Massachusetts Institute of Technology
Laboratory for Computer Science
Technical Report 619

Formal Specification Techniques for Promoting Software Modularity, Enhancing Documentation, and Testing Specifications

Yang Meng Tan

June 1994

©Massachusetts Institute of Technology, 1994
All rights reserved

This report is a revised copy of the author's thesis, submitted to the Department of Electrical Engineering and Computer Science in May 1994 in partial fulfillment of the requirements for the degree of Doctor of Philosophy, at the Massachusetts Institute of Technology. The thesis was supervised by Professor John V. Guttag.

The research was supported by grants from DARPA (N00014-89-J-1988), NSF (9115797-CCR), and the Mitsubishi Electric Research Laboratory Inc, Cambridge, MA. The author's current address is: Information Technology Institute, 71 Science Park Drive, National Computer Board Building, Singapore 0511, Republic of Singapore.

Abstract

This thesis presents three ideas. First, it presents a novel use of formal specification to promote a programming style based on specified interfaces and data abstraction in a programming language that lacks such supports. Second, it illustrates the uses of *claims* about specifications. Third, it describes a software reengineering process for making existing software easier to maintain and reuse. The process centers around specifying existing software modules and using the specifications to drive the code improvement process.

The Larch/C Interface Language, or LCL, is a formal specification language for documenting ANSI C software modules. Although C does not support abstract types, LCL is designed to support abstract types. A lint-like program, called LCLint, enforces type discipline in clients of LCL abstract types. LCL is structured in a way that enables LCLint to extract information from an LCL specification for performing some consistency checks between the specification and its implementation.

LCL also provides facilities to state claims, or redundant, problem-specific assertions about a specification. Claims enhance the role of specifications as a software documentation tool. Claims can be used to highlight important or unusual specification properties, promote design coherence of software modules, and aid in program reasoning. In addition, claims about a specification can be used to test the specification by proving that they follow semantically from the specification. A semantics of LCL suitable for reasoning about claims is given.

A software reengineering process developed around LCL and claims is effective for improving existing programs. The impact of the process applied to an existing C program is described. The process improved the modularity and robustness of the program without changing its essential functionality or performance.

A major product of the process is the specifications of the main modules of the reengineered program. A proof checker was used to verify some claims about the specifications; and in the process, several specification mistakes were found. The specifications are also used to illustrate specification writing techniques and heuristics.

Biographical Note

Yang Meng Tan was born in Singapore in 1963. He attended Bukit Ho Swee West School, Gan Eng Seng School, and Hwa Chong Junior College in Singapore. From late 1981 to August 1983, he served in the Singapore Armed Forces. He enrolled at MIT on a Singapore National Computer Board Overseas Scholarship. He completed his bachelor's degree in Computer Science and Engineering in 1987, with a minor in Economics. His bachelor's thesis, *ACE: A Cliché-Based Program Structure Editor*, won the William A. Martin Prize for Best Undergraduate Computer Science Thesis.

In September 1987, Yang Meng continued his graduate studies in computer science at MIT. He obtained his Master's degree in September 1990. His Master's thesis, *Supporting Reuse and Evolution in Software Design*, was done in the Programmer's Apprentice Group, MIT Artificial Intelligence Laboratory. He was a teaching assistant for an undergraduate Artificial Intelligence class in 1990. His graduate minor was in Econometrics.

Yang Meng spent a summer each at Information Technology Institute, National Computer Board, Singapore (1987), and Institute of Systems Science, National University of Singapore (1991). In 1991, he won the Singapore Tan Kah Kee Post-Graduate Scholarship. He can be reached at Information Technology Institute, 71 Science Park Drive, National Computer Board Building, Singapore 0511, Republic of Singapore.

Acknowledgments

John Guttag, my thesis supervisor and mentor, guided my work with steady encouragement, enthusiasm, and understanding. He has given me enough freedom to explore on my own, but sufficient firm guidance so that I can complete my thesis in time. His insightful comments led to many of the good ideas in my work and flushed my bad ideas before they got in the way. I also thank him for helping me with my writing, and for giving me timely and thoughtful feedback despite his many responsibilities.

Dick Waters, my other mentor, personally saw me through my entire graduate studies. He supervised my Master's thesis, guided me in my initial search for doctoral thesis topic, fought hard for securing initial funding for me, and served as my thesis reader.

Steve Garland, my other reader, taught me much about formal proofs and specifications, and responded to my LP requests quickly. I also thank him for his many suggestions in the course of my work.

Members of the Systematic Programming Methodology group made many technical contributions to my work and provided a supportive and stimulating research environment. Mark Vandevoorde taught me the subtleties of data abstraction, helped me with many technical issues, and provided much moral support. Dave Evans helped implement some last-minute LCL features into the LCLint tool. My work is improved by discussions with many people, including Jim Horning, Daniel Jackson, Anna Pogosyants, Mark Reinhold, Raymie Stata, Matthias Weber, and Jeannette Wing. Dorothy Curtis helped me with systems and language questions.

Boon Seong Ang, Shail Gupta, Beng-Hong Lim, and Kah-Kay Sung gave warm friendship throughout my studies. Past and present members of the Singapore and Malaysian student community provided much homely friendship.

My association with MIT was first made possible by an undergraduate scholarship from the National Computer Board of Singapore. I also thank the Board for granting me leave to pursue graduate work and for their patience.

I thank my in-laws for their love and support. I am deeply indebted to my mum, brothers, and sisters-in-law who have made many sacrifices for me. They have been very patient and supportive during my long absence from home.

My wife, Tze-Yun Leong, taught me there's life beyond work. My days in graduate school have been sustained by her love, encouragement, patience, and confidence in me. Despite her own demanding graduate studies, she has always been there for me. I thank her with all my heart.

Contents

1 Introduction 11
- 1.1 The Problems and the Approach . . . 11
  - 1.1.1 Software Modularity . . . 12
  - 1.1.2 Software Documentation . . . 12
  - 1.1.3 Checking Formal Specifications . . . 13
  - 1.1.4 Code Improvement . . . 13
- 1.2 Larch/C Interface Language . . . 14
- 1.3 Related Work . . . 15
  - 1.3.1 Specification Languages . . . 16
  - 1.3.2 Supporting Programming Styles . . . 17
  - 1.3.3 Checking Formal Specifications . . . 17
- 1.4 Lessons Learned . . . 18
  - 1.4.1 Software Reengineering Using LCL . . . 18
  - 1.4.2 Specification Proof Experiences . . . 19
  - 1.4.3 Specification Tool Support . . . 19
- 1.5 Contributions . . . 20
- 1.6 Thesis Organization . . . 21

2 Overview of LCL 23
- 2.1 Larch . . . 23
- 2.2 LCL Basics . . . 24
- 2.3 LCL Function Specification . . . 24
- 2.4 LCL Abstract Type Specification . . . 26
- 2.5 Historical Note . . . 27

3 Supporting Programming Styles 31
- 3.1 Specified Interfaces and Data Abstraction . . . 31
- 3.2 Supporting Specified Interfaces in C . . . 32
  - 3.2.1 LCL Interface Convention . . . 32
- 3.3 Supporting Abstract Types in C . . . 32
  - 3.3.1 Design Goals . . . 33
  - 3.3.2 Implementing Abstract Types in C . . . 33
- 3.4 Tool Support: LCLint . . . 35
  - 3.4.1 Checking Abstract Types . . . 35
  - 3.4.2 Additional Program Checks . . . 35

3.5 Summary 37

4 Specification Techniques and Heuristics 39
4.1 Requirements of the Program 39
4.2 The Design of the PM Program 43
4.3 The date Interface 43
4.4 The date Traits 47
4.5 The trans Traits 51
4.6 The trans Interface 51
4.7 The trans_set Interface and Trait 53
4.8 The position Traits 59
4.9 The position Interface 60
4.10 Summary 68

5 Using Redundancy in Specifications 69
5.1 Specification Testing Approach 69
5.2 Semantics of Claims 70
5.3 Claims Help Test Specifications 73
  5.3.1 Examples of Specification Errors 74
5.4 Claims Help Specification Regression Testing 76
5.5 Claims Highlight Specification Properties 79
5.6 Claims Promote Module Coherence 81
5.7 Claims Support Program Reasoning 82
5.8 Claims Support Test Case Generation 83
5.9 Experiences in Checking LCL Claims 84
  5.9.1 Assessment 84
  5.9.2 Modularity of Claims 85
  5.9.3 Support for Proving Claims 86
5.10 Claims or Axioms? 87
5.11 Summary 88

6 Reengineering Using LCL 89
6.1 Software Reengineering Process Model 89
6.2 A Reengineering Exercise 91
  6.2.1 Study Program 92
  6.2.2 Write Specifications 92
  6.2.3 Improve Code 94
  6.2.4 Write Claims 94
  6.2.5 Check Claims 94
6.3 Effects of Reengineering 95
  6.3.1 Effects on Program Functionality 95
  6.3.2 Effects on Program Structure 95
  6.3.3 Effects on Program Performance 97
  6.3.4 Effects on Program Robustness 97
  6.3.5 Documentation of Program Modules 98

6.4 Specification Tool Support 98
6.5 Summary 99

7 The Semantics of LCL 101
7.1 Basic LCL Concepts 101
7.2 LCL Storage Model 102
  7.2.1 LCL Abstract State 103
  7.2.2 Typed Objects 103
7.3 LCL Type System 103
  7.3.1 LCL Exposed Types 106
  7.3.2 Linking LCL Types to LSL Sorts 107
7.4 LCL Function Specification 108
  7.4.1 Translation Schema 110
  7.4.2 Implicit Pre and Post Conditions 111
  7.4.3 The Modifies Clause 111
  7.4.4 Fresh and Trashed 112
  7.4.5 The Claims Clause 113
7.5 LCL Module 113
  7.5.1 Type Containment 115
  7.5.2 A Simple Type Induction Rule 116
  7.5.3 A Second Type Induction Rule 118
  7.5.4 A Third Type Induction Rule 120
  7.5.5 Jointly Defined Abstract Types 122
  7.5.6 Module Induction Principle 123
  7.5.7 Module Claims 124
7.6 Type Safety of Abstract Types 125
7.7 Summary 125

8 Further Work and Summary 127
8.1 Further Work 127
  8.1.1 More Checks on Specifications 127
  8.1.2 More Checks on Implementations 128
  8.1.3 LCL and Larch Extensions 129
  8.1.4 Reengineering Case Studies 129
8.2 Summary 129

A LCL Reference Grammar 131

B Relating LCL Types and LSL Sorts 135
B.1 Modeling LCL Exposed Types with LSL Sorts 135
B.2 Assigning LSL Sorts to LCL Variables 136
B.3 Assigning LSL Sorts to C Literals 137

C LCL Built-in Operators 139

D Specification Case Study 143
D.1 The char Trait 143
D.2 The cstring Trait 144
D.3 The string Trait 144
D.4 The mystdio Trait 145
D.5 The genlib Trait 146
D.6 The dateBasics Trait 147
D.7 The dateFormat Trait 148
D.8 The date Trait 149
D.9 The security Trait 149
D.10 The lot Trait 150
D.11 The list Trait 150
D.12 The lot_list Trait 151
D.13 The kind Trait 151
D.14 The transBasics Trait 152
D.15 The transFormat Trait 152
D.16 The transParse Trait 153
D.17 The trans Trait 154
D.18 The trans_set Trait 154
D.19 The income Trait 155
D.20 The positionBasics Trait 156
D.21 The positionMatches Trait 157
D.22 The positionExchange Trait 157
D.23 The positionReduce Trait 158
D.24 The positionSell Trait 158
D.25 The positionTbill Trait 159
D.26 The position Trait 160
D.27 The genlib Interface 161
D.28 The date Interface 162
D.29 The security Interface 163
D.30 The lot_list Interface 164
D.31 The trans Interface 165
D.32 The trans_set Interface 167
D.33 The position Interface 168

Chapter 1

Introduction

Software is difficult to develop, maintain, and reuse. One contributing factor is the lack of modular design. A related issue is the lack of good program documentation. The lack of modular design in software makes software changes more difficult to implement. The lack of good program documentation makes programs more difficult to understand and to maintain.

Program modularity is often encouraged through programming language design [27, 34]. In this thesis, we describe a novel approach towards promoting program modularity. We present a formal specification language that is designed to promote software modularity through the use of abstract data types even though the underlying programming language does not have such support. In addition, our specification language is structured in a way that allows useful information to be extracted from a specification and used to perform some consistency checks between the specification and its implementation.

Our specification language supports the precise documentation of programs, and provides facilities to state redundant information about specifications. The redundant information can be used to highlight important properties of specifications so as to enhance the role of specifications as a documentation tool.

While specifications can encourage program modularity, they often contain errors. A specification may not state what is intended. Furthermore, many specification errors occur as a result of evolving program requirements. One approach is to design specifications that can be executed and tested [42]. Here, we study an alternate approach: how redundant information in a specification can be used to test the specification.

We also describe a specification-driven software reengineering process model for improving existing programs. The process is aimed at making existing programs easier to maintain and reuse while keeping their essential functionalities unchanged. We described the results of applying the process to a case study.

1.1 The Problems and the Approach

Programs are difficult to maintain and reuse if they are not modular and not well-documented. Formal specifications can encourage program modularity, and are a good means of documenting programs. Specifications, however, often contain errors that lessen their utility.

Our approach uses formal specifications to promote program modularity and to document program modules, and redundant information in specifications to highlight important properties of specifications and to test specifications.

1.1.1 Software Modularity

Our approach to addressing the problem of software modularity is to design a formal specification language to encourage a more modular style of programming, based on interface specifications and abstractions. In particular, we support a style of programming where data abstraction [27] is a key program structuring principle.

1.1.2 Software Documentation

Our formal specification language can be used for documenting programs. Program documentation is often obsolete with respect to the code it documents. For example, the use of global variables documented in a program comment may be out-of-date with respect to the program. It is useful to have documentation that can be checked against code to detect inconsistencies between the two.

We structure our specification language in a way that makes it easy to build tools for checking the syntax and the static semantics of specifications, and for detecting certain kinds of inconsistencies between a specification and its implementation.

To serve as good program documentation, specifications should be unambiguous and they should highlight important and useful properties of the program design. This helps both the implementor and the client of a program module to understand the module design quickly and easily. Towards this end, our specification language supports constructs for stating *claims*, semantically redundant information about a specification.

Claims are useful for highlighting important or unusual properties of a specification. A specification in our formal specification language defines a logical theory, and claims are conjectures in such a theory. There are infinitely many consequences in a logical theory. Most of them are neither interesting nor useful. It can be difficult for readers of a specification to pick up the important or useful properties of the specification. Specifiers can use claims to highlight these properties. Readers of a specification can use them to check their understanding of the specification.

Claims can also be used to highlight unusual properties in the design of a module. For example, a module that represents and manipulates dates might have an unexpected interpretation of a two-digit representation of a year: it may interpret any number over fifty as the corresponding year in the current century, and any positive number under fifty as the corresponding year in the next century. It is important to highlight such unusual interpretations.

Claims can help support program reasoning. If a claim about a specification has been proved, it states a property that must be true of any valid implementation of the specification, since the specification is an abstraction of all its valid implementations. Claims can sometimes serve as useful lemmas in program verification. In particular, claims about a module can help the implementor of the module exploit special properties of the design.

