

  introduces
    { }: -> erc
    add, yield, delete: eref, erc -> erc
    val, wereYielded, toYield: erc -> erefBag
    __ \in __: eref, erc -> Bool

   asserts
     erc generated by { }, add, yield
     erc partitioned by val, wereYielded
     forall ic: erc, e, e1: eref
       val({ }) == { };
       val(add(e, ic)) == insert(e, val(ic));
       val(yield(e, ic)) == val(ic);
       val(delete(e, ic)) == delete(e, val(ic));

       wereYielded({ }) == { };
       wereYielded(add(e, ic)) == wereYielded(ic);
       wereYielded(yield(e, ic)) == insert(e, wereYielded(ic));
       wereYielded(delete(e, ic)) == delete(e, wereYielded(ic));

       toYield(ic) == val(ic) - wereYielded(ic);

       e \in ic == e \in val(ic)
```

Figure 25: erc.lsl

```
bag(Elem, Bag): trait
  introduces
    { }: -> Bag
    insert, delete: Elem, Bag -> Bag
    __ \in __: Elem, Bag -> Bool
    {__}: Elem -> Bag
    __ \union __, __ - __: Bag, Bag -> Bag
    % ...

   asserts
     Bag generated by { }, insert
     Bag partitioned by \in, delete
     forall e, e1: Elem, b, b1: Bag
       {e} == insert(e, { });
       delete(e, { }) == { };
       delete(e, insert(e1, b)) ==
              if e = e1 then b else insert(e1, delete(e, b));
       not(e \in { });
       e \in insert(e1, b) == e = e1 \/ e \in b;
       b \union { } == b;
       b \union insert(e, b1) == insert(e, b) \union b1;
       b - { } == b;
       b - insert(e, b1) == delete(e, b - b1)
       % ...
```

Figure 26: bag.lsl fragment

Typically, client code that uses these functions will be of the form,

```
eref er;
erc s;
  . . .
for(er = erc_iterStart(s);
    er != erefNIL;
    er = erc_yield(s)) {
       Body of loop that does something with each er from s.
    }
```

If the body of the loop were guaranteed not to change the erc being iterated over, both the specification and the implementation of erc could be considerably simplified. However, such a restriction is not usually reasonable. Allowing for modifications within the body of the loop raises several questions about the semantics of the functions, among them,

- If an element is inserted in the erc within the body of the loop will it be yielded?
- If, within the body of the loop, an element gets deleted before it has been yielded

does it get yielded?

- If, within the body of the loop, an element gets yielded, deleted and then reinserted, does it get yielded again?

The answers, according to this specification, are Yes, No, and Yes, respectively.

Again, the implementation is deferred to Appendix A, Figures 33 and 34.

3.8 Ereftab

Ereftab, Figures 28 and 29, is the last module in our example. It is used to create a one-to-one mapping from employees to erefs. It makes it unnecessary to store multiple copies of the same employee record within the implementation of empset.

The intended use of ereftab_insert is to put an employee in the table only after a lookup

```
imports eref;

abstract type erc;

uses erc, sprint(erc, char[]);

void erc_init(erc *c) {
   modifies *c;
   ensures (*c)' = { };
   }
void erc_clear(erc *c) {
   modifies *c;
   ensures (*c)' = { };
   }
void erc_insert(erc *c, eref er) {
   modifies *c;
   ensures (*c)' = add(er, (*c)^);
   }
bool erc_delete(erc *c, eref er) {
   modifies *c;
   ensures result = er \in (*c)^
           /\ (*c)' = delete(er, (*c)^);
   }
bool erc_member(eref er, erc *c) {
   ensures result = er \in (*c)^;
}
```

Figure 27: erc.lcl, part 1

```
eref erc_iterStart(erc *c) {
   modifies *c;
   ensures if val((*c)^) = { }
             then result = erefNIL /\ unchanged(*c)
             else result \in val((*c)^)
                 /\ val((*c)') = val((*c)^)
                 /\ wereYielded((*c)') = {result};
   }
eref erc_yield(erc *c) {
   modifies *c;
   ensures if toYield((*c)^) = { }
             then result = erefNIL /\ unchanged(*c)
             else result \in toYield((*c)^)
                   /\ (*c)'= yield(result, (*c)^);
   }
void erc_join(erc *c1, erc *c2) {
    modifies *c1;
    ensures val((*c1)') = val((*c1)^) \union val((*c2)^)
            /\ wereYielded((*c1)') = { };
    }
int erc_sprint(char s[], erc *c) {
    requires maxIn-
dex(s) >= (size(val((*c)^)) * eref_maxPrintSize);
    modifies s;
    ensures isSprint(s', (*c)^)
            /\ result = lenStr(s')
            /\ re-
sult <= (size(val((*c)^)) * eref_maxPrintSize);
    }
void erc_initMod(void) {
    ensures true;
    }
void erc_final(erc *c) {
   modifies *c;
   ensures trashed(*c);
   }
```

Figure 27: erc.lcl, part 2

has failed to find an eref for that employee. The requires clause of ereftab_insert formalizes this property, and allows the implementation not to duplicate a test that has just been made by the client.

The implementation of ereftab is unremarkable, and is not presented.

```
imports employee, eref;

abstract type ereftab;

uses ereftab, sprint(ereftab, char[]);

void ereftab_init(ereftab *t) {
   modifies *t;
   ensures (*t)' = empty;
   }
eref ereftab_insert(ereftab *t, employee e) {
   requires getERef((*t)^, e) = erefNIL;
   modifies *t;
   ensures (*t)' = add((*t)^, e, result) /\ fresh(*result);
   }
bool ereftab_delete(ereftab *t, eref er) {
   modifies *t, *er;
   ensures result = in((*t)^, er)
        /\ (if result
            then (*t)' = delete((*t)^, er) /\ trashed(*er)
            else unchanged(*t, *er));
   }
```

Figure 28: ereftab.lcl, part 1

```
eref ereftab_lookup(employee e, ereftab *t) {
   ensures result = getERef((*t)^, e);
   }
int ereftab_sprint(char s[], ereftab *t) {
    requires maxIndex(s) >= (size((*t)^) * eref_maxPrintSize);
    modifies s;
    ensures isSprint(s', (*t)^)
            /\ result = lenStr(s')
            /\ result <= (size((*t)^) * eref_maxPrintSize);
    }
void ereftab_final(ereftab *t) {
   modifies *t, reach((*t)^);
   ensures trashed(*t)
           /\ \forall e:employee
              ((getERef((*t)^, e) != erefNIL)
               => trashed(*getERef((*t)^, e)));
   }
void ereftab_initMod(void) {
    ensures true;
    }
```

Figure 28: ereftab.lcl, part 2

```
ereftab: trait

includes integer

introduces
   empty: -> ereftab
   add: ereftab, employee, eref -> ereftab
   delete: ereftab, eref -> ereftab
   getERef: ereftab, employee -> eref
   erefNIL: -> eref
   in: ereftab, eref -> bool
   size: ereftab -> int

asserts
   ereftab generated by empty, add
   ereftab partitioned by getERef

   forall e, e1: employee, er, er1: eref, t: ereftab
     delete(empty, er) == empty;
     delete(add(t, e, er), er1) ==
        if er = er1 then t else add(delete(t, er1), e, er);

     in(empty, er) == false;
     in(add(t, e, er), er1) == er = er1 \/ in(t, er);

     getERef(empty, e1) == erefNIL;
     getERef(add(t, e, er), e1) ==
        if e = e1 then er else getERef(t, e1);

     size(empty) == 0;
     size(add(t, e, er)) == 1 + (if in(t, er) then 0 else 1)
```

Figure 29: ereftab.lsl

```
typedef struct _elem{eref val; struct _elem *next;} ercElem;
typedef ercElem * ercSet;
typedef struct {ercSet vals; ercSet nextY; ercSet prevY;} erc;
```

Figure 30: erc's representation

```
typedef erc empset;   /* This is in empset.h  */

ereftab known; /* This is in empset.c */
```

Figure 31: empset's representation

3.9 Notes on the implementations

Appendix A contains implementations of the interfaces erc, empset, and dbase. We include them not because they are intrinsically interesting, but for completeness. Here we take opportunity to make some comments about the relationship of the specifications to these implementations.

In writing specifications, the emphasis is entirely on ease of understanding. Code should be reasonably easy to understand, but efficiency must also be considered. Consider, for example, the representation for ercs, Figure 30. Though the specification is written as if an erc consists of a pair of bags, the implementation uses a single linked list and three pointers into it. The pointer val points to the head of the list, prevY to the most recently yielded element, and nextY to the element to be yielded next. Within erc.c, erc is treated as an exposed type, that is, erc values are treated as structs. LCLint will allow this exposure within the implementation of an interface, even though it will generate an error message if client code attempts to treat an erc as a struct.

The implementation of empset uses an erc to represent an empset, Figure 31. It also uses a non-exported module-level variable, known, of type ereftab, declared in empset.c. Known is used to avoid allocating space for the same employee multiple times. The first time an employee is inserted into any empset it is also inserted into known and a newly allocated eref is inserted into the erc. On subsequent inserts of the same employee into any empset, the old eref is reused. This auxiliary data structure is shared by the implementation of all objects of type empset, but this sharing is not visible to clients.

The implementation of dbase is considerably longer than that of the other modules specified here. It is also somewhat different in structure. Unlike empset.h and erc.h, dbase.h contains no typedef (though it does inherit typedefs, of exposed type, from dbase.lh). This is because dbase.lcl exports no abstract types and the implementation of dbase doesn't use any macros that depend on locally defined types. Information pertinent to compiling only the implementation itself is restricted to dbase.c, Figure 32.