A well-designed module is not a random collection of procedures. There are often invariants that should be maintained by the procedures in the module. Such invariants can be stated as claims, and proved to hold from the interfaces of the exported procedure. Organizing a module around some useful or interesting claims promotes the design coherence of the module. It helps the designer to focus on overall module properties.

1.1.3 Checking Formal Specifications

Most uses of a formal specification assume that the specification is *appropriate*, in the sense that it states what the specifier has in mind. However, this is often not true, especially when large specifications are involved.

Our general approach towards tackling this problem is: given a formal specification, a specifier can attempt to prove some conjectures that the specifier believes should follow from the specification. Success in the proof attempt provides the specifier with more confidence that the specification is appropriate. Failures can lead to a better understanding of the specification and can identify errors.

A problem related to the problem of checking whether a specification is appropriate is: if a formal specification is modified, how can we avoid inadvertent consequences? Using our methodology, we will attempt to re-prove the conjectures that were true before the change. This regression testing can uncover some of the unwanted consequences.

In our approach, we focus on problem-specific conjectures stated as claims. It is frequently easier to state and prove such conjectures. While this idea is not new [13], a weakness of earlier work is that it gave specifiers little guidance on how to find conjectures that are useful for testing specifications and how to go about proving them. We strengthen this methodology by adding facilities in a specification language so that a specifier can make claims about specifications. A tool can be built to translate such claims, together with the specifications, into inputs suitable for a proof checker. This will enable the specifier to check the claims.

1.1.4 Code Improvement

Many existing programs are written in languages that do not support data abstraction. As a result, they often lack modularity. It is difficult and expensive to maintain and extend such programs to meet changing requirements. It is often cost-effective to improve them in ways that make their maintenance easier.

The process of improving an existing program while keeping its essential functionality unchanged is termed *reengineering*. Using the ideas described in the previous subsections, we give a specification-centered reengineering process model for making programs easier to maintain and reuse. Our reengineering process model is depicted in Figure 1-1. An oval in the figure is a step in the process, and an arrow shows the next step one may take after the completion of a step. We outline the steps of the process below.

1. Study the existing program: We study the program to extract the structure of the program in terms of its constituent modules, and to understand the intended roles and behaviors of these modules.

2. Write specifications for the modules of the program: In this step, we write specifications for the modules of the program. This step is the most significant step of the

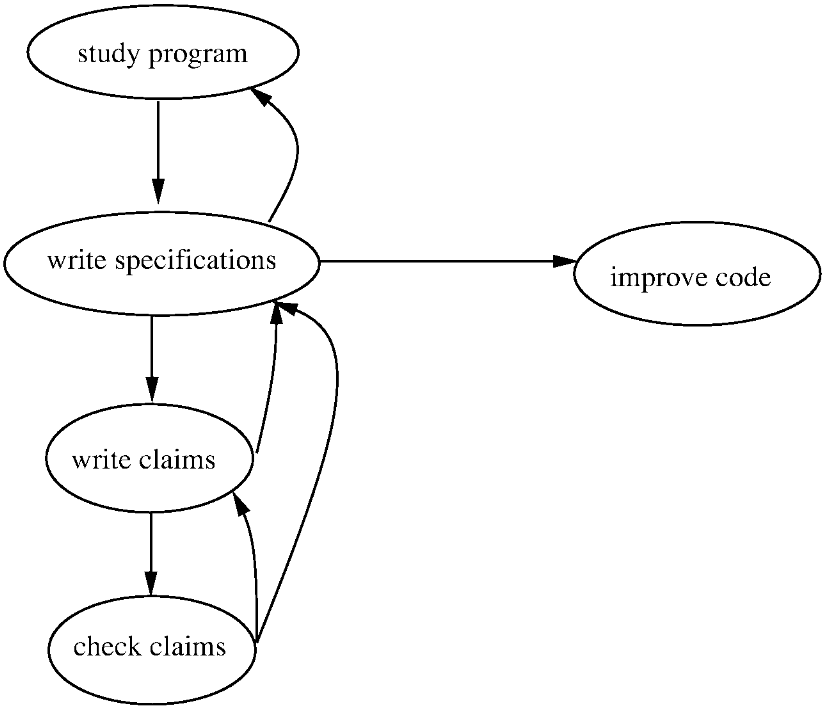

Figure 1-1: Specification-centered software reengineering process model.

reengineering process. The major activities in this step include choosing to make some existing types into data abstractions, identifying new procedural and data abstractions, and uncovering implicit preconditions of procedures.

3. Improve code: This step is driven by the previous step. While the overall requirements of the program do not change, how the requirements are met by the modules of the program can change.

4. Write claims about the specifications of the program modules: In this step, we analyze the specification of each module and its clients to extract properties about the design of the module. We codify some of these properties as claims. This step may lead to changes in the specification of a module that make it more coherent.

5. Check claims: We check that the claims we wrote about a module in the previous step are met by the specification of the module. Depending on the desired level of rigors, this step may range from an informal argument of why a claim should hold, to a formal proof of the claim with the help of a mechanical proof checker.

1.2 Larch/C Interface Language

We designed and implemented a new version of the formal specification language, the Larch/C Interface Language (or LCL), as a vehicle for exploring the main ideas of this thesis. Our language builds on and supersedes a previous design [14].

LCL specifications can serve as precise and formal documentation for the modules that make up the design of an ANSI C program. LCL is an interface language designed in the Larch tradition [38].

A distinguishing feature of a Larch specification language is its two-tiered approach: A Larch specification is composed of two parts: one part is specified in the Larch Shared Language (LSL) and the other in an interface language specific to the intended implementation language. LSL is common to all interface languages [15]. It is used to specify mathematical abstractions that are programming language independent. It supports an algebraic style of specification.

Larch interface languages are programming language dependent. For each programming language of interest, there is a distinct interface language to specify the interfaces between different program modules. The interface specification uses operators that are defined at the LSL level. Relations on program states, exceptions, and other programming language dependent features are specified at the interface level.

Besides providing a language for specifying C interfaces, an important goal of LCL is to encourage a simple, modular, and effective style of programming that combines the strengths of abstract data types and the popularity and flexibility of C. Even though C does not have abstract types, LCL supports the specification of abstract types. A lint-like program, called LCLint, performs many of the usual checks that a lint program [20] does, and in addition, ensures that the LCL-specified type barriers are not breached by clients of an abstract type. This allows the implementation of an abstract type to be changed without having to modify its clients. The resulting improved program modularity is the key practical benefit of using LCL abstract types.

To provide better design documentation and a means of testing specifications, LCL supports constructs for making claims. We also provide a semantics of LCL suitable for reasoning about claims.

There is a facility for stating conjectures about properties that all the functions in an LCL module must maintain. This can be used to make claims about invariants about abstract types, or about the properties that must hold for the private state of the module. We call these *module claims*. Another facility allows conjectures to be associated individual functions of a module. We call these *procedure claims*. A third facility allows conjectures to be associated with the outputs of individual functions; these are called *output claims*.

LCL specifications are structured in such a way that LCLint can efficiently check that certain constraints implied by the specifications are obeyed [5]. For example, LCL requires that the global variables a function accesses be explicitly given in the specification of the function. LCLint uses this information to ensure that only specified global variables are accessed in the implementation of the function. LCL also highlights the objects that a function may modify. This allows LCLint to detect situations where objects that should not be modified are changed. Such checks help uncover programming mistakes.

1.3 Related Work

We classify work related to this research into three categories. First, there are specification languages that are comparable to LCL. Second, there are other approaches to supporting a more modular programming style. Third, there are studies on checking formal specifications.

1.3.1 Specification Languages

LCL is one of several Larch interface languages. Other interface languages include Larch/CLU [38], Larch/Modula-2 [16], Larch/Generic [3], Larch/Ada [12], Larch/ML [40], Larch/C++ [25], Larch/Smalltalk [4], Larch/Modula-3 [15], and Larch/Speckle [37].

Larch/Generic [3] gives a description of a Larch interface language for a programming language that models program executions as state transformations. Each of the other interface languages is designed for the programming language given in its name. All of them share the same underlying Larch Shared Language. Their differences stem mainly from differences in the programming languages that they are designed to be used with. Unlike C, many of these programming languages support data abstraction. New specification constructs have been introduced in LCL to support more concise specifications and to codify some specification conventions.

Like Larch, VDM [22] provides a language for specifying the functional behavior of computer programs. Specifiers use VDM to design specifications for programs and to reason about them. The VDM specification for a program is based on logical assertions about an abstract state, and hence it is not tied to any programming language. It is a uniform language used to define mathematical abstractions and interface specifications. The VDM method is designed to support a rigorous approach to the development of programs by successively proving that a more concrete specification (which in the extreme, would be an operational program) is an acceptable implementation of a more abstract specification.

In contrast to VDM, Larch is two-tiered, and interface specifications are tied to specific programming languages. In Larch, only the Larch Shared Language is programming language independent. Since LCL is designed for the C programming language, it can more easily incorporate specification features that make possible a checking tool such as LCLint.

Z [33] is a specification language based on set theory and predicate logic. Z specifications are composed of pieces of descriptions, called *schemas*. Z is distinguished by having a schema calculus to combine schemas. This makes it easy for specifiers to separate specifications of normal functioning of systems and error conditions, and then combine them later using the Z schema calculus. The schema for an operation must also indicate whether the operation modifies any part of the global state. This feature is similar to the LCL modifies clause. Invariants can also be associated with the global state.

Like VDM, the Z method emphasizes the formal derivation of implementations from formal specifications. Since Z is programming language independent, it is more difficult to perform LCLint-like checks on programs implementing Z specifications.

Anna [29] extends the Ada programming language with annotations that are meant to serve as specifications for Ada programs. An Anna program is an Ada program with machine-manipulable formal comments. Different kinds of Anna annotations can be placed at different levels of an Ada program. For example, there can be constraints on the inputs and outputs of a subprogram unit, or assertions about both the public and private sections of an Ada package. Some Anna annotations can be transformed into assertions that can be compiled into runtime checks. This ability to execute a program against its formal specification is a powerful tool for testing and debugging Ada programs.

The Analyzer [28] is an Anna debugging tool used to locate errors using failures in such runtime checks. It assumes that correct formal specifications defining a program's required behavior are given and placed at various structural levels. Work in Anna is complementary to our work: our work can be used to help discharge an important assumption made in the Anna Analyzer, that the formal specifications used in an Anna program are reasonably free from errors.

1.3.2 Supporting Programming Styles

A traditional approach to promoting modular software designs has been through the design of new programming languages. For example, the following languages support data abstraction through features implemented by a compiler: CLU [26], Modula-3 [31], and C++ [34].

In contrast, our approach extends an existing programming language to better promote modular software designs. Our approach retains the original characteristics of the programming language, and orthogonally adds support for data abstraction. In this way, the programmer is able to use data abstraction techniques where they are desired and is free to exploit the strengths of the programming language where they are needed.

Like LCLint, the FAD system [30] adds abstract types to an existing programming language, Fortran. It extends the syntax of Fortran and uses a preprocessor to convert FAD declarations into standard Fortran. Programs using FAD abstract types cannot be compiled by standard Fortran compilers or easily understood by experienced Fortran programmers unfamiliar with FAD. In contrast, the implementation of an LCL specification is standard ANSI C.

By combining specifications and programming conventions, our approach offers not only a different way of achieving the same goals, but also added advantages. In our approach, specifications provide valuable information that can be used in many ways. Specifications provide concise documentation for the clients of program interfaces, supporting modular program development and maintenance. Specifications contain information that is essential to formal program verification and the kind of design analysis described in Chapter 5 of this thesis. They also contain information that is useful for program optimization [37]. Furthermore, the information can be extracted by LCLint to perform useful quick checks on the implementation of a specification.

1.3.3 Checking Formal Specifications

Our work is inspired by related and complementary work that allows LSL claims to be made and checked [8]. In LSL, the *implies* construct is analogous to the LCL *claims* clause. It allows a specifier to state conjectures about LSL traits. A tool, called *lsl2lp*, translates LSL traits and implies conjectures into LP [7] proof obligations. Since LCL specifications use LSL traits, *lsl2lp* can be used to help test such LSL traits.

The LSL traits used by an LCL interface are auxiliary to the specification; the operators exported by the traits need not be implemented. Unlike LSL implies, which are assertions about auxiliary operators, LCL claims are about properties of interfaces that can be invoked by the clients of the interfaces. LCL claims can refer to values of objects in different computational states, and can specify invariants maintained by interfaces.

The *mural* system [21] is an experimental system that supports the VDM approach to software development. It consists of an interactive generic theorem-proving assistant and a specification tool that supports the writing of VDM specifications. The specification tool can generate the proof obligations for checking the consistency of VDM specifications and the correctness of VDM refinements. The proof obligations can be discharged by the proof assistant with the help of the user. While *mural* can be used to check VDM specifications [6], it is tedious to perform specification proofs in its prover component because its reasoning power is relatively weak.

PAISLey [42] is an executable language for specifying requirements of embedded systems. A specifier can test PAISLey specifications directly, by running them on an interpreter. The shortcoming, however, is that executable specification languages sacrifice ease of use and expressiveness in return for being directly executable. In contrast, Larch specification languages are designed to be simple and concise, rather than executable. In place of executing specifications as a means of testing them, LCL claims allow specifiers to state and check conjectures at design time.

1.4 Lessons Learned

Using LCL, we applied our reengineering model to an existing 1800-line C program, named PM for portfolio manager. In doing so, we learned quite a lot about the ideas presented earlier in this chapter.

1.4.1 Software Reengineering Using LCL

The software reengineering exercise demonstrated how LCL can be used to improve existing C programs.

The most visible product of the reengineering exercise was the formal specifications of the main modules of the program. The specifications serve as precise documentation for the program modules. With well-documented modules, future changes to the program will be easier and parts of the program are more likely to be reused than if the formal documentation were absent. Furthermore, maintenance of the program is eased because our documentation makes explicit a number of implicit design decisions in the program, and it contains claims which highlight some of the central properties of the PM program.

The specification of PM also shows that LCL is adequate for specifying the main modules of a class of real programs. Furthermore, we use the specification to demonstrate how to go about using claims to document and test specifications.

Besides the new specification product, the reengineering process helped to make the program more modular, helped to uncover some new abstractions, and contributed to a more coherent module design. In addition, the process made the program more robust by removing some potential errors in the program. The service provided by the reengineered program also improved because the process helped us identify new useful checks on the user's inputs to the program. We have achieved these effects without changing the essential functionality or performance of the program.

While the benefits of the reengineering process we observed could be obtained with careful analysis and without specifications, we believe that our specification-centered reengineering process provides a methodology by which the benefits can be brought about systematically. Formal specifications have an edge over informal ones because of their precision. The precision sharpens the analysis process, and leaves no room for misinterpretation of the specification. Formal specifications are also more amenable to mechanical tool support, which has improved the program and the specification considerably.

1.4.2 Specification Proof Experiences

Using the specification of the PM program, we experimented with how redundant information in specifications can be used to find errors in specifications and to highlight the design of software modules.

We specified a number of claims about the modules of the PM program. We translated some of the LCL specifications and claims into inputs for the Larch Prover (LP) [7], and used LP to check a few claims. By doing this, we uncovered some mistakes in our original specifications.