```
#define firstERC mMGRS
#define lastERC fNON
#define numERCS (lastERC - firstERC + 1)

typedef enum {mMGRS, fMGRS, mNON, fNON} employeeKinds;

erc db[numERCS];

/* Invariant: The data base is partitioned by
   val(db[mMGRS]), val(db[mNON]), val(db[fMGRS]), val(db[fNON]) */

bool initDone = FALSE;
```

Figure 32: dbase.c fragment

The private variables d and initNeeded from dbase.lcl are implemented by the variables db and initDone, in dbase.c. We chose different names for the variables in the implementation to emphasize that there is no necessary correspondence between module-level variables appearing in the implementation and private variables appearing in the specification. It is purely accidental that both of our private specification variables correspond to single implementation variables; one of our earlier implementations of the interface used four distinct ercs to represent d.

The correctness of the implementations of the functions in dbase.c depends upon the maintenance of a representation invariant. That this holds can be shown by an inductive argument:

- It is established by dbase_initMod,
- For each function specified in dbase.lcl, if the invariant and the requires clause hold on entry, the invariant will hold upon termination. In discharging this step of the proof, it is necessary to examine even those functions whose specification does not allow them to modify d, since they might still modify the representation of d.

The implementation of dbase includes several functions that do not appear in dbase.lcl and therefore are not accessible to clients. It would be acceptable for these functions to break the invariant temporarily (though, in fact, they don't).

4 Summary

We have tried to present enough information to allow the C programmer to begin to use LCL. Our example specifications demonstrate most features of the language. Our example implementations illustrate a style of C programming in which specifications are used to establish firewalls between modules.

People writing client programs need look only at the specifications to discover what they need to know about the functional behavior of the modules that they use. This saves them the trouble of examining the code (which, even given our rather simplistic implementations, is considerably longer than the specifications). Furthermore, it increases the likelihood of client programs continuing to work despite changes to the implementations of of modules that they are built on.

LCL 1.0 is not sufficiently expressive to to specify all reasonable modules that one might implement in C. For example, there is no provision made for function parameters and no treatment of concurrency. We expect to address both of these in a future version.

Despite these omissions, we feel that LCL 1.0 is ready for practical use. Many modules of most programs can be well-specified using LCL. The LSL and LCL checkers, though still under development, have proved extremely useful in early trials. We don't yet have an LCLint. However, hand simulations of the checks planned for LCLint indicate that such a tool, combined with careful specifications, can uncover a large number of typical errors.

5 Acknowledgements

Bill McKeeman provided the impetus for the design of LCL by insistently reminding us that Larch should be applied to "a language that real programmers take seriously." He also helped us recruit Joe Wild and Gary Feldman, whose interest, intellectual involvement, and hard work made LCL and the LCL Checker real. Jeannette Wing designed the first Larch interface language (for CLU); Kevin Jones designed one for Modula-3 concurrent with the design of LCL; Steve Garland implemented early versions of the LSL and LCL Checkers. All three participated in the design and debugging of the LCL language.

Mike Burrows reviewed an earlier version of this report and contributed many substantive (and useful) comments and suggestions. Cynthia Hibbard did her usual thorough copy editing job. Their efforts are greatly appreciated.

LCL is built on a foundation of many years of work on Larch. We have been helped by too many people to list them all, but here are a few: Martin Abadi, Andy Gordon, Daniel Jackson, Leslie Lamport, Butler Lampson, Andrés Modet (who constructed the first complete LSL Checker), Jim Saxe, and Mark Vandevoorde. Other colleagues at MIT and DEC/SRC and the members of IFIP Working Group 2.3 have provided constructively critical sounding boards for many versions of our ideas.

Our work has also benefited from our study of other specification languages. Since this report is primarily a description of LCL, we have not included references to related work that has influenced us. References to this work can be found in the papers cited

in section 6.

Finally, we are grateful to Bob Taylor, Sam Fuller, and Becky Will for believing in and supporting this work.

6 References

[ANSI] American National Standards Institute, *American National Standard for Information Systems—Programming Language C*.

[Garland et al. 90] Stephen J. Garland, John V. Guttag, and James J. Horning, "Debugging Larch Shared Language Specifications," *IEEE Trans. Software Engineering*, vol. 16, no. 9, pp. 1044–1057, September 1990.

[Guttag et al. 85] J.V. Guttag, J.J. Horning, and J.M. Wing, "The Larch Family of Specification Languages," *IEEE Software*, vol. 2, no. 5, pp.24–36, September 1985.

[Guttag et al. 90] John V. Guttag, James J. Horning, and Andrés Modet, *Report on the Larch Shared Language: Version 2.3*, Digital Equipment Corporation, Systems Research Center Research Report 58, April 14, 1990.

[Jones 91] Kevin D. Jones, *LM3: A Larch Interface Language for Modula-3. A Definition and Introduction. Version 1.0*, Digital Equipment Corporation, Systems Research Center Research Report 72, June 10, 1991.

A Implementations

This appendix contains the implementations of the interfaces erc, empset, and dbase. We present them not because they are intrinsically interesting, but for completeness.

```
#if !defined(erc_h_expanded)
#define erc_h_expanded

#include "eref.h"

typedef struct _elem{eref val; struct _elem *next;} ercElem;
typedef ercElem * ercSet;
typedef struct {ercSet vals; ercSet nextY; ercSet prevY;} erc;

#include "erc.lh"

#define erc_initMod( ) do {bool_initMod(); employee_initMod();\
                           eref_initMod();} while (0)
#endif
```

Figure 33: erc.h

```
#include "erc.h"
#define ercSetEmpty ((ercSet) 0)

void erc_init(erc *c) {
  c->vals = ercSetEmpty;
  c->nextY = ercSetEmpty;
  c->prevY = ercSetEmpty;
}
void erc_final(erc *c) {
  ercSet elem;
  ercSet prevElem;

  if (c->vals == ercSetEmpty) return;
  elem = c->vals;
  while (elem->next != ercSetEmpty) {
    prevElem = elem;
    elem = elem->next;
    free(prevElem);
    }
  free(elem);
}
void erc_clear(erc *c) {
  erc_final(c);
  erc_init(c);
}
void erc_insert(erc *c, eref er) {
  ercSet newElem;

  newElem = (ercElem *) malloc(sizeof(ercElem));
  newElem->val = er;
  newElem->next = c->nextY;
  if (c->prevY != ercSetEmpty) (c->prevY)->next = newElem;
  if (c->vals == c->nextY) c->vals = newElem;
  c->nextY = newElem;
}
```

Figure 34: erc.c, part 1

```
bool erc_member(eref er, erc *c) {
  ercSet tmpc;

  tmpc = c->vals;
  while (tmpc != ercSetEmpty) {
    if (tmpc->val == er) return TRUE;
    tmpc = tmpc->next;
  }
  return FALSE;
}
bool erc_delete(erc *c, eref er) {
  ercSet elem;
  ercSet prevElem;

  if (c->vals == ercSetEmpty) return FALSE;
  elem = c->vals;
  if (elem->val == er) {
     if (c->nextY == c->vals) c->nextY = elem->next;
     if (c->prevY == c->vals) c->prevY = ercSetEmpty;
     c->vals = elem->next;
     free(elem);
     return TRUE;
   }
  while (elem->next != ercSetEmpty) {
    prevElem = elem;
    elem = elem->next;
    if (elem->val == er) {
       if (c->nextY == elem) c->nextY = elem->next;
       if (c->prevY == elem) c->prevY = prevElem;
       prevElem->next = elem->next;
       free(elem);
       return TRUE;
     }
  }
  return FALSE;
}
```

Figure 34: erc.c, part 2

```
eref erc_iterStart(erc *c) {
  c->nextY = c->vals;
  c->prevY = ercSetEmpty;
  return erc_yield(c);
}
eref erc_yield(erc *c) {
  eref res;
  if (c->nextY == ercSetEmpty) return erefNIL;
  res = (c->nextY)->val;
  c->prevY = c->nextY;
  c->nextY = (c->nextY)->next;
  return res;
}
void erc_join(erc *c1, erc *c2) {
  ercSet tmpc2;

  tmpc2 = c2->vals;
  while (tmpc2 != ercSetEmpty) {
    erc_insert(c1, tmpc2->val);
    tmpc2 = tmpc2->next;
  }
}
int erc_sprint(char s[], erc *c) {
  int len;
  ercSet tmpc;

  tmpc = c->vals;
  len = 0;
  while (tmpc != ercSetEmpty) {
    if (tmpc->val != erefNIL) {
       len += employee_sprint(&(s[len]), eref_get(tmpc->val));
       s[len] = '\n';
       len++;
       }
    tmpc = tmpc->next;
  }
  s[len] = '\0';
  return len;
}
```

Figure 34: erc.c, part 3

```
#if !defined(empset_h_expanded)
#define empset_h_expanded

#include "eref.h"
#include "erc.h"
#include "ereftab.h"

typedef erc empset;

#include "empset.lh"

#define empset_init(s)  erc_init(s)
#define empset_final(s) erc_final(s)
#endif
```

Figure 35: empset.h

```
#include "empset.h"

ereftab known; /* Table of employees that have been put in sets  */

eref _empset_get(employee e, erc *s) {
  eref er;
  employee e1;

  for(er = erc_iterStart(s); er != erefNIL; er = erc_yield(s)) {
    e1 = eref_get(er);
    if ((e1.ssNum == e.ssNum) && (e1.gen == e.gen) &&
          (e1.j == e.j) && strcmp(e1.name, e.name)) return er;
    }
  return erefNIL;
}
void empset_clear(empset *s) {
  eref er;

  for(er = erc_iterStart(s); er != erefNIL; er = erc_yield(s))
     eref_free(er);
  erc_clear(s);
}
bool empset_insert(empset *s, employee e) {
  eref er;

  er = _empset_get(e, s);
  if (er != erefNIL) return FALSE;
  er = ereftab_lookup(e, &known);
  if (er == erefNIL) er = ereftab_insert(&known, e);
  erc_insert(s, er);
  return TRUE;
  }
void empset_insertUnique(empset *s, employee e) {
  eref er;

  er = ereftab_lookup(e, &known);
  if (er == erefNIL) er = ereftab_insert(&known, e);
  erc_insert(s, er);
  }
```

Figure 36: empset.c, part 1