We found simple careless errors as well as deeper logical mistakes in our specifications. While some of these errors could have been found by careful inspection, others cannot be so easily detected. When a proof does not proceed as we expect, it is often because we have glossed over some fine points in our specifications, or we have omitted some essential information in the specification. Besides improving the quality of the specifications, we note that the chief benefits the proof exercises provide are a better understanding of our design and increased confidence in the specifications.

Verifying claims is a time-consuming and difficult process. We found that the places where we were stuck longest in the proof were also where we learned most about our specifications. However, some of the expended efforts could be reduced with better theorem proving techniques.

1.4.3 Specification Tool Support

We found tool support to be indispensable in writing formal specifications. We used three main kinds of tools. First, there are tools that check the syntax and static semantics of specifications. The LSL checker checks LSL traits, and the LCL checker checks LCL specifications. These help uncover many careless errors such as spelling mistakes and syntax errors.

Second, we used LP to verify LCL claims. LP was instrumental in catching the mistakes we found in the specification. The proof checker lessens the proof effort by helping us to be meticulous in our proof steps, and supports regression testing of specifications.

Third, we used LCLint to check both the implementations and clients of an LCL specification. The tool helped us find errors in our code. Two classes of errors stood out. One, LCLint was useful in locating code that violated an abstract type barrier. Two, LCLint found places in a function where global variables were accessed even though such access was not sanctioned by the specification of the function.

We note another benefit of LCLint: Since LCLint checks aspects of consistency between an LCL specification and its implementation, when an error is detected, it is sometimes a specification error rather than a coding error. For example, when LCLint reports that a global variable is accessed in an implementation when its specification does not allow such access, it is sometimes a specification omission. LCLint dutifully reports the inconsistency that leads us to correct our specifications, and thus improves the documentation of the program.

This experience argues for the use of formal description techniques rather than informal ones, because we can build better tools to support formal description techniques.

1.5 Contributions

This thesis makes the following contributions:

First, we designed and implemented a new version of the formal specification language, LCL. LCL can be used to specify ANSI C program modules. We implemented an LCL checker program that checks the syntax and the static semantics of LCL specifications. We provided a detailed description of the semantics of LCL, including a data type induction principle for LCL abstract types. Our language design also provides the framework for the construction of the LCLint program [5].

Our LCL language builds on and supersedes a previous design, LCL version 1.0 [14]. The principal innovations in our version include a richer abstract type model that adds immutable types, a new parameter passing convention that provides more freedom to the implementors of abstract types, an extension of the modifies clause to handle collections of objects, and new language constructs to enable more compact specifications and to state claims. [1] A more detailed description is given in Section 2.5.

Second, by the design of LCL, we illustrated an approach for supporting programming styles using a formal specification language, programming conventions, and checking tools. In particular, we designed a formal specification language that supports a style of programming based on interfaces and abstract types. Our approach combines the strengths of using interfaces and abstractions and the flexibility of the underlying programming language. We illustrated the application of this approach on a substantive example.

Third, we demonstrated how redundant information in a formal specification can be used to improve the quality of the specification. We showed how claims can highlight important specification properties, promote module coherence, support program reasoning, help test specifications, and support regression testing of specifications. To the extent that a formal specification is the codification of a software design, our approach allows software designs to be analyzed and studied before code is written for them. We also provided some practical experiences in using a proof checker to verify specification properties.

Fourth, we gave a software reengineering process model for improving existing programs in ways that make their maintenance and reuse easier. Our process model centers around specifying existing program modules and using the specifications to drive the code improvement process. We applied our process to an existing, working, 1800-line C program and described the effects of the process.

---

[1] [15] adopted many of the key design changes made in our current version.

1.6 Thesis Organization

Chapter 2 gives an overview of LCL that is detailed enough for understanding the main points of the thesis. It covers the Larch framework underlying the design of LCL, and some features of LCL as a specification language.

Chapter 3 describes how LCL specifications can be used to support a style of C programming based on specified interfaces and data abstraction.

Chapter 4 describes the specification of the reengineered PM program. We can also view the reengineering exercise as a specification case study. We use the specification to illustrate the techniques and heuristics we employ in writing specifications. We illustrate ways to achieve more compact and easier to understand specifications. This includes LCL constructs that highlight checks that must be performed by the implementor, and those that codify specification conventions. We also point out some common errors in writing specifications. Many of the techniques we document are general; they are specific neither to LCL nor Larch.

Chapter 5 describes the claims concept and describes the various uses of claims in a formal specification. We illustrate how claims can be used to highlight important properties of specifications, test specifications, support program reasoning, and promote the design coherence of software modules.

Chapter 6 combines the ideas in Chapter 3 and Chapter 5 to describe a specification-centered software reengineering process model for improving existing programs in ways that make them easier to maintain and reuse. The impact of applying the process to the original PM program is described. It also gives our experiences in using various tools for writing and checking formal specifications.

Chapter 7 provides a more complete description of LCL. It describes the interesting aspects of LCL's semantics. In particular, a data type induction principle is given for LCL abstract types. This chapter is useful as a reference for the subtler points of LCL and for other specification language designers.

Chapter 8 contains a discussion of further work and summarizes the achievements of the thesis.

The reference grammar of LCL is given in Appendix A. A number of static semantic issues are addressed in Appendices B and C. The LCL specifications of the main modules of PM are given in Appendix D.

Chapter 2

Overview of LCL

The ideas we study in this thesis are exercised in the context of the Larch/C Interface Language, LCL. LCL is a formal specification language designed to document C interfaces, and to support a programming style based on specifications and abstract data types, even though C does not support abstract types.

In this chapter, we describe LCL as a formal specification language in sufficient detail so that the main ideas in the thesis can be understood. A tutorial-style description of LCL can be found in [15]. The use of LCL to support programming styles is described in the next chapter. The semantics of the LCL language is described in Chapter 7.

2.1 Larch

LCL is a formal specification language designed in the Larch tradition [38, 15]. Larch is a family of specification languages designed for practical application of formal specifications to programming. It embodies a software engineering approach in which problem decomposition, abstraction, and specification are central [27].

Larch specification languages are used to formally specify the abstraction units that make up the design of a computer program. Different programmers can tackle the implementation of different abstraction units independently and concurrently. The formal specifications serve as contracts between the implementors of different units.

Even before a specification is sent to implementors to be constructed, the specifier can analyze the specification to minimize errors in the specification. This can save costly mistakes early in the production process. Larch specifications are designed to facilitate the construction of tools that help specifiers check the syntax of specifications and analyze the semantics of the specifications.

Larch specifications are distinguished by their two-tiered structure. A Larch specification is composed of two parts: one part is written in the Larch Shared Language (LSL) and the other in a Larch interface language specific to the intended implementation language.

- Larch Shared Language: The Larch Shared Language is common to all interface languages [15]. It is used to capture mathematical abstractions that are programming language independent.

- Larch Interface Languages: An interface specification must describe how data and control are transferred to and from the caller of a procedure. Programming languages have different parameter passing mechanisms and exception handling capabilities. It is desirable to design an interface language that is specific to a programming language so that specifications written in the interface language can be more precise than those written in some universal interface language. It is also easier for a programmer to implement a Larch interface specification.

  Interface specifications use operators that are defined at the LSL level. This connection between a Larch interface specification and an LSL specification is made by a link in the interface specification. Relations on program states, exceptions, and other programming language dependent features are specified at the interface level.

  The specification of a procedure is modeled as a predicate on a sequence of states. In the special case of sequential programs, this is a relation between two states, the state before and after the execution of the procedure.

2.2 LCL Basics

Since LCL specifications describe the effects of C-callable interfaces, the semantic model of LCL supports that of C. Each scope in a C program has an *environment* that maps program variables to typed locations. A C function can read and modify the contents of a memory *store*, which maps locations to values. Since C uses call by value, the callee cannot affect the environment of the caller. Therefore, the state of a C computation can be modeled as a store. In addition to supporting the basic computational view provided by C, the semantic model of LCL also supports abstractions of locations, called *objects*. Like memory locations, objects are containers of values. Locations can be viewed as a special kind of object whose operators are predefined by the C programming language. The binding of objects to their values is a *state*.

LCL is statically typed: the type of a value that can be assigned to an LCL object in a state is fixed. A type is viewed as a collection of values with a set of operations that can act on those values. There are two categories of types in LCL. LCL *exposed types* are the built-in types of C, and *abstract types* are data abstractions that can be specified in LCL and implemented in C. Since exposed types are not used extensively in this thesis, their description is not given here. A tutorial-style description is given in [15], and their finer semantic details are given in Chapter 7 and Appendices B and C.

LCL supports two kinds of abstract types: mutable and immutable types. Instances of an immutable type cannot be modified; they are analogous to mathematical values and C `int`s or `char`s. Instances of a mutable type can be modified by C function calls.

2.3 LCL Function Specification

The basic specification unit in LCL is a C function specification. A key feature of LCL function specifications is that each of them can be understood independently of other specifications. The state before a function is invoked is called the *pre state*, and the state after the function returns is called the *post state*.

```
int count;
spec int hidden;
int add (int i, int j) int count, hidden; {
  requires hidden^ < 100;
  modifies count, hidden;
  ensures result = i + j ∧ count′ = count^ + 1 ∧ hidden′ = hidden^ + 1;
}
```

Figure 2-1: Simple examples of LCL specifications.

Figure 2-1 shows the LCL specifications of a C global variable named `count`, an LCL *spec* variable named `hidden`, and a simple C function named `add`. C global variables can be read and changed from any program context, and they are exported to modules that import the module containing the above specifications. Spec variables are like global variables, except that they are private to the module that defines them. They are specification constructs, and are not exported. As such, they need not be implemented.

The specification of `add` indicates that it takes two integer formals, accesses a global variable named `count`, a spec variable named `hidden`, and returns an integer. The `requires` clause indicates that a precondition is needed; the value of the `hidden` variable must be less than 100 in the pre state. The `modifies` clause specifies which of the input objects may potentially be changed. In this case, it says that `count` and `hidden` may be changed. The `ensures` clause describes the effects this function is supposed to achieve. The reserved word `result` is used to refer to the returned value of the function. The symbol ^ is used to extract the value of an object in the pre state, and the symbol ′ is used to extract its value in the post state. The specification says that `add` returns the sum of its formals, and increments both `count` and `hidden` by one. An LCL function specification has an implicit ensures clause that the function terminates if the requires clause is satisfied. The meaning of an LCL function specification is: if the preconditions specified in the requires clause hold, then the relation specified by the modifies clause and the ensures clause must hold between the pre and the post states.

Since C function calls pass parameters by value, the formal parameters of a C function denote values. An exception to this rule is C arrays: they can be viewed as pass by reference. As such, LCL models C arrays as objects so that any change to an array is visible outside the function. Since changes to global and spec variables contain values that persist across function invocations, they always denote objects. Hence, the formal parameters `i` and `j` in Figure 2-1 are used without state decorations whereas `count` and `hidden` need state decorations to extract their values from the pre or post state.

The *input arguments* of a function consist of the formal parameters and the global variables the function accesses. The set of objects that appear explicitly or implicitly in the modifies clause of a function specification is called its *modified set*. The *output results* of the function consist of `result` and the modified set of the function.

While LCL can be used to specify programs in which only C built-in types are used, it is not best suited for specifying such programs. LCL is designed for specifying the behaviors of a class of C programs in which abstract types play a major role.

2.4 LCL Abstract Type Specification

An interface can contain global and private variable declarations, type specifications, and function specifications. An interface serves three functions. It is used to group C variables, types, and functions together so they can be imported or exported as a single unit. Second, an interface supports data encapsulation: only the functions exported by the interface can access private data that are declared within it. Third, an interface can define an abstract data type.

```
mutable type intset;
uses set (int, intset);
intset create (void) {
  ensures result′ = {} ∧ fresh(result);
}
int choose (intset s) {
  requires s^ ≠ {};
  ensures result ∈ s^;
}
bool add (int i, intset s) {
  modifies s;
  ensures (result = i ∈ s^) ∧ s′ = insert(i, s^);
}
bool remove (int i, intset s) {
  modifies s;
  ensures (result = i ∈ s^) ∧ s′ = delete(i, s^);
}
```

Figure 2-2: The LCL specification of an abstract type.

The specification of an interface defining an abstract data type is shown in Figure 2-2. The first line in Figure 2-2 declares a new mutable abstract type named `intset`. Clients of `intset` do not have direct access to the implementation of this type; they manipulate instances of the `intset` type by calling functions exported in the `intset` interface. The second line links operators used in the specification to an LSL specification. In this case, the LSL specification is the `set` trait shown in Figure 2-3. The trait parameters `E` and `C` are instantiated as `int` and `intset` respectively in the `intset` interface.

The `set` trait in Figure 2-3 introduces a number of sorts, operators, and axioms that constrain the meaning of the operators. LSL sorts are used to model LCL types. The lines that follow the `introduces` construct give the signatures of the operator symbols. The next section adds different kinds of axioms. There are two special kinds of axioms: the `generated by` clause asserts that all values of the `C` sort can be generated by the operators {} and `insert`. This provides an induction schema for the `C` sort. The `partitioned by` clause asserts that all distinct values of the `C` sort can be distinguished by ∈. Terms of the `C` sort that cannot be distinguished by ∈ are equal. The rest of the axioms are in the form of universally quantified equations. The precise semantics of LSL traits is given in [15]. It suffices to know that a trait provides a multi-sorted first-order theory with equality for the operators and sorts that the trait introduces, plus any given induction schemas for the sorts.

```
set (E, C): trait
  introduces
    {}: → C
    insert, delete: E, C → C
    __ ∈ __: E, C → Bool
  asserts
    C generated by {}, insert
    C partitioned by ∈
    ∀ s: C, e, e1, e2: E
      ¬ (e ∈ {});
      e1 ∈ insert(e2, s) == e1 = e2 ∨ e1 ∈ s;
      e1 ∈ delete(e2, s) == e1 ≠ e2 ∧ e1 ∈ s;
```

Figure 2-3: The set trait.

The rest of the `intset` interface in Figure 2-2 contains the specifications of the functions it exports. These functions create, modify, and observe `intset`'s. The built-in operator `fresh` is used to indicate objects newly created by a function, i.e., objects that are not aliased to any existing object. The specification of `create` says that `create` takes no arguments and returns a fresh `intset` object whose value in the post state is the empty set. A function that returns some instances of a type is termed a *creator* of the type. The `create` function is a creator of the `intset` type.

An omitted modifies clause, like that in `choose`, means that the abstract value of no reachable object can be modified. However, the representation of these reachable objects may be changed; only their abstract values must remain the same. This allows for benevolent side-effects. For example, `choose` may re-arrange the order of the elements in the representation of the input set without affecting its abstract set value. The `choose` function in the interface also illustrates non-determinism in the specification: the returned integer can be any element in the given `intset`. The specification does not constrain which one. A function that does not produce or modify instances of a type is termed an *observer* of the type. The `choose` function is an observer of the `intset` type.

A *mutator* of a type is a function that may modify some instance of the type. The `add` function inserts an integer and returns true if the integer was already in the set. The input set is modified if the integer was not already in it. Similarly, `remove` deletes an integer from the set and returns true if the integer was already in the set. They are both mutators of the `intset` type.