```
bool empset_delete(empset *s, employee e) {
  eref er;

  er = _empset_get(e, s);
  if (er == erefNIL) return FALSE;
  erc_delete(s, er);
  return TRUE;
  }
empset *empset_union(empset *s1, empset *s2) {
  erc *em;
  eref er;

  em = (erc *) malloc(sizeof(erc));
  erc_init(em);
  erc_join(em, s1);
  for (er = erc_iterStart(s2); er != erefNIL; er = erc_yield(s2))
    empset_insert(em, eref_get(er));
  return em;
}
empset *empset_disjointUnion(empset *s1, empset *s2) {
  erc *em;

  em = (erc *) malloc(sizeof(erc));
  erc_init(em);
  erc_join(em, s1);
  erc_join(em, s2);
  return em;
}
void empset_intersect(empset *s1, empset *s2) {
  eref er1, er2;

  for (er1 = erc_iterStart(s1); er1 != erefNIL; er1 = erc_yield(s1))
    for (er2 = erc_iterStart(s2); er2 != erefNIL; er2 = erc_yield(s2))
      if ((er1 == er2) || eref_get(er1).ssNum == eref_get(er2).ssNum) {
        erc_delete(s1, er1);
        return;
     }
}
```

Figure 36: empset.c, part 2

```
int empset_size(empset *s) {
  int size;
  eref er;

  size = 0;
  for(er = erc_iterStart(s); er != erefNIL; er = erc_yield(s))
      size ++;
  return (size);
}
bool empset_member(employee e, empset *s) {
  employee e1;

  return _empset_get(e, s) != erefNIL;
}
bool empset_subset(empset *s1, empset *s2) {
  employee e;
  eref er;

  for (er = erc_iterStart(s1); er != erefNIL; er = erc_yield(s1))
    if (empset_member(eref_get(er), s2) == FALSE) return FALSE;
  return TRUE;
  }
employee empset_choose(empset *s) {
   return eref_get(erc_iterStart(s));
   }
int empset_sprint(char s[], empset *es) {
  int len;
  eref er;

  len = 0;
  for (er = erc_iterStart(es); er != erefNIL; er = erc_yield(es)) {
     len += employee_sprint(&(s[len]), eref_get(er));
     s[len] = '\n';
     len++;
     }
   s[len] = '\0';
}
```

Figure 36: empset.c, part 3

```
void empset_initMod(void) {
  bool_initMod();
  employee_initMod();
  eref_initMod();
  erc_initMod();
  ereftab_initMod();
  ereftab_init(&known);
}
```

Figure 36: empset.c, part 4

```
#if !defined(dbase_h_expanded)
#define dbase_h_expanded

#include "eref.h"
#include "erc.h"
#include "dbase.lh"
#endif
```

Figure 37: dbase.h

```
#include <string.h>
#include "dbase.h"

#define firstERC mMGRS
#define lastERC fmNONMGRS
#define numERCS (lastERC - firstERC + 1)

typedef enum {mMGRS, fmMGRS, mNONMGRS, fmNONMGRS} employeeKinds;

erc db[numERCS];

/* Invariant: The data base is partitioned by
      val(db[mMGRS]), val(db[mNON]), val(db[fMGRS]), and val(db[fNON]) */

bool initDone = FALSE;

void dbase_initMod(void) {
  int i;

  if (initDone) return;
  bool_initMod();
  employee_initMod();
  eref_initMod();
  erc_initMod();
  empset_initMod();
  for (i = firstERC; i <= lastERC; i ++)
     erc_init(&(db[i]));
  initDone = TRUE;
}

eref _dbase_ercKeyGet(erc *c, int key) {
  eref er;

  for (er = erc_iterStart(c); er != erefNIL; er = erc_yield(c))
    if (eref_get(er).ssNum == key) return er;
  return erefNIL;
}
```

Figure 38: dbase.c, part 1

```
eref _dbase_keyGet(int key) {
  int i;
  eref er;

  for (i = firstERC; i <= lastERC; i++) {
     er = (_dbase_ercKeyGet(&(db[i]), key));
     if (er != erefNIL) return er;
   }
  return erefNIL;
}

int _dbase_addEmpls(erc *c, int l, int h, empset *s) {
  eref er;
  employee e;
  int numAdded;

  numAdded = 0;
  for (er = erc_iterStart(c); er != erefNIL; er = erc_yield(c)) {
     e = eref_get(er);
     if ((e.salary >= l) && (e.salary <= h)) {
        empset_insertUnique(s, e);
        numAdded ++;
        }
   }
  return numAdded;
}

dbase_status hire(employee e) {
  if (e.gen == gender_ANY) return genderERR;
  if (e.j == job_ANY) return jobERR;
  if (e.salary < 0) return salERR;
  if (_dbase_keyGet(e.ssNum) != erefNIL) return duplERR;
  uncheckedHire(e);
  return dbase_OK;
}
```

Figure 38: dbase.c, part 2