2.5 Historical Note

The design of the LCL language described in this thesis, LCL Version 2.4, builds on and supersedes a previous design, LCL Version 1.0, described in [14]. The chapter on LCL in [15] adopted a previous version of our current design. There are many differences between the Version 1.0 and Version 2.4; the following are the key ones:

- A Description of LCL Semantics: The previous design explains the features of LCL through examples, but no formal or informal semantics are given. Our current design provides a more rigorous and detailed description of LCL semantics. In particular, we provide induction rules for deriving type invariants from the specifications of abstract types.

- A Richer Abstract Type Model: The previous design supports only mutable abstract types. Our current design adds another kind of abstract type, the immutable types. Immutable abstract types are useful because they are simpler, and they suffice for abstractions where modifications are not needed.

- A Better Parameter Passing Convention: The previous design requires abstract values be passed to and returned from functions indirectly by pointers. This requirement is dropped in our design, making abstract types more like C native types. It also makes LCL specifications easier to read and understand. Furthermore, it allows more implementation freedom for immutable types.

- Checks clause: A new kind of clause, the **checks** clause, is added to LCL. The checks clause is a compact way of specifying checks that the implementor of a function specification must carry out.[1] It helps to highlight the difference between programmer errors and user errors, and promote defensive programming.

- Exposed Types with Constraints: Specifiers can associate a constraint with an exposed type via a `typedef` declaration. This feature allows specifications to be more compact, and hence easier to read.

- Type Checking of Exposed Types: The type checking of exposed types was changed from type equivalence by name to type equivalence by structure. This makes LCL more compatible with C type checking.

- Modifies Clause Extensions: The previous design does not have a way of conveniently permitting modification of a collection of objects. Our current design allows the modifies clause of a function specification to accept a type name (denoting a mutable type). This indicates that all instances of the named type may be modified by the function. This is useful in specifications involving a type whose instances contain other mutable objects. For example, suppose we have a type that is a stack of mutable `intset`'s, and a function that may modify any set in its input stack. We can then specify this in the modifies clause as **`modifies intset`**.[2] The information in the modifies clause can be easily extracted by LCLint so that in principle, LCLint can perform better checks on the implementation of the function specification.

- Claims: A new syntactic category called the *claims* clause is added to the syntax of an LCL function specification. An LCL claim is intended to be a logical conjecture about an LCL specification. A new construct is also added to support claims that pertain to an entire interface. The form and uses of claims are discussed in Chapter 5.

---

[1] Our checks clause is inspired by a similar construct in Larch/Modula-3 for specifying Modula-3's checked run-time errors: an implementation must ensure that a failed checks clause must result in a checked runtime error.

[2] The ensures clause can be used to more concisely restrict the scope of modifications to the sets that are contained in the input stack.

Chapter 3

Supporting Programming Styles

Software, if written in a good programming style, is easier to maintain and reuse. The traditional way of encouraging a desired programming style is to design a new programming language with features that codify that style. In this chapter, we describe a different approach: we show how a specification language, together with some programming conventions and a checking tool, can support a programming style.

The C programming language is a portable and flexible programming language. Two important shortcomings in C are a weak notion of interface and no support for abstract data types. A design goal of LCL is to address these weaknesses through a stylized use of C with the help of a checking tool, called LCLint [5]. Another goal of the design is to support a desired programming style without changing the programming language so as to retain the versatility of C.

Chapter 2 described LCL as a formal specification language. In this chapter, we show how LCL specifications can be used to support a style of C programming based on specified interfaces and data abstraction.

3.1 Specified Interfaces and Data Abstraction

A software module is a collection of procedures and data. In general, an interface of a software module is a description of the module that provides a defined means of interaction between the module and its clients.[1] An interface describes the types of the data and the procedures that are exported by the module. An interface isolates some implementation details of a module from its clients. Clients use the module by calling the procedures exported by a module without relying on its implementation details. The type information provided by an interface can enable some static type checking of procedure calls in clients in the absence of the implementation module.

A specified interface is an interface with a precise description of the behavior of the interface. It contains sufficient information so that clients can rely on a valid implementation of the interface without looking at the actual implementation.

---

[1] This notion of interface is compatible with and more general than our notion of LCL interface introduced in the previous chapter.

A special kind of specified interface is an abstract data type [27]. The interface provided by an abstract type is narrow: clients can only manipulate the instances of the abstract type by calling the procedures that are exported by the type. They do not have access to the representation of the type. This barrier promotes program modularity by allowing the implementor of an abstract type to modify the representation type without affecting programs that use the abstract type.

3.2 Supporting Specified Interfaces in C

A C interface is a function prototype. It consists of the returned type of the function and the types of the input parameters.

The prototypes of C functions are kept in C header files. They are included by clients to enable type checking by the C compiler. Since a type must be defined before it is used, the types used in an implementation must be provided in the header file too. This means that client programmers have access to implementation information. This reduces the independence between the clients and the implementation of a module.

3.2.1 LCL Interface Convention

LCL separates interface information from implementation details by putting interface information in LCL specifications and disallowing clients access to the header files. An LCL specification contains all the information its clients will need and can rely upon. The implementor is free to change the implementation as long as the specification is satisfied.

The conventional way of using an LCL interface is illustrated in Figure 3-1 using the `intset` example introduced in the last chapter. The LCL specification of the `intset` module is contained in the file `intset.lcl`, the code implementing the functions exported by the `intset` module in `intset.c`, and the header file of the `intset` module in `intset.h`. As usual, `intset.c` includes the header file `intset.h`, and so do client code such as `client.c`. The headers of specified functions and any specified exposed type declarations in `intset.lcl` are extracted by the LCLint tool to produce an `intset.lh` file. This file should be included in `intset.h` so that the compilation of both `intset.c` and `client.c` have the appropriate type information. Specified functions can be implemented as macros; their macro implementations should be placed in the header file. Clients of the `intset` module should only consult the LCL specifications of `intset`; they should not rely on information in `intset.h`. This achieves the goal of effecting a clear separation between the clients and the implementation of an LCL interface.

3.3 Supporting Abstract Types in C

LCL interface conventions provide a physical separation between the clients and the implementation of a C module. Without abstract types, however, it only hides an extra piece of information that was unavailable in the C header file convention: whether a function is implemented as a macro or not. This is because to specify the functions in a module, the specifier inevitably needs to declare the types involved. This means that the client still has much of the information about the exposed types used. The introduction of abstract types,

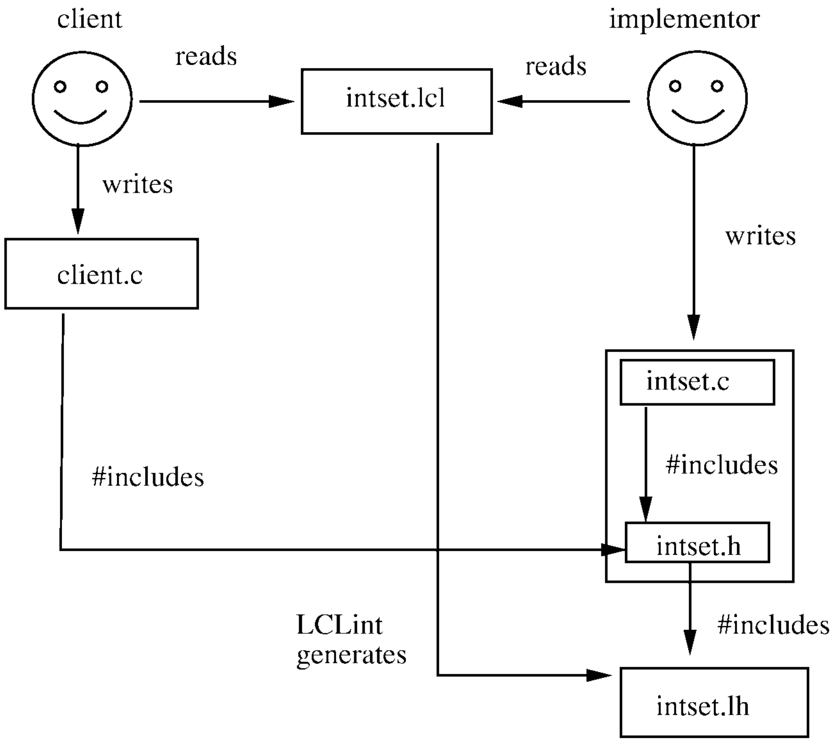
client
reads
intset.lcl
reads
implementor
writes
client.c
writes
intset.c
#includes
#includes
intset.h
LCLint
generates
#includes
intset.lh

Figure 3-1: LCL interface conventions.

however, requires the strict isolation of information about the types used to implement abstract types from clients.

3.3.1 Design Goals

A few goals guide our design of LCL abstract types. First, from the clients' perspective, the semantics of an abstract type must be independent of the type chosen to implement the abstract type, called the *rep type*.

Second, C programs using abstract types should be syntactically similar to those using exposed types. This makes learning abstract types in C easier, and provides a more elegant introduction of abstract types in C. In particular, variables of abstract types should be declarable and assignable in client programs, and instances of abstract types can be passed to and returned from function calls.

3.3.2 Implementing Abstract Types in C

LCL design goals and the language constraints of C combine to motivate two guidelines that the implementor of an LCL abstract type must follow in order to give a uniform semantics to abstract types.

First, since a variable of an abstract type must be assignable, the rep type of an abstract type must be assignable in C. This excludes the use of C arrays as rep types. Pointers can, of course, be used instead.

There are two kinds of abstract types in LCL. Since instances of an immutable type cannot be modified, their sharing properties are immaterial to the semantics of the type. The implementor is free to choose any assignable C builtin type or other abstract types to implement an immutable type.

The semantics of mutable types, however, requires that assignments cause sharing. For example, in Figure 3-2, after the assignment of `s2` to `s`, the two names denote the same object so that any change to one is observable in the other.

```
{ intset s, s2;
  s = create();
  s2 = s;
  add(1, s); /* s2 sees the change in s */
}
```

Figure 3-2: Assignments of mutable types cause sharing.

However, C assignments have copying semantics. This motivates the second implementation requirement of LCL abstract types: the rep type of a mutable type must be chosen so that C assignments of its instances cause sharing. This can be achieved in at least three ways:

One, a mutable type can be implemented using C pointers. Two, a mutable type can be implemented using handles. A handle to an object is an index into some privately maintained storage that stores the object. The storage must be local to the module implementing the abstract type so that the only way it can be modified is through the exported functions.[2] These exported functions can hence interpret a handle in a consistent and shared manner. Three, a mutable type can be implemented by some other mutable type.

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

typedef struct _list {int data; struct _list *next;} list;
typedef struct {int size; list *contents;} setRep;
typedef setRep *intset;

#include "intset.lh"

#define choose(s) ((s)->contents->data)

#endif
```

Figure 3-3: A C type implementing the intset abstract type.

Figure 3-3 shows a particular rep type of the `intset` type given in the previous chapter.

---

[2] It can be declared `static` in C.

The rep type is a pointer to a structure that contains the set cardinality, and a linked list of the members of the set. The `choose` operation is implemented as a macro in the header file. The implementation of the `intset` interface using this rep type is ordinary, and hence it is not given.

3.4 Tool Support: LCLint

It is possible to use LCL conventions effectively without tool support. Tool support, however, is desirable to allow errors to be detected earlier and quickly. LCLint is a lint-like tool that performs additional checks on C programs without affecting the compilation of the programs. Like lint, it is designed to find errors in programs quickly [5].

The use of LCLint to generate function prototypes from LCL specifications for compilation has already been mentioned. One key function of LCLint is to detect abstract type barrier violations in clients.

3.4.1 Checking Abstract Types

LCLint ensures that the only way client programs use an abstract type is through interfaces defined by the type. This is achieved by the following checks.

First, LCLint treats an abstract type as a new type, and does type checking by name.

Second, LCLint disallows type casting to and from abstract types.

Third, instances of abstract types cannot be used with any C built-in operator except the assignment operator (`=`) and the `sizeof` operator. In particular, the comparison operator (`==`) cannot be used. It does not have a consistent meaning on immutable types: its meaning could depend on the choice of the rep type of the abstract type. Furthermore, it can cause potential confusion when it is exported for mutable types. Should it mean object identity or value equality? To preserve the uniformity of the semantics of abstract types, the comparison operator is not exported automatically. Like other user defined functions, it can be exported if the user specifies it. Its implementation can be made efficient through the use of macros.

Besides checking the clients of an abstract type, LCLint also performs some checks on the implementation of the type. As explained in the previous section, a valid representation of an abstract type must be assignable in C. LCLint ensures that the chosen rep type is not an array. It is not possible to check if the rep type chosen for a mutable type is such that assignments cause sharing.

3.4.2 Additional Program Checks

Up to this point, the use of programming conventions and specifications for supporting abstract types offers few added advantages to the language design approach. There is, however, one important difference: an LCL specification contains information that can be used to check its implementation. LCLint supports the following additional checks that can be turned on or off by the programmer:

Macro Checking: Clients of a specified function should not rely on whether the function is implemented by a macro or by a function. This entails additional checks on macros that implement specified functions. LCLint treats macros implementing a specified function as if they are C functions. Additional safety checks are performed on such macros. For example, each parameter to a macro implementing a specified function must be used exactly once in the body of the macro. This ensures that side-effects on its arguments behave as expected. While the syntax of a C macro definition does not allow the types of macro parameters to be included, LCLint can obtain the necessary information from the specifications of the function being implemented by the macro.

Global Variable Checking: LCL requires that each global variable a function accesses be listed in the specification of the function. This information enables two checks on the implementation of the function: LCLint checks that every global variable used in the implementation is in the global list of the function, and that every global variable listed is potentially accessed by the implementation.

Complete Definition Checking: LCLint ensures that every function exported from a module is specified, and every specified function is implemented in the module.

Modification Checking: The modifies clause in the specification of a C function highlights the side-effects of the function. This information can easily be extracted from LCL specifications. It can be used to detect potential errors in code. For example, consider the specifications given in Figure 3-4. The specification of `P` states that its input set must not be modified, and the contrary is true in the specification of `Q`. Figure 3-5 illustrates a potential error in the implementation of `P`: the implementation of `P` passes its input set to a call to `Q`, which may modify the set.

```
void P (intset s) {
  modifies nothing;
  ensures ...;
}
void Q (intset s) {
  modifies s;
  ensures ...;
}
```

Figure 3-4: Modifies checking illustration Part I: specifications.

```
void P (intset s) {
 Q(s); ...
}
```

Figure 3-5: Modifies checking illustration Part II: unsafe code.

There are a number of difficulties in checking object modifications. Since LCL specifications only constrain the abstract values of objects, it is possible for a program to modify the concrete representation of an object without changing the abstract value of the object. Thus, the only reliable way to tell if the abstract value of an object has changed requires proving that the concrete value has changed in a way that makes the abstract value different. In addition, checking object mutation using the modifies clause requires aliasing analysis. The problem is undecidable in general. Despite these difficulties, we believe that detecting potential errors in object modification is useful in practice.

3.5 Summary

We have described how LCL supports a C programming style based on specified interfaces and abstract types. The traditional approach towards supporting programming styles is through the design of a new programming language. We take a different approach: we use a specification language, together with programming conventions and a checking tool, to support the desired programming style. This approach retains the characteristics of the programming language, and orthogonally adds the strengths of the programming style introduced.

In some ways, LCL can be viewed as an attempt to address inadequacies of C. To the extent that it does this, it can be used as a model for other situations where specific programming styles are desired but are inadequately supported by the chosen programming language or platform.

Our approach is distinguished by the use of specifications, which contain information useful for performing certain consistency checks between the specifications and their clients, and between the specifications and their implementations. Such checks help uncover programming mistakes.

Chapter 4

Specification Techniques and Heuristics

In this chapter, the LCL specifications of the key modules of an existing, working, 1800-line C program are described. The program has been in use for several years. The specifications describe the reengineered version of the program. The new version of the program is the result of a reengineering process to be described in Chapter 6, where the two versions of the program are also compared.

The specification of the program is used to discuss some specification issues and illustrate specification techniques. Hence, we also refer to the reengineering exercise as a specification case study. In addition, the presentation of the specification provides the background necessary for better understanding the next chapter, which uses the specification to illustrate the various uses of redundant information in specifications.

While some issues raised in this chapter are specific to Larch specifications, most issues are not. Users of other specification languages are likely to find correspondences in their favorite languages.

The organization of the chapter is as follows. A brief description of the functionality of the specified program, PM, is given first. Next, the design of the program is sketched, and the specification of the key modules of PM are described in a bottom-up fashion. If an abstraction we specify is a familiar one, we describe its LCL interfaces before describing its supporting traits. Otherwise, we introduce the abstraction by describing some traits before describing the interfaces that use the traits.

The details of some interfaces are omitted in the chapter. The description is intended to highlight the techniques used to specify the interfaces. The complete specifications in the case study are given in Appendix D. The specifications have been checked by the LSL checker and the LCLint tool for syntax and type correctness. The implementation of the specification has also been checked by the LCLint tool.

4.1 Requirements of the Program

We name the program we have specified PM, for portfolio manager. It keeps track of the portfolio of an investor. It processes a sequence of financial security transactions, checks

```
Yang Meng Tan, Acct 1
CommonX B 10 1.00 10 1/1/92 1 from John
CommonX B 10 2.00 20.0 1/1/92 2
CommonX B 20 3.00 60 3/1/92 3
CommonX S 25 4.00 100 4/1/93 3,1
CommonY B 10 1.00 10 LT 1
CommonY C 10  2.00 12/1/92 1
MuniZ B 1000 92.113 92113 11/1/82 1 9.4%
MuniZ IM 1000  4700 1/04/93
MuniZ S 1000 102 102000 1/01/93 1
TBill92 B 10 90 900 1/2/92 1
TBill92 M 10  100 1/2/93 1
```

Figure 4-1: An example input file to the PM program.

their consistency, and summarizes the portfolio. It handles stocks, bonds, options, Treasury bills, and cash transactions. Securities can be bought and sold. Stocks pay dividends; they may be split, and their cost bases may be changed. Treasury bills are redeemed when they mature, and bonds pay interests. The program handles different kinds of interest payments: government, municipal, and others.

PM takes in an ASCII file of transactions and produces three files summarizing the portfolio that results from the transactions. It also accepts two numbers: the year for which the summary is to be used for filing tax returns, and the holding period, for the purpose of calculating long and short term capital gains. The holding period is measured in days, and is greater than zero but less than or equal to 365. It produces a tax file that shows, for each security, the different kinds of interest payments received, the dividends received, and the capital gains realized for the tax year. It also sums up the respective categories of annual income. PM produces a second file listing the securities that are still being held along with their cost bases. A third file lists not only the breakdowns for the income in the tax year, but also the cumulative income for the transactions.

Figure 4-1 shows an example input file to PM. The first line is taken to be a documentation string, intended to identify the user and the investment account. Subsequent lines record transactions grouped according to the securities involved. Each group of transactions is sorted by the transaction date. The figure shows four groups of transactions. Each line records a transaction and has a fixed format, with each field separated by a space. It shows in order: the security name, the transaction kind encoded by one or two characters, the amount of the transaction, the unit price of the security, the net of the transaction, the transaction date, the lot number, and the comment field. For certain transaction kinds, some fields may be empty. This shows up as consecutive blank spaces.

The second line in Figure 4-1 shows a transaction buying ten shares of `CommonX` stock at the share price of one dollar a share. This gives a net of ten dollars, and the transaction took place on January 1^{st}, 1992. The transaction for this security is designated lot number one. The rest of the line, `from John`, is taken to be a comment string. The second transaction is similar to the first except that the share price has doubled on the same day. To distinguish the two different buys, there is a unique lot number associated with each transaction of the same security. In this case, the second buy transaction of `CommonX` is designated lot number two. The lot numbers do not have to be in any order, but they must be unique within the buy transactions of the same security.

The fifth line of Figure 4-1 shows a sell transaction. Its format is similar to that of a buy transaction except that multiple lots may be sold. The lots in the sell transaction identify the specific lots of the security sold. This is significant for reporting capital gains in tax returns. The order of the lots recorded is also important because partial lots may be sold. The interpretation of the lot field of a sell transaction is as follows: the complete amount of all lots but the last one in a sell must be sold, but part or all of the last lot can be sold. In this sell transaction, all of lot number three is sold and half of lot number one is sold. This can be computed from the amount of the sell transaction. The sixth line in Figure 4-1 shows a buy transaction of `CommonY` security. Its transaction date is `LT`, which stands for long term. The special date is used to record buy transactions that took place an indefinitely long time in the past.

A key requirement of the PM program is the consistency checking of input transactions. Since PM users may accidentally mistype security names, dates, and the various amounts, PM performs checks on user inputs. For example, for a buy transaction, PM requires the user to supply the number of shares, the price of each share, as well as the net of the transaction. It checks that the product of the amount and the price is sufficiently close to the net. PM also checks that users do not sell securities they do not own.

Other transaction kinds illustrated in Figure 4-1 include capital distribution of security `CommonY`, the municipal interest payment of `MuniZ`, and the Treasury bill maturity of `TBill92`. A few other kinds of transactions supported by PM are not shown in the figure. Some are discussed later in the chapter.

```
Income Yang Meng Tan, Acct 1 Printed \today
CommonX 0.00 0.00 0.00 0.00 35.00 Sold~25.00~netting~0.00~on~4/1/93~
      ~~~LT~Gain~of~20.00~{\it~vs}.~Purchase~of~0.00~costing~0.00~on~3/1/92
      ~~~LT~Gain~of~15.00~{\it~vs}.~Purchase~of~0.00~costing~0.00~on~1/1/92
MuniZ 0.00 0.00 4700.00 0.00 9887.00 Sold~1000.00~netting~0.00~on~1/01/93~
      ~~~LT~Gain~of~9887.00~{\it~vs}.~Purchase~of~0.00~costing~0.00~on~11/1/82
TBill92 0.00 0.00 0.00 100.00 0.00
TOTAL 0.00 0.00 4700.00 100.00 9922.00
```

Figure 4-2: Output tax file of the PM program.

Figure 4-2 and Figure 4-3 show two output files generated by the PM program from the input given in Figure 4-1.[1] They are not intended to be read directly the user. A separate formatting program, not specified in the case study, turns them into LaTeX sources from which prettier outputs are generated. The formatted output corresponding to Figure 4-2 and Figure 4-3 are shown in Figure 4-4 and Figure 4-5.[2]

---

[1] The current tax year of 93 and the holding period of 182 are used to generate the output files.

[2] In Figure 4-4, the the display of the transaction field is elided for brevity.

```
Open Lots Yang Meng Tan, Acct 1 Printed \today
CommonX 5 1.00 5 1/1/92 from~John
CommonX 10 2.00 20 1/1/92
\cline{2-4}
~ 15 1.67 25
\halfline
CommonY 10 0.80 8 LT
\halfline
TOTAL   33.00
```

Figure 4-3: Output open lots file of the PM program.

Income Yang Meng Tan, Acct 1 Printed June 23, 1994

Security	Div	Tax Int	Muni Int	Gov't Int	Cap Gn	Transactions
CommonX	0.00	0.00	0.00	0.00	35.00	Sold ...
MuniZ	0.00	0.00	4,700.00	0.00	9,887.00	Sold ...
TBill92	0.00	0.00	0.00	100.00	0.00	
TOTAL	0.00	0.00	4,700.00	100.00	9,922.00	

Figure 4-4: Processed output tax file of the PM program.

Open Lots Yang Meng Tan, Acct 1 Printed June 23, 1994

Security	Amt	Cost/Item	Cost Basis	Date	Comments
CommonX	5	1.00	5	1/1/92	from John
CommonX	10	2.00	20	1/1/92	
	15	1.67	25		
CommonY	10	0.80	8	LT	
TOTAL			33.00		

Figure 4-5: Processed output open lots file of the PM program.

4.2 The Design of the PM Program

The PM program is made up of the following basic modules. The `security` module models financial securities. The `date` module hides the representation of transaction dates. The `lot_list` module hides the representation of a lot and supports operations on lists of lots. The `trans` module represents transactions, and the `trans_set` module supports operations on sets of `trans`'s. The `genlib` module collects a few useful supporting operations for the program. The `format` module supports the printing routines for generating output files.

The central module of the program is the `position` module. It is built out of the above mentioned modules. A position summarizes a snapshot of the current portfolio. It is updated by new transactions. It contains all the relevant information needed to generate the three output files. It contains the income breakdowns for the tax year, the cumulative income breakdowns, and the open lots of the portfolio. The open lots of a position are the lots owned by the user, that is, the lots that have been bought but have not been sold.

The specification case study consists of the LCL specifications for the following interfaces: `genlib`, `date`, `security`, `lot_list`, `trans`, `trans_set`, and `position`. The following are the main traits supporting the interfaces: `genlib`, `date`, `security`, `lot`, `lot_list`, `kind`, `trans`, `trans_set`, `income`, and `position`. The complete list of traits used by the interfaces are given in Appendix D. These traits use traits from the Larch LSL handbook described in [15]. They are briefly mentioned where they are used in the case study.

Our specifications exclude the `format` module because its specification is tedious and it does not offer significant utility. A good specification of a program does not necessarily specify every detail of the program. It is adequate for the purposes the specification is intended. In our case, the specification is intended to formally document the design of the PM program so that the specifier can analyze the design, and the implementor can make use of the design to implement the main modules. The `format` module deals with what we consider to be secondary issues of pretty-printing the output of the program.

Since many specifications in the case study are straightforward, they are not discussed in this chapter. The rest of the chapter presents specifications of the following four modules: `date`, `trans`, `trans_set`, and `position`.

4.3 The date Interface

The `date` interface, in Figure 4-6, exports an immutable abstract type `date`. The date interface uses the `date` trait.

The `date` interface exports nine functions. The `date_parse` function parses an input string and returns a boolean flag indicating whether the parse is successful. It returns the parsed date via a pointer passed from the caller. It takes another string that is used only if an error occurs. This latter string is intended to be a line from a user's input from which the date string is extracted. The `create_null_date` function creates a special date. The other functions are observers of the date type.

The `date` interface imports the `genlib` interface which defines a number of useful exposed types and exports some generic library functions. Pertinent to the `date` interface is the introduction of two exposed types, `cstring` and `nat`, which have constraints associated with them. The specification of `cstring` from the `genlib` interface is shown below:

```
imports genlib;
immutable type date;
uses date (cstring for String);

bool date_parse (cstring indate, cstring inputStr, out date *d) FILE *stderr; {
  let dateStr be getString(indate^),
      fileObj be *stderr^;
  modifies *d, fileObj;
  ensures result = okDateFormat(dateStr)
          ∧ if result then (*d)′ = string2date(dateStr) ∧ unchanged(fileObj)
             else ∃ errm: cstring (appendedMsg(fileObj′, fileObj^,
                                                 inputStr^ || errm));
}
date create_null_date (void) {
  ensures result = null_date;
}
nat date_year (date d) {
  checks isNormalDate(d);
  ensures result = year(d);
  claims result ≤ 99;
}
bool is_null_date (date d) {
  ensures result = (d = null_date);
}
bool date_is_LT (date d) {
  ensures result = isLT(d);
}
bool date_same (date d1, date d2) {
  ensures result = (d1 = d2);
}
bool date_is_later (date d1, date d2) {
  ensures result = (d1 > d2);
}
bool is_long_term (date buyD, date sellD, nat hp) {
  checks isNormalDate(buyD) ∧ isNormalDate(sellD);
  ensures result = (buyD ≤ sellD ∧ hp ≤ 365
                    ∧ ((year(sellD) - year(buyD)) > 1 ∨ (sellD - buyD) > hp));
}
char *date2string (date d) {
  let res be getString(result[]′);
  ensures fresh(result[]) ∧ nullTerminated(result[]′)
          ∧ (isNormalDate(d) ⇒ res = date2string(d))
          ∧ (isLT(d) ⇒ res = "LT")
          ∧ (isNullDate(d) ⇒ res = "null");
}
```

Figure 4-6: date.lcl.

```
typedef char cstring[] {constraint ∀ s: cstring (nullTerminated(s))};
```

The specification defines `cstring` to be an abbreviation for the C type `char []` and associates a constraint with the type name `cstring`. It does not define a new type; it is only a shorthand for associating a constraint with an exposed type. The LSL operator `nullTerminated` is defined in the `cstring` trait; `nullTerminated(s)` is true when the character string `s` contains a null character. Hence, the specification codifies the C convention of treating character arrays as strings.

The `cstring` type provides a compact way of specifying operations involving C strings. For example, in Figure 4-6, the specification of the `date_parse` function shows that two of its formal parameters (`indate` and `inputStr`) have the `cstring` type. The use of `cstring` as the type of a parameter implicitly adds the constraint to the specification:

```
requires nullTerminated(indate^) ∧ nullTerminated(inputStr^);
```

If the type appears as the output of a function specification, its semantics is to add the corresponding constraint to the ensures clause to the specification:

```
nullTerminated(result')
```

The `getString` operator is often used with C strings to extract the string content of a C character array. It is defined in the `string` trait given in Appendix D.

Since C functions cannot return multiple values directly, a common C programming idiom is to return a value indirectly via a pointer passed to the function, like the date pointer in `date_parse`. The `out` parameter type qualifier, which is applicable only to pointer types, formalizes the idiom. It indicates to the implementor that in the pre state, what an `out` pointer points to is storage that is allocated but not necessarily initialized. It is an error to use the initial value of an out pointer. The distinction between out pointers and non-out pointers is important in inductive reasoning. For example, it means that `date_parse` can be treated as a primitive constructor for the `date` abstract type. A primitive constructor for a type `T` builds an instance of `T` from other non-`T` types. Primitive constructors form the bases for deriving inductive properties of abstract types.

The modifies clause of `date_parse` says that the input date pointer may be made to point to a different date and that the standard error stream may be modified. The ensures clause of `date_parse` illustrates a specification technique for avoiding a common specification mistake: over-specification. In the case of a bad date string, the specification of `date_parse` requires that the input line and some error message be written out to the standard error stream. It does not constrain the details of the error message. It gives the implementor of `date_parse` more freedom in generating the error message.[3]

The dual of the over-specification mistake is under-specification. Omissions in a specification say as much as the explicit constraints the specification states. The specification of `date_parse2`, in Figure 4-7, is identical to that of `date_parse` in Figure 4-6, except for the conditional expression in the ensures clause. The assertion `unchanged(fileObj)` is omitted in the consequent clause of the conditional. This means that the implementor of

---

[3] There are, of course, situations in which more exact specifications of error messages are appropriate.