```
void uncheckedHire(employee e) {
  eref er;

  er = eref_alloc();
  eref_set(er, e);
  if (e.gen == MALE)
    if (e.j == MGR)
       erc_insert(&(db[mMGRS]), er);
       else erc_insert(&(db[mNONMGRS]), er);
    else if (e.j == MGR)
       erc_insert(&(db[fmMGRS]), er);
       else erc_insert(&(db[fmNONMGRS]), er);
}

bool fire(int ssNum) {
  int i;
  eref er;

  for (i = firstERC; i <= lastERC; i++)
     for(er = erc_iterStart(&(db[i]));
         er != erefNIL;
         er = erc_yield(&(db[i])))
           if (eref_get(er).ssNum == ssNum) {
              erc_delete(&(db[i]), er);
              return TRUE;
           }
  return FALSE;
}
```

Figure 38: dbase.c, part 3

```
bool promote(int ssNum) {
  eref er;
  employee e;
  gender g;

  g = MALE;
  er = _dbase_ercKeyGet(&(db[mNONMGRS]), ssNum);
  if (er == erefNIL) {
    er = _dbase_ercKeyGet(&(db[fmNONMGRS]), ssNum);
    if (er == erefNIL) return FALSE;
    g = FEMALE;
    }
  e = eref_get(er);
  e.j = MGR;
  eref_set(er, e);
  if (g == MALE) {
     erc_delete(&(db[mNONMGRS]), er);
     erc_insert(&(db[mMGRS]), er);
     }
  else {
     erc_delete(&(db[fmNONMGRS]), er);
     erc_insert(&(db[fmMGRS]), er);
     }
  return TRUE;
}

bool setSalary(int ssNum, int sal) {
  eref er;
  employee e;

  er = _dbase_keyGet(ssNum);
  if (er == erefNIL) return FALSE;
  e = eref_get(er);
  e.salary = sal;
  eref_set(er, e);
}
```

Figure 38: dbase.c, part 4

```
int query(dbase_q q, empset *s) {
  eref er;
  employee e;
  int numAdded;
  gender g;
  job j;
  int l, h;
  int i;

  g = q.g;
  j = q.j;
  l = q.l;
  h = q.h;
  switch(g) {
    case gender_ANY:
      switch(j) {
        case job_ANY:
          numAdded = 0;
          for (i = firstERC; i <= lastERC; i++)
             numAdded += _dbase_addEmpls(&(db[i]), l, h, s);
          return numAdded;
        case MGR:
          numAdded = _dbase_addEmpls(&(db[mMGRS]), l, h, s);
          numAdded += _dbase_addEmpls(&(db[fmMGRS]), l, h, s);
          return numAdded;
        case NONMGR:
          numAdded = _dbase_addEmpls(&(db[mNONMGRS]), l, h, s);
          numAdded += _dbase_addEmpls(&(db[fmNONMGRS]), l, h, s);
          return numAdded;
        }
```

Figure 38: dbase.c, part 5

```
/* Implementation of query, continued */

    case MALE:
      switch(j) {
        case job_ANY:
          numAdded = _dbase_addEmpls(&(db[mMGRS]), l, h, s);
          numAdded += _dbase_addEmpls(&(db[mNONMGRS]), l, h, s);
          return numAdded;
        case MGR:
          return _dbase_addEmpls(&(db[mMGRS]), l, h, s);
        case NONMGR:
          return _dbase_addEmpls(&(db[mNONMGRS]), l, h, s);
        }
    case FEMALE:
      switch(j) {
        case job_ANY:
          numAdded = _dbase_addEmpls(&(db[fmMGRS]), l, h, s);
          numAdded += _dbase_addEmpls(&(db[fmNONMGRS]), l, h, s);
          return numAdded;
        case MGR:
          return _dbase_addEmpls(&(db[fmMGRS]), l, h, s);
        case NONMGR:
          return _dbase_addEmpls(&(db[fmNONMGRS]), l, h, s);
        }
    }
}

int dbase_sprint(char s[]) {
  int len;
  int i;

  (void) sprintf(&(s[0]), "Employees:\n");
  len = strlen(&(s[0]));
  for (i = firstERC; i <= lastERC; i++) {
     len += erc_sprint(&(s[len]), &(db[i]));
     s[len] = '\n';
     }
  s[len++] = '\0';
  return len;
}
```

Figure 38: dbase.c, part 6