```
bool date_parse2 (cstring indate, cstring inputStr, out date *d) FILE *stderr; {
  let fileObj be *stderr^,
      dateStr be getString(indate^);
  modifies *d, fileObj;
  ensures result = okDateFormat(dateStr)
          ∧ if result then (*d)' = string2date(dateStr)
            else ∃ errm: cstring (appendedMsg(fileObj', fileObj^,
                                              inputStr^ || errm);
}
```

Figure 4-7: An example of under-specification.

`date_parse2` is free to print error messages to the standard error stream even when the date string has the right format. Unchanged assertions are often omitted inadvertently.

The specification of `date_year` in Figure 4-6 returns a result that has type `nat`. Like `cstring`, `nat` is an exposed type with a constraint defined in the `genlib` interface:

```
  typedef long nat {constraint ∀ n: nat (n ≥ 0)};
```

The type `nat` is defined to be the integers that are greater than or equal to 0, or the natural numbers. Using `nat` in the specification of `date_year` allows the specification to be more compact: it specifies that the function should return an integer that is non-negative.

The specification of `date_year` also illustrates the use of the `checks` clause in LCL. The function is designed to be used only on a normal date, not on `LT` or `null_date`. The checks clause is a convenient shorthand for specifying conditions that the implementor must check. If the conditions are not met, the implementor should print an error message, and halt the program without modifying any other objects. That is, the semantics of an LCL function specification with the checks clause is:

```
RequiresP ⇒
 (ModifiesP
  ∧ if ChecksP then EnsuresP ∧ StdErrorChanges
    else halts ∧ ModifiesP
         ∧ ∃ errm: cstring (appendedMsg((*stderr^)', (*stderr^)^,
                                        FatalErrorMsg || errm)))
```

where `RequiresP` stands for the requires clause of the function, `ChecksP`, the checks clause, `ModifiesP`, the translation of the modifies clause, and `EnsuresP`, the ensures clause. `stderr` is C's standard error file pointer. The object `*stderr^` is implicitly added to the modifies clause and the list of global variables accessible by the function. `StdErrorChanges` is defined to be `true` if the specifier explicitly adds `*stderr^` to the modifies clause or if the checks clause is absent, and `unchanged(*stderr^)` otherwise. This semantics allows a specifier to override the default assumption that the standard error stream is unchanged if the checks clause holds by explicitly adding the standard error stream on the modifies clause. An omitted checks clause means `ChecksP = true`.

The specification of `date_year` could be written without using the checks clause, as illustrated in Figure 4-8. The checks clause, however, codifies a common specification idiom. It is useful for specifying checks that are aimed at detecting a class of *programmer errors*:

```
nat date_year (date d) FILE *stderr; {
  let fileObj be *stderr^;
  modifies fileObj;
  ensures if isNormalDate(d)
          then result = year(d) ∧ unchanged(fileObj)
          else ∃ errm: cstring (appendedMsg(fileObj′, fileObj^, errm));
}
```

Figure 4-8: Specification of date_year without the checks clause.

a client programmer calls a function without ensuring that the conditions of the call are respected. An alternative is to use the requires clause to outlaw such calls. The semantics of the requires clause, however, is very weak: the implementor is free to do anything if the requires clause is false. For example, the function is not required to terminate. It is desirable to take a more defensive approach whenever feasible. The checks clause encourages such defensive design by making the specification more concise.

Besides preventing programmer errors from doing damage to the program state, a function often has to check for errors in the inputs it ordinarily receives from the user. We term such errors *user errors*. For example, the specification of `date_parse` requires the function to check that the date string has an appropriate format. Otherwise, an error message is generated and the function returns normally. By separating programmer errors from input errors, the checks clause makes a specification easier to read and understand.

4.4 The date Traits

The theory formalizing dates is constructed in three traits. The `dateBasics` trait in Figure 4-9 codifies the meanings of operators on *normal* dates. A normal date is a tuple of month, day, and year. Our PM program requires two other special encodings of dates: a special date given as `"LT"`, for long term, and `null_date`, which is used to mark an uninitialized position. The `date` trait in Figure 4-11 extends date operators to handle these two special dates.

A few operators in the `dateBasics` trait need brief mention. The `dayOfYear` of a date gives the ordinal of the date in a year. For example, the `dayOfYear` of January 1^{st} is 1, and the `dayOfYear` of February 2^{nd} is 32. The `daysToEnd` of a date is the complement of the day of year: it gives the number of days until the end of the year. For example, the `daysToEnd` of December 31^{st} is 0.

The normal dates codified by the `dateBasics` trait has one unorthodox aspect: the month or the day of a date may be zero. For example, `"0/0/93"` represents the indeterminate month in the year 1993 and `"1/0/93"` represents the indeterminate day in January 1993. The indeterminate month of a year is arbitrarily chosen to be before January in the year with respect to the `<: date, date → Bool` strict total order. Similarly, the indeterminate day of a month and year is chosen to be before the first day of the month and year. We do not constrain dates with a zero month but a non-zero day.

The `date` trait uses a supporting trait named `dateFormat`. The `dateFormat` trait,

```
dateBasics: trait
 includes Integer, TotalOrder (date)
 date tuple of month, day, year: Int % unknown month is 0, jan 1, ... dec 12.
 introduces
  isInLeapYear: date → Bool
  isLeapYear: Int → Bool
  validMonth: Int → Bool
  __ - __ : date, date → Int
  daysBetween: Int, Int → Int
  dayOfYear, daysToEnd: date → Int
  dayOfYear2: Int, Int, Int, Int → Int
  daysInMonth: Int, Int → Int
 asserts ∀ d, d2: date, k, m, yr, yr2: Int, mth, mth2: Int
   isInLeapYear(d) == isLeapYear(d.year);
   isLeapYear(yr) == mod(yr, 400) = 0 ∨ (mod(yr, 4) = 0 ∧ mod(yr, 100) ≠ 0);
   validMonth(mth) == mth ≥ 0 ∧ mth ≤ 12;
   d < d2 == d.year < d2.year
             ∨ (d.year = d2.year ∧ dayOfYear(d) < dayOfYear(d2));
   d ≥ d2 ⇒
    d - d2 = (if d.year = d2.year then dayOfYear(d) - dayOfYear(d2)
              else daysToEnd(d2) + dayOfYear(d) +
                    daysBetween(succ(d2.year), d.year));
   yr ≤ yr2 ⇒
    daysBetween(yr, yr2) =
     (if yr = yr2 then 0
      else (if isLeapYear(yr) then 366 else 365) + daysBetween(succ(yr), yr2));
   (validMonth(d.month) ∧ (d.month ≠ 0 ∨ d.day = 0)) ⇒
    dayOfYear(d) = (if d.month = 0 then 0
                    else dayOfYear2(d.month, 1, d.day, d.year));
   (validMonth(mth) ∧ validMonth(mth2)) ⇒
    dayOfYear2(mth, mth2, k, yr) =
     (if mth = mth2 then k
      else dayOfYear2(mth, succ(mth2), k + daysInMonth(mth2, yr), yr));
   validMonth(d.month) ⇒
    daysToEnd(d) = (if isInLeapYear(d) then 366 else 365) - dayOfYear(d);
   (validMonth(mth) ∧ mth ≠ 0) ⇒
    daysInMonth(mth, yr) =
     (if mth = 2 then if isLeapYear(yr) then 29 else 28
      else if mth = 1 ∨ mth = 3 ∨ mth = 5 ∨ mth = 7 ∨ mth = 8 ∨ mth = 10
              ∨ mth = 12
           then 31 else 30);
 implies
  converts isInLeapYear, isLeapYear
```

Figure 4-9: dateBasics.lsl.

```
dateFormat: trait
 includes genlib, dateBasics
 introduces
  okDateFormat, isNormalDateFormat: String → Bool
  validDay: Int, Int, Int → Bool
 asserts ∀ s: String, i, m, yr: Int
   okDateFormat(s) == (len(s) = 2 ∧ s[0] = 'L' ∧ s[1] = 'T')
                      ∨ isNormalDateFormat(s);
   isNormalDateFormat(s) == (len(s) ≥ 5) ∧ (len(s) ≤ 8)
     ∧ countChars(s, 'slash') = 2 ∧ NthField(s, 1, 'slash') != empty
     ∧ isNumeric(NthField(s, 1, 'slash'))
     ∧ validMonth(string2int(NthField(s, 1, 'slash')))
     ∧ NthField(s, 2, 'slash') != empty
     ∧ isNumeric(NthField(s, 2, 'slash'))
     ∧ NthField(s, 3, 'slash') != empty
     ∧ isNumeric(NthField(s, 3, 'slash'))
     ∧ validDay(string2int(NthField(s, 2, 'slash')),
                string2int(NthField(s, 1, 'slash')),
                string2int(NthField(s, 3, 'slash')));
   validDay(i, m, yr) == (i ≥ 0) ∧ (i ≤ 31)
     ∧ ((m = 0 ∧ i = 0) ∨ % reject 0/non-0-day/yr format
        (m > 0 ∧ m ≤ 12 ∧ i ≤ daysInMonth(m, yr)));
  implies converts okDateFormat, isNormalDateFormat, validDay
```

Figure 4-10: dateFormat.lsl.

in Figure 4-10, includes the `dateBasics` trait and the `genlib` trait. It defines the string format of a date in the PM program. The `isNormalDateFormat` operator defines what input date format is acceptable to PM. It accepts dates written in the style of `"mm/dd/yr"`. Days, months and years that are less than ten can (but need not) be prefixed by zero. For example, the following are examples of valid dates: `1/1/1`, `01/01/01`, `12/31/93`. Indeterminate dates are acceptable date formats except that a date with a zero month must have zero as its day.

Figure 4-11 gives the LSL specification of the `date` trait. It includes the supporting trait `dateFormat` trait in Figure 4-10, but with `date` renamed to `ndate`. It also includes the Larch handbook trait `TotalOrder`, which introduces and mutually constrains the following four operators on dates: `<=, <, >, >=:date, date → Bool`. The `TotalOrder` trait also defines `<=:date, date → Bool` as a total order.

The `date` trait defines the sorts and operators needed to specify the `date` interface. A `date` sort is a tagged union of a normal date, with the tag `normal`, and a boolean with the tag `special`.

The semantics of most operators in the `date` trait are ordinary. There is one key difference from ordinary interpretation of dates: the interpretation of the year in a date string is unusual. It is traditional to have a two-digit encoding of the year. The impending turn of the century, however, imposes some difficulties. Should the year encoding `00` indicate the year 1900 or the second millennium? In this date trait, the latter interpretation is used. The difference shows up in the definition of the supporting operator `fixUpYear`.

```
date: trait
 includes dateFormat (ndate for date), TotalOrder (date)
 date union of normal: ndate, special: Bool
 introduces
  null_date: → date % serves as an uninitialized date.
  isLT, isNullDate, isNormalDate, isInLeapYear: date → Bool
  year: date → Int
  __ - __ : date, date → Int
  is_long_term: date, date, Int → Bool
  string2date: String → date
  date2string: date → String
  fixUpYear: Int → Int
 asserts ∀ d, d2: date, nd: ndate, s: String, i, day, yr: Int
   null_date == special(false);
   isNullDate(d) == d = null_date;
   isLT(d) == tag(d) = special ∧ d.special;
   isNormalDate(d) == tag(d) = normal;
   isNormalDate(d) ⇒ isInLeapYear(d) = isInLeapYear(d.normal);
   isNormalDate(d) ⇒ year(d) = d.normal.year;
   (isNormalDate(d) ∧ isNormalDate(d2)) ⇒ (d - d2 = d.normal - d2.normal);
   (isNormalDate(d) ∧ isNormalDate(d2)) ⇒
    is_long_term(d, d2, i) = ((d.normal - d2.normal) > i);
   (isNormalDate(d) ∧ isNormalDate(d2)) ⇒ (d < d2 = d.normal < d2.normal);
   (isLT(d) ∧ isNormalDate(d2)) ⇒ (d < d2);
   null_date < d == not(d = null_date); % non-reflexive
   okDateFormat(s) ⇒
    string2date(s) =
     (if (len(s) = 2 ∧ s[0] = 'L' ∧ s[1] = 'T') then special(true)
      else normal([string2int(NthField(s, 1, 'slash')),
                   string2int(NthField(s, 2, 'slash')),
                   fixUpYear(string2int(NthField(s, 3, 'slash')))]));
   yr ≥ 0 ⇒ fixUpYear(yr) = (if yr < 50 then 2000 + yr else 1900 + yr);
   isNormalDate(d) ⇒ string2date(date2string(d)) = d;
  implies
    ∀ d: date
     isNormalDate(d) ⇒ dayOfYear(d.normal) + daysToEnd(d.normal) =
                         (if isInLeapYear(d) then 366 else 365)
```

Figure 4-11: date.lsl.

4.5 The trans Traits

The `trans` trait defines the format and interpretation of the inputs accepted by the PM program. It is shown in Figure 4-12, and is built out of three supporting traits. It includes the `transParse` trait which describes how an input string representing a transaction is parsed and converted into a transaction. The `transParse` trait in turn includes the `transFormat` trait which defines the valid format of a string representing a transaction. All three traits directly or indirectly include the `transBasics` trait shown in Figure 4-13. The `transFormat` and `transParse` traits are given in Appendix D and are not discussed here.

In Figure 4-13, the `transBasics` trait defines a key abstraction used in the `trans` interface. A `trans` is a tuple of a security, a transaction kind, the amount, price, and net of the transaction, the transaction date, a list of lots, and two documentation strings.

In Figure 4-12, the `transIsConsistent` operator codifies the non-syntactic constraints a transaction must maintain. They are mostly numerical constraints on the fields of a transaction. The constraints are different for each transaction kind. The `<=` operator imposes a partial order on `trans`. It is useful for supporting the processing of transactions in order. It is lexicographically derived from the order on securities, and the order on dates.

Specifications are intended to be read by humans. The first and foremost criterion of a good specification is readability. As such, attention should be paid to making it easy for humans to read and understand. In particular, there is a style of LSL specifications that make traits easier to read. Observe the separate equations that jointly define `transIsConsistent` in Figure 4-12. It is an algebraic style of defining a function acting on arguments of disjoint cases one at a time, each by a separate equation. A different specification style using a deeply nested conditional would make the specification more difficult to read.

4.6 The trans Interface

The `trans` interface shown in Figure 4-14 exports two types: `kind`, an exposed type, and `trans`, an abstract type. The `kind` interface illustrates the use of LCL exposed types. In Figure 4-14, the type `kind` is defined to be a C enumeration of the following transaction kinds: buy, sell, dividend payment, capital distribution, maturity of a Treasury bill, security exchange, ordinary interest payment, municipal interest payment, government interest payment, new security, and other. A `new_security` kind is used to introduce the name of a security. The `other` kind is used to indicate an error in the creation of a transaction.

The trans type could be specified as a C struct type. An abstract type is used instead because doing so limits the kind of changes the client can make to `trans` instances. This makes it possible for the interface to maintain invariants about the trans type that would be impossible if an exposed type were used. For example, an invariant that the `trans` interface maintains is: a buy transaction has non-negative net, amount, and price, and the product of its amount and its price is within one of its net. Such invariants are useful because they serve as checks on the intermediate values the program calculates. In addition, an abstract type is preferred because a client does not need to know the specific implementation of the trans type. The `kind` type is specified as an exposed type because it has no useful invariants, and making it abstract involves exporting many trivial interfaces.

The two types could have been specified in separate modules. We choose to put them in