```
table: trait
   includes cardinal
   introduces
      empty: -> Tab
      insert: Tab, Ind, Val -> Tab
      __ \in __: Ind, Tab -> Bool
      eval: Tab, Ind -> Val
      isEmpty: Tab -> Bool
      size: Tab -> Card
   asserts forall i, i1: Ind, v: Val, t: Tab
        eval(insert(t, i, v), i1) ==
           if i = i1 then v else eval(t, i1);
        not(i \in empty);
        i \in insert(t, i1, v) == i = i1 \/ i \in t;
        size(empty) == 0;
        size(insert(t, i, v)) ==
            if i \in t then size(t) else size(t) + 1
```

Figure 39: table.lsl

B LSL for LCL users

LSL specifications define two kinds of symbols, *operators* and *sorts*. The notions of operator and sort are closely related to the C notions of function and type, but it is important not to confuse them. When discussing LSL specifications, we will consistently use the words operator and sort. When talking about C constructs, we will use the words function and type.

An operator is what mathematicians call a "function symbol": it stands for a total mapping from a cross product of values (the *domain* of the operator) to a value (the *range* of the operator). Sorts stand for disjoint sets of values, and are used to indicate the domains and ranges of operators.

The *trait* is the basic unit of specification in LSL. A trait introduces operators and specifies their properties. Sometimes the collection of operators will correspond to an abstract data type. Frequently, however, it is useful to define properties that do not fully characterize a type.

Figure 39 shows a trait that specifies a class of tables that store values in indexed places. It is similar to the specifications in many "algebraic" specification languages.

The specification begins by *including* another specification, cardinal. This specification, which can be found in an LSL handbook, supplies information about the operators +, 0, and 1, which are used in defining the operators introduced in table.

The *introduces* clause declares a set of *operators*, each with its *signature* (the sorts of

```
container: trait
  introduces
    new: -> C
    insert: Elem, C -> C
  asserts C generated by new, insert
```

Figure 40: container.lsl

its domain and range). These signatures are used to sort-check *terms* in much the same way as function calls are type-checked in programming languages.

The body of the specification (following asserts) contains equations between terms containing operators and variables. The first equation resembles a recursive function definition, since the operator eval appears on both the left and right sides. However, it does not fully define eval; it states a relation that must hold among eval, add, and the builtin operator if then else . The fourth and fifth equations together define the operator size relative to the operators 0, 1, and +.

The set of theorems that can be proved about the terms defined in a trait is called its *theory*. It is an infinite set of formulas in first-order logic with equality. (The == symbol used in LSL equations has the same semantics as =. It is used only to introduce another level of precedence into the language.) The theory contains the trait's equations and everything that follows from them, but nothing else. The theory associated with table contains equalities and disequalities that can be proved by substitution of equals for equals. There is no meta-rule implying that terms that are not provably equal are unequal, nor is there one implying that terms that are not provably unequal are equal. For example, neither the formula add(add(t, i, v), i1, v) = add(add(t, i1, v), i, v) nor the formula not(add(add(t, i, v), i1, v) = add(add(t, i1, v), i, v)) is in table's theory. Shortly, we will discuss LSL constructs for non-equational rules; they can be used to generate stronger (larger) theories.

The next series of examples defines a number of properties that can be combined in different ways to define traits that correspond to familiar abstract data types.

The trait container, Figure 40, abstracts the common properties of data structures that contain elements, such as sets, bags, queues, stacks, and strings. container is useful both as a starting point for specifications of many different data structures and as an assumption when defining generic operators over such data structures.

The *generated by* clause in container asserts that each value of sort C can be constructed from new by repeated applications of insert. This assertion is carried along when container is used in other traits, even if they introduce additional operators with range C. Theorems proved by induction over new and insert remain theorems in the theories associated with all such traits.

The trait linearContainer, Figure 41, includes container. It constrains operators inherited

```
linearContainer: trait
  includes container
  introduces
    isEmpty: C -> Bool
    next: C -> Elem
    rest: C -> C
  asserts
    C partitioned by next, rest, isEmpty
    forall c: C, e: Elem
      isEmpty(new);
      not(isEmpty(insert(e, c)));
      next(insert(e, new)) == e;
      rest(insert(e, new)) == new
  implies converts isEmpty
```

Figure 41: linearContainer.lsl

from container (new and insert) as well as the additional operators it introduces. The *partitioned by* clause indicates that next, rest, and isEmpty form a complete set of *observers* for sort C: for any terms $t1$ and $t2$ of sort C, if the equalities next($t1$) == next($t2$), rest($t1$) == rest($t2$), and isEmpty($t1$) == isEmpty($t2$) all hold, then so does the equality $t1$ == $t2$.

The *converts* clause adds nothing to the theory of the trait. Instead, it supplies some checkable redundancy, claiming that this trait fully defines isEmpty by providing equations that allow any variable-free term to be converted to an equivalent term that doesn't contain isEmpty. This can be proved by induction over new and insert, because of the generated by clause inherited from container.

The axioms for next and rest are intentionally weak (defining their meanings only for single-element containers) so that linearContainer can be specialized to define stacks, queues, priority queues, lists, vectors, strings, etc.

In Figure 42, trait priorityQueue specializes linearContainer by adding another operator, \in, and by further constraining next, rest, and insert. The *assumes* clause indicates that whenever priorityQueue is used there must be a total order on Elem. (The reference totalOrder(Elem for T) means the trait totalOrder, which can be found in an LSL handbook, with each occurrence of the sort T replaced by the sort Elem.)

Trait priorityQueue's first implication is a theorem that can be proved using the induction rule inherited from container. It may be helpful in reasoning about priorityQueue and may help readers solidify their understanding of the trait.

The second implication claims that the trait defines next and rest (except when applied to new), isEmpty, and \in. The *exempting* clause indicates that the lack of equations defining next(new) and rest(new) is intentional. The axioms that convert isEmpty are