```
trans (String): trait
 includes transParse
 introduces
  transIsConsistent: trans, kind → Bool
  __ ≤ __ : trans, trans → Bool
 asserts ∀ t, t2: trans
   transIsConsistent(t, buy) == t.net ≥ 0 ∧ t.amt > 0 ∧ t.price ≥ 0
     ∧ length(t.lots) = 1 ∧ within1(t.amt * t.price, t.net);
    % sell amount may be 0 to handle special court-ordered settlements.
    % also cannot give away securities for free.
   transIsConsistent(t, sell) == t.net > 0 ∧ t.amt ≥ 0 ∧ t.price > 0
    ∧ isNormalDate(t.date) ∧ uniqueLots(t.lots)
    ∧ (t.amt > 0 ⇒ within1(t.amt * t.price, t.net));
   transIsConsistent(t, cash_div) == t.amt > 0;
   transIsConsistent(t, exchange) == t.amt > 0 ∧ length(t.lots) = 1;
   transIsConsistent(t, cap_dist) == t.net > 0 ∧ t.amt > 0
                                    ∧ length(t.lots) = 1;
   transIsConsistent(t, tbill_mat) == t.net > 0 ∧ t.amt > 0
                                     ∧ uniqueLots(t.lots);
    % negative interests arise when bonds are purchased between their interest
    %  payment periods.
   transIsConsistent(t, interest);
   transIsConsistent(t, muni_interest);
   transIsConsistent(t, govt_interest);
   transIsConsistent(t, new_security);
   ¬ transIsConsistent(t, other);
   t ≤ t2 == (t.security < t2.security)
             ∨ (t.security = t2.security ∧ t.date ≤ t2.date);
 implies converts transIsConsistent, __ ≤ __: trans, trans → Bool
```

Figure 4-12: trans.lsl.

the same module because the two types go together conceptually: every client of the kind type is a client of the trans type, and vice versa.

The `trans` interface exports an integer constant `maxInputLineLen`. This represents the maximum length of the input line representing a transaction. The key function the interface exports is `trans_parse_entry`, which converts a valid C string into a `trans`. The function returns true if the string has an acceptable format and the information given in the string corresponds to a valid `trans`. The abbreviations in an LCL `let` clause nest, that is, an earlier abbreviation can be used in a later one.

Besides `trans_parse_entry`, two other functions in the interface can create `trans` instances: `trans_adjust_net` and `trans_adjust_amt_and_net`. The latter function can achieve its effect if it is given either the new amount or the new net. It takes both as arguments in order to check that they are consistent with the old price. Both functions are specified to maintain the invariants. The other functions exported by the `trans` interface are simple observers of transactions.

```
transBasics: trait
 includes genlib, date, kind, security, lot_list
 trans tuple of security: security, kind: kind, amt, price, net: double,
               date: date, lots: lot_list, input: String, comment: String
```

Figure 4-13: transBasics.lsl.

4.7 The trans_set Interface and Trait

The `trans_set` interface supports operations on sets of `trans`'s that are buy transactions. It is shown in Figure 4-15 and is built using the `trans_set` trait shown in Figure 4-16. The specification of the `trans_set` interface is adapted from a similar example in Chapter 5 of [15]. The basic LSL trait for modeling an iterator is reused, and the approach of specifying different functions for coordinating the iteration process in C is also adopted. There are minor differences in the way clients use the interfaces. The specification of `trans_set` illustrates an important principle in writing specifications: specifications should be reused whenever appropriate.

The `trans_set` interface defines two mutable types: `trans_set` and `trans_set_iter`. Together, these two types support ordinary set operations and iterations over sets. A `trans_set_iter` records the state of a set iteration. In Figure 4-16, it is modeled as a pair consisting of a `trans_set` object and the elements of the set that have yet to be yielded. It supports multiple simultaneous iterations over a `trans_set`, such as those occurring in nested loops.

The syntax `obj trans_set` in the `uses` construct in Figure 4-15 needs some explanation. An LCL mutable type is modeled by two underlying sorts: an *object sort* that represents the object identity of a mutable object, and a *value sort* that represents the value of the mutable object in some state. By default, an LCL type in a `uses` clause is implicitly mapped to the value sort of the type, unless the `obj` type qualifier is used. The `uses` clause in Figure 4-15 says that the object sort modeling the `trans_set` LCL type should be used to rename the second parameter of the `trans_set` trait, that is, the `trans_set_obj` sort in the trait. A more detailed explanation of the implicit mapping of LCL types to LSL sorts is given in Section 7.3.2.

In Figure 4-15, the first five functions support set operations to create, modify, copy, and destroy a set object. The `trans_set_insert` function adds a `trans` only if the `trans` is single-lot and if there are no elements already in the set with the matching security and lot. This ensures that each element in a `trans_set` has a unique identifier made up of its security and its lot. The trans_set type is designed to represent the open lots of a security. The open lots of a security has the uniqueness property. The `trans_set_delete_match` removes all elements with the matching security and lot. The `trans_set_free` should only be called with a set object that will never be referenced again. The `trashed(s)` assertion indicates that nothing can be assumed about the object `s` upon the return of the function. The function is used to deallocate storage occupied by set objects.

The last four functions exported by the `trans_set` interface in Figure 4-15 together supports iteration over `trans_set`'s. The `trans_set_iter_start` function takes a `trans_set`

```
imports security, date, lot_list;
typedef enum {buy, sell, cash_div, cap_dist, tbill_mat, exchange, interest,
              muni_interest, govt_interest, new_security, other} kind;
immutable type trans;
constant nat maxInputLineLen;
uses trans (cstring, kind for kind);

bool trans_parse_entry (cstring instr, out trans *entry) FILE *stderr; {
  let input be prefix(getString(instr^), maxInputLineLen),
      parsed be string2trans(input),
      fileObj be *stderr^;
  modifies *entry, fileObj;
  ensures result = (okTransFormat(input)
                    ∧ transIsConsistent(parsed, parsed.kind))
          ∧ if result then (*entry)′ = parsed ∧ unchanged(fileObj)
             else ∃ errm: cstring (appendedMsg(fileObj′, fileObj^, errm));
}
trans trans_adjust_net (trans t, double newNet) {
  checks t.kind = buy ∧ newNet ≥ 0;
  ensures result = set_price(set_net(t, newNet), newNet/t.amt);
}
trans trans_adjust_amt_and_net (trans t, double newAmt, double newNet) {
  checks t.kind = buy ∧ within1(newNet/newAmt, t.price) ∧ newNet ≥ 0
         ∧ newAmt > 0;
  ensures result = set_amt(set_net(t, newNet), newAmt);
}
bool trans_match (trans t, security s, lot e) {
  ensures result = (t.security = s ∧ length(t.lots) = 1 ∧ car(t.lots) = e);
}
bool trans_less_or_eqp (trans t1, trans t2) {
  ensures result = (t1 ≤ t2);
}
double trans_get_cash (trans entry) {
  ensures if isCashSecurity(entry.security) ∧ entry.kind = buy
          then result = entry.net
          else result = 0;
}
char *trans_input (trans t) {
  ensures nullTerminated(result[]′) ∧ getString(result[]′) = t.input
          ∧ fresh(result[]);
}
char *trans_comment (trans entry) {
  ensures nullTerminated(result[]′) ∧ getString(result[]′) = entry.comment
          ∧ fresh(result[]);
}
lot_list trans_lots (trans entry) {
  ensures result′ = entry.lots ∧ fresh(result);
}
security trans_security (trans entry) {
  ensures result = entry.security;
}
```

Figure 4-14: trans.lcl, part 1

```
kind trans_kind (trans entry) {
  ensures result = entry.kind;
}
double trans_amt (trans entry) {
  ensures result = entry.amt;
}
double trans_net (trans entry) {
  ensures result = entry.net;
}
date trans_date (trans entry) {
  ensures result = entry.date;
}
bool trans_is_cash (trans entry) {
  ensures result = isCashSecurity(entry.security);
}
```

Figure 4-14: trans.lcl, part 2.

object and returns a `trans_set_iter` object in which the set to be yielded is the value of the `trans_set` object. It also increments the number of active iterators associated with the `trans_set` object by one. When the function `trans_set_yield` is called with the `trans_set_iter` object, it produces an element of the set and updates the object by removing the element from the set of elements yet to be yielded. The function should only be called if there are still elements to be yielded. The function `trans_set_iter_more` tells if there are more elements in a `trans_set_iter` to be yielded. For each call to `trans_set_iter_start`, a client of the `trans_set` module is expected to call a matching `trans_set_iter_final` function which restores the state of the `trans_set` object back to its original state before the iteration started. A typical way to use them is illustrated in the code fragment shown in Figure 4-17.

There is one feature in the design of the `trans_set` operations that differs from ordinary set interfaces: `trans_set` mutators can only be called on a `trans_set` object if the set is not being iterated over. The specifications of the set mutators use the checks clause to make sure that the requirement is met. The `trans_set` trait in Figure 4-16 defines a `trans_set` to be a pair consisting of a `tset` and an integer indicating the number of iterations that are currently being performed on the `trans_set`.

It is harder to specify an iterated set that allows mutations in the midst of an iteration; an example can be found in [14]. The constraint on set iteration enables more efficient implementations of such iterated sets.

In the `trans_set` trait, three operators are defined to support the specification of the `position` module: `sum_net`, `sum_amt`, and `findTrans`. The first two sum up the net and amount fields of the elements in a `trans_set`. The `findTrans` operator finds a `trans` in a `trans_set` with the matching security and lot.

The specification of `findTrans` illustrates a potentially subtle and common error in LSL specifications, i.e., definitions by induction when the generators are not free. Consider a similar-looking specification of `findTrans` below:

```
imports trans;
mutable type trans_set;
mutable type trans_set_iter;
uses trans_set (cstring, obj trans_set);

trans_set trans_set_create (void) {
  ensures result′ = [{}, 0] ∧ fresh(result);
}
bool trans_set_insert (trans_set s, trans t) {
  checks s^.activeIters = 0;
  modifies s;
  ensures (result = matchKey(t.security, car(t.lots), s^.val)
                    ∧ length(t.lots) = 1)
           ∧ if result then unchanged(s) else s′ = [insert(t, s^.val), 0];
}
bool trans_set_delete_match (trans_set s, security se, lot e) {
  checks s^.activeIters = 0;
  modifies s;
  ensures result = matchKey(se, e, s^.val) ∧ s′.activeIters = 0
          ∧ s′.val ⊆ s^.val
          ∧ ∀ t:trans (t ∈ s^.val ⇒
                          if t.security = se ∧ car(t.lots) = e
                          then ¬ (t ∈ s′.val)
                          else (t ∈ s′.val));
}
trans_set trans_set_copy (trans_set s) {
  ensures fresh(result) ∧ result′ = [s^.val, 0];
}
void trans_set_free (trans_set s) {
  modifies s;
  ensures trashed(s);
}
trans_set_iter trans_set_iter_start (trans_set ts) {
  modifies ts;
  ensures fresh(result) ∧ ts′ = startIter(ts^) ∧ result′ = [ts^.val, ts];
}
trans trans_set_iter_yield (trans_set_iter tsi) {
  checks tsi^.toYield ≠ {};
  modifies tsi;
  ensures yielded(result, tsi^, tsi′)
          ∧ ∀ t: trans (t ∈ tsi′.toYield ⇒ result ≤ t);
}
bool trans_set_iter_more (trans_set_iter tsi) {
  ensures result = (tsi^.toYield ≠ {});
}
void trans_set_iter_final (trans_set_iter tsi) {
  let sObj be tsi^.setObj;
  modifies tsi, sObj;
  ensures trashed(tsi) ∧ sObj′ = endIter(sObj^);
}
```

Figure 4-15: trans_set.lcl

```
trans_set (String, trans_set_obj): trait
 includes trans, Set (trans, tset)
 trans_set tuple of val: tset, activeIters: Int
 trans_set_iter tuple of toYield: tset, setObj: trans_set_obj
 introduces
  yielded: trans, trans_set_iter, trans_set_iter → Bool
  startIter: trans_set → trans_set
  endIter: trans_set → trans_set
  matchKey: security, lot, tset → Bool
  findTrans: security, lot, tset → trans
  sum_net, sum_amt: tset → double
 asserts ∀ t: trans, ts: tset, s: security, e: lot, trs: trans_set,
          it, it2: trans_set_iter
   yielded(t, it, it2) ==
    (t ∈ it.toYield) ∧ it2 = [delete(t, it.toYield), it.setObj];
   startIter(trs) == [trs.val, trs.activeIters + 1];
   endIter(trs) == [trs.val, trs.activeIters - 1];
   ¬ matchKey(s, e, {});
   matchKey(s, e, insert(t, ts)) ==
    (s = t.security ∧ length(t.lots) = 1 ∧ e = car(t.lots))
     ∨ matchKey(s, e, ts);
   matchKey(s, e, ts) ⇒ (findTrans(s, e, ts) ∈ ts
    ∧ car(findTrans(s, e, ts).lots) = e ∧ findTrans(s, e, ts).security = s
    % buy trans has single lots, only interested in matching buy trans
    ∧ length(findTrans(s, e, ts).lots) = 1);
   sum_net({}) == 0;
   t ∈ ts ⇒ sum_net(insert(t, ts)) = sum_net(ts);
   ¬ (t ∈ ts) ⇒ sum_net(insert(t, ts)) = t.net + sum_net(ts);
   sum_amt({}) == 0;
   t ∈ ts ⇒ sum_amt(insert(t, ts)) = sum_amt(ts);
   ¬ (t ∈ ts) ⇒ sum_amt(insert(t, ts)) = t.amt + sum_amt(ts);
 implies converts matchKey, sum_net, sum_amt
```

Figure 4-16: trans_set.lsl

```
trans tr;
 trans_set ts;
 trans_set_iter tsi;
 tsi = trans_set_iter_start(ts);
 while (trans_set_iter_more(tsi)) {
   tr = trans_set_iter_yield(tsi);
   /* body of loop uses tr */
 }
 trans_set_iter_final(tsi);
```

Figure 4-17: Code fragment showing the use of trans_set iterator functions.

```
matchKey(s, e, insert(t, ts)) ⇒
 findTrans(s, e, insert(t, ts)) =
  if (t.security = s ∧ length(t.lots) = 1 ∧ car(t.lots) = e)
  then t else findTrans(ts, e);
```

Since a `tset` is inductively constructed by {} and `insert`, the above definition of `findTrans` is expected. The guard on the definition ensures that it is applied only when a match exists. The sort `tset` is partitioned by the membership test, ∈. This fact comes from the `Set` handbook trait, which includes the `SetBasics` trait shown in Figure 4-18. From the `tset` axioms, we can prove that the order of set insertions does not matter. For example, there are two equivalent term representations of a two-element set: `insert(e1, insert(e2, {}))` and `insert(e2, insert(e1, {}))`. The problem with the definition of `findTrans` lies in its order-dependence: it finds the first matching `trans` in a `tset`. But since two `tset`s may be equal without having the same representation, it is possible to derive a contradiction if both happen to have matching security and lot. This is because we can use the partitioned by axiom to show that two matching `trans` must be equal even though they may disagree with each other in other fields.