```
priorityQueue: trait
  assumes totalOrder(Elem for T)
  includes linearContainer(Q for C)
  introduces __ \in __: Elem, Q -> Bool
  asserts forall e, e1: Elem, q: Q
    next(insert(e, q)) ==
       if q = new then e
         else if next(q) < e then next(q) else e;
    rest(insert(e, q)) ==
       if q = new then new
         else if next(q) < e then insert(e, rest(q)) else q;
    not(e \in new);
    e \in insert(e1, q) == e = e1 \/ e \in q
  implies
    forall q: Q, e: Elem e \in q => not(e < next(q))
    converts next, rest, isEmpty, \in
        exempting next(new), rest(new)
```

Figure 42: priorityQueue.lsl

inherited from linearContainer.

Trait priorityQueue is a typical example of the kind of trait used in the specification of LCL interfaces that export abstract data types. In such a trait there is a *distinguished sort* (in this case, Q). The operators can be categorized as *generators*, *extensions*, and *observers*. A set of generators produces all the values of the distinguished sort. The remaining operators whose range is the distinguished sort are extensions. The operators whose domain includes the distinguished sort and whose range is some other sort are observers. For example, in priorityQueue, new and insert form a generator set, rest is an extension, next, isEmpty, and \in are observers, and next, rest, and isEmpty form a partitioning set.

A good heuristic for generating enough equations to adequately define an abstract data type is to write an equation defining the result of applying each observer or extension to each generator. This heuristic suggests writing equations for

```
next(new)
rest(new)
isEmpty(new)
e \in new
next(insert(e, q))
rest(insert(e, q))
isEmpty(insert(e, q))
e \in insert(e, q)
```

```
bag(Elem, Bag): trait
  includes container(Bag for C, { } for new), cardinal
  introduces
    delete: Elem, Bag -> Bag
    __ \in __: Elem, Bag -> Bool
    count: Elem, Bag -> Card
    {__}: Elem -> Bag
    __ \union __, __ - __: Bag, Bag -> Bag
    numElems: Bag -> Card
  asserts
    Bag generated by {}, insert
    Bag partitioned by count
    Bag partitioned by \in, delete
    forall e, e1: Elem, b, b1: Bag
      delete(e, { }) == { };
      delete(e, insert(e1, b)) ==
                     if e = e1 then b else in-
sert(e1, delete(e, b));
      not(e \in { });
      e \in insert(e1, b) == e = e1 \/ e \in b;
      count(e, {}) == 0;
      count(e, insert(e1, b)) ==
         count(e, b) + (if e = e1 then 1 else 0);
      {e} == insert(e, {});
      count(e, b \union b1) == count(e, b) + count(e, b1);
      numElems({}) == 0;
      numElems(insert(e, b)) ==
         numElems(b) + (if e \in b then 0 else 1)
  implies
    forall e, e1, e2: Elem, b: Bag
      insert(e1, insert(e2, b)) == insert(e2, insert(e1, b));
      e \in b == count(e, b) > 0
```

Figure 43: bag.lsl

The trait priorityQueue contains explicit equations for four of the eight, and inherits equations for two more from linearContainer. The remaining two terms, next(new) and rest(new), are explicitly exempted.

Figure 43, bag.lsl, contains another specialization of container. It is quite different from linearContainer, because the order of insertion is irrelevant.

The theories associated with priorityQueue and bag say quite a bit about the properties of these data structures, which have some things in common and some important differences. It is instructive to note some of the things that have *not* yet been specified about these data structures.

- We have not specified how they are to be represented.
- We have not given the algorithms to manipulate them.
- We have not said what functions are to be provided to operate on them.
- We have not specified how errors are to be handled.

The first two decisions are in the province of the implementation. The third and fourth are recorded in interface specifications.

The examples in this appendix (like all simple examples) give a somewhat misleading picture of the process of developing specifications. We almost never define new abstractions starting from first principles. We expect LSL traits to be the principal reusable units in Larch specifications. Reuse reduces the need for invention, helps to avoid previously discovered pitfalls and to benefit from past improvements, and improves communication by standardizing notation.

LSL handbooks are collections of abstractions that experienced specifiers have found to be useful. The creation and refinement of these handbooks represent an intellectual capital investment that will yield dividends in future applications. Handbooks for particular application domains will provide more specialized traits useful in their domains. Handbook traits will frequently supply more operators than are needed for a particular specification; this is not a problem for either the specifier or the implementor, since the operators are not part of the interface and don't have to be implemented.

Index