```
SetBasics (E, C): trait
  introduces
    {}: → C
    insert: E, C → C
    __ ∈ __: E, C → Bool
  asserts
    C generated by {}, insert
    C partitioned by ∈
    ∀ s: C, e, e1, e2: E
      ¬ (e ∈ {});
      e1 ∈ insert(e2, s) == e1 = e2 ∨ e1 ∈ s
  implies
    InsertGenerated ({} for empty)
    ∀ e, e1, e2: E, s: C
      insert(e, s) ≠ {};
      insert(e, insert(e, s)) == insert(e, s);
      insert(e1, insert(e2, s)) == insert(e2, insert(e1, s))
    converts ∈
```

Figure 4-18: The Larch handbook trait: SetBasics.

An inconsistency can be viewed as an extreme form of over-specification. The mistake discussed above lies in over-constraining the `findTrans` operator. For convenience, we reproduce our specification of `findTrans` from Figure 4-16 below:

```
 matchKey(s, e, ts) ⇒ (findTrans(s, e, ts) ∈ ts
  ∧ car(findTrans(s, e, ts).lots) = e ∧ findTrans(s, e, ts).security = s
  ∧ length(findTrans(s, e, ts).lots) = 1);
```

The above axiom does not define the value of `findTrans`. It only constrains the value of `findTrans(s, e, ts)` to have the relevant properties when `matchKey(s, e, ts)` holds.

The mistake also raises a specification issue that is specific to Larch's two-tiered approach. There are often two theories being developed in the Larch specification process. An LCL interface specification typically uses some underlying LSL trait. Both an LCL interface and the LSL trait it uses define logical theories. The two theories are, in general, different but related. The LSL theory is often strictly weaker than the LCL theory because we can derive inductive properties based on data type induction at the interface level.

The LSL operators specifiers write are often motivated by the design of the interfaces they have in mind. For example, in the design of the `trans_set` interface, a `trans_set` has the special property that its elements have unique security and lot fields. This is maintained as an invariant of the interface. This means that our order-dependent definition of `findTrans` is not wrong for the *interface theory* since, at the interface level, `findTrans` would yield a unique `trans` for the `trans_set`'s that maintained the invariant. Unfortunately, `findTrans`, defined as an LSL operator, introduces the inconsistency in the LSL theory discussed in the preceding paragraphs. Failure to appreciate the distinction between the two theories contributes to the mistake.

A special `trans_set` trait that has the property that its `tset` elements have unique security and lot fields could be written. This approach is not taken because it tends to be less robust. It is easy for interface specifications to violate such invariants, and it is difficult to detect such inconsistencies. Further discussion of this issue is given in Section 5.10.

4.8 The position Traits

The `position` trait defines the processing of valid transactions for the PM program. It is the largest trait in the case study. In this section, we describe the overall function and structure of the trait; the complete specification is given in Appendix D.

The `position` module performs most of the processing the PM program is required to do. The specification of the `position` trait is large because many different kinds of transactions have to be supported. The specification is structured into seven different parts to make it easier to understand. The main trait shown in Figure 4-19 includes the following four traits: `positionExchange`, `positionReduce`, `positionSell`, and `positionTbill` They describe how exchange, capital distribution, sell, and Treasury bill maturity transactions are processed, respectively. As an example, the `positionExchange` trait is shown in Figure 4-20. Other transaction kinds are handled within the main trait. The four transaction kinds are kept in separate traits because their processing is more complicated than the others. All four traits include the `positionMatches` trait in Figure 4-21, which defines predicates for detecting transaction processing errors. The `positionMatches` trait includes the `positionBasics` trait in Figure 4-22, which defines the basic data structures that are used to model a position. The processing of some transaction kinds share similar processing patterns, updating the same fields each time. Hence, a few supporting operators are introduced to capture the common processing steps in the `positionBasics` trait.

In Figure 4-22, a `position` is a tuple consisting of six fields. The fields are the security, amount, income, last transaction date, open lots, and a tax documentation string of the position. The `postionBasics` trait includes the `income` trait in Figure 4-23, which defines a tuple named `income` consisting of different kinds of incomes derivable from financial security transactions. An `income` is a tuple with nine fields, each recording one type of income obtainable from security transactions. The fields can be conveniently classified into two categories. First, there are fields keeping cumulative incomes: the capital gain, dividends, and total interest of an income. Second, there are fields keeping current year incomes: the long-term capital gain, short-term capital gain, dividends, tax interest, municipal interest, and government interest of the income. The `income` trait also defines operators that codify how the different fields of an income are adjusted by different kinds of interest payments, dividend payments, and capital gains.

The main operators used by the `position` interface are the predicates for detecting input errors and the update operators. The predicates for checking if a transaction has encountered an error are defined separately from the operators that define the processing of the transaction. An alternative is to mix the two. We choose to separate them to make the specification simpler and more direct. Our specifications, of course, do not require an implementation to have separate checking and processing steps. The separation of checking and processing is one freedom specifiers can exploit at their convenience.

The workhorse for detecting input errors is `validMatch`, given in the `positionMatches` trait. It takes a `position` and a `trans`, and returns true if two conditions hold. First, the input `trans` must have only one lot. Second, the open lots of the position, which is a `trans_set`, must contain a `trans` whose security and lot match those of the input `trans`. While `validMatch` finds a match for a single lot, the operator `validMatches` finds matches for all the lots given in the input transaction. The operator ensures that there is a matching `trans` in the open lots of the position for each lot in the given transaction. In addition, the amounts sold must be covered by all the lots, and if there is a partial lot, it must be the last one. The third argument `validMatches` takes is a boolean value that indicates whether the match on the last lot must be a complete lot.

An example of an update operator is `update_buy` given in the `position` trait. It updates a position with a transaction as follows: it increments its amount field by the amount of the transaction, adds the new transaction to its open lots, and sets its the last transaction date as the date of the transaction. There is an update operator for each transaction kind.

4.9 The position Interface

The specification of the `position` interface is given in Figure 4-24. It exports two types: `income`, an exposed type, and `position`, a mutable abstract type. The `position` type is mutable so that positions can be updated in place. It also declares a constant, `maxTaxLen`, which is the maximum length of documentation strings the functions in the interface generate. The interface declares three `spec` variables. The spec variables, `cur_year` and `holding_period`, hold the two constants, the current year and the holding period, needed for updating positions. These constants are established at module initialization, by calling `position_initMod`.[4] Alternatives would be to store the constants in global variables accessible to every module in the program, or to pass them as input parameters to functions that need them. Using spec variables to specify them allows them to be encapsulated in the module that needs them without the penalty of passing them around in function calls.

---

[4] LCL conventions require that if a module has an initialization procedure, the initialization procedure must be called by its client before any other procedures of the module can be invoked.

```
position (String): trait
 includes positionExchange, positionReduce, positionSell, positionTbill
 introduces
  isInitialized: position → Bool
  create: String → position
  update_buy: position, trans → position
  update_dividends, update_interest, update_cap_dist, update_tbill_mat:
   position, trans, Int → position % need cur_year
  validMatchWithBuy: position, trans → Bool
  totalCost: position → double
   % formatting details, leave unspecified
  position2string, position2taxString, position2olotsString: position → String
 asserts ∀ p, p2: position, taxl, yr: Int, t: trans, s: String
  isInitialized(p) == ¬ (p.lastTransDate = null_date);
  create(s) == [ [s], 0, emptyIncome, null_date, {}, empty];
  validMatchWithBuy(p, t) ==
   if t.kind = sell then validMatches(p, t, false)
   else if t.kind = tbill_mat
        then validMatches(p, t, true) ∧ tbillInterestOk(p, t)
        else if t.kind = exchange
             then validMatch(p, t) ∧ findMatch(p, t).amt ≥ t.amt
             else t.kind = cap_dist ∧ validMatch(p, t);
  totalCost(p) == if p.lastTransDate = null_date ∨ p.amt = 0 then 0
                  else sum_net(p.openLots);
  t.kind = buy ⇒
   update_buy(p, t) =
    set_amtOlotsDate(p, p.amt + t.amt, insert(t, p.openLots), t.date);
  t.kind = cash_div ⇒
   update_dividends(p, t, yr) =
    set_lastTransDate(
     set_income(p, incDividends(p.income, t.net, year(t.date), yr)),
                      t.date);
  isInterestKind(t.kind) ⇒
   update_interest(p, t, yr) =
    set_lastTransDate(
     set_income(p, incInterestKind(p.income, t.net, t.kind, yr, year(t.date))),
                      t.date);
 implies converts create, validMatchWithBuy, totalCost, isInitialized
```

Figure 4-19: position.lsl

```
positionExchange: trait
 includes positionMatches
 introduces
  match_exchange: position, trans → tset
  update_exchange: position, trans → position
 asserts ∀ p: position, t: trans
  validMatch(p, t) ⇒
   match_exchange(p, t) =
    (if t.amt ≥ findMatch(p, t).amt then delete(t, p.openLots)
     else update_olots(p.openLots, t, findMatch(p, t).amt - t.amt));
  (t.kind = exchange ∧ validMatch(p, t)) ⇒
   update_exchange(p, t) =
    set_amtOlotsDate(p, p.amt - t.amt, match_exchange(p, t), t.date);
```

Figure 4-20: positionExchange.lsl

```
positionMatches: trait
 includes positionBasics
 introduces
  validMatch: position, trans → Bool
  validMatches: position, trans, Bool → Bool
  validAllMatches: tset, security, lot_list, double, Bool → Bool
  findMatch: position, trans → trans
 asserts ∀ amt: double, p: position, e: lot, y: lot_list, se: security,
          t: trans, ts: tset, completeLot: Bool
  validMatch(p, t) == matchKey(t.security, car(t.lots), p.openLots)
                      ∧ length(t.lots) = 1;
  validMatches(p, t, completeLot) == (t.kind = sell ∧ t.amt = 0)
   % above: selling zero shares is for special court-ordered settlements.
   ∨ (t.lots ≠ nil
       ∧ validAllMatches(p.openLots, t.security, t.lots, t.amt, completeLot));
  validAllMatches(ts, se, nil, amt, completeLot) ==
   if completeLot then amt = 0 else amt ≤ 0;
  validAllMatches(ts, se, cons(e, y), amt, completeLot) ==
   amt > 0 ∧ matchKey(se, e, ts)
   ∧ validAllMatches(ts, se, y, amt - findTrans(se, e, ts).amt, completeLot);
  validMatch(p, t) ⇒     % an abbreviation
   findMatch(p, t) = findTrans(t.security, car(t.lots), p.openLots);
 implies converts validMatch, validMatches, validAllMatches
```

Figure 4-21: positionMatches.lsl

```
positionBasics: trait
 includes trans_set, date, income
 position tuple of security: security, amt: double, income: income,
                   lastTransDate: date, openLots: tset, taxStr: String
 introduces
  set_amtOlotsDate: position, double, tset, date → position
  adjust_amt_and_net: trans, double → trans
  update_olots: tset, trans, double → tset
  __.capGain, __.dividends, __.totalInterest, __.ltCG_CY, __.stCG_CY,
   __.dividendsCY, __.taxInterestCY, __.muniInterestCY,
   __.govtInterestCY: position → double
 asserts ∀ amt: double, p: position, yr, tyr: Int, sd: date,
           t, mt: trans, ts: tset
  set_amtOlotsDate(p, amt, ts, sd) ==
   set_amt(set_openLots(set_lastTransDate(p, sd), ts), amt);
  adjust_amt_and_net(t, amt) =
   set_net(set_amt(t, t.amt - amt), t.net - ((t.net / t.amt) * amt));
  update_olots(ts, t, amt) =
   insert(adjust_amt_and_net(t, amt), delete(t, ts));
   % convenient abbreviations
  p.capGain == p.income.capGain;
  p.dividends == p.income.dividends;
  p.totalInterest == p.income.totalInterest;
  p.ltCG_CY == p.income.ltCG_CY;
  p.stCG_CY == p.income.stCG_CY;
  p.dividendsCY == p.income.dividendsCY;
  p.taxInterestCY == p.income.taxInterestCY;
  p.muniInterestCY == p.income.muniInterestCY;
  p.govtInterestCY == p.income.govtInterestCY;
 implies converts adjust_amt_and_net, set_amtOlotsDate
```

Figure 4-22: positionBasics.lsl

```
income (String, income): trait
 includes kind (String, kind), genlib (String, Int)
 income tuple of capGain, dividends, totalInterest, ltCG_CY, stCG_CY,
                 dividendsCY, taxInterestCY, muniInterestCY,
                 govtInterestCY: double
 introduces
  emptyIncome: → income
  sum_incomes: income, income → income
  incCYInterestKind: income, double, kind → income
  incInterestKind: income, double, kind, Int, Int → income
  incDividends: income, double, Int, Int → income
  incCapGain: income, double, double, Int, Int → income
   % formatting details, leave unspecified
  income2string, income2taxString: income → String
 asserts ∀ amt, lt, st: double, i, i2: income, yr, tyr: Int, k: kind
   emptyIncome == [0, 0, 0, 0, 0, 0, 0, 0, 0];
   sum_incomes(i, i2) ==
    [i.capGain + i2.capGain, i.dividends + i2.dividends,
     i.totalInterest + i2.totalInterest, i.ltCG_CY + i2.ltCG_CY,
     i.stCG_CY + i2.stCG_CY, i.dividendsCY + i2.dividendsCY,
     i.taxInterestCY + i2.taxInterestCY, i.muniInterestCY + i2.muniInterestCY,
     i.govtInterestCY + i2.govtInterestCY];
   incCYInterestKind(i, amt, interest) ==
    set_taxInterestCY(i, i.taxInterestCY + amt);
   incCYInterestKind(i, amt, muni_interest) ==
    set_muniInterestCY(i, i.muniInterestCY + amt);
   incCYInterestKind(i, amt, govt_interest) ==
    set_govtInterestCY(i, i.govtInterestCY + amt);
   isInterestKind(k) ⇒
    incInterestKind(i, amt, k, yr, tyr) =
     (if yr = tyr
      then set_totalInterest(incCYInterestKind(i, amt, k),
                             i.totalInterest + amt)
      else incCYInterestKind(i, amt, k));
   incDividends(i, amt, tyr, yr) ==
    set_dividends(if tyr = yr then set_dividendsCY(i, i.dividendsCY + amt)
                  else i, i.dividends + amt);
   incCapGain(i, lt, st, tyr, yr) ==
    set_capGain(if tyr = yr
                then set_ltCG_CY(set_stCG_CY(i, i.stCG_CY + st),
                                 i.ltCG_CY + lt)
                else i, st + lt);
 implies converts incDividends, incCapGain
```

Figure 4-23: income.lsl

These two spec variables are most easily implemented as C `static` variables in the `position` module.

The interface also declares a boolean `spec` variable named `seenError`. The `seenError` variable divides the abstract state into two: one in which at least one error has been detected by some exported function of the interface, and one in which no error has been detected. It is used to state invariants maintained by the interface in the absence of errors.

The `position` interface exports many functions, most of which are observers with simple specifications or printing routines. A few functions require brief mention. Calling `position_create` with a string returns a new `position` object with a security that has the string as its name. The `position_reset` function is similar, except that no new position is created; it reuses the position. The observer `position_is_uninitialized` indicates if a position has already been updated by a transaction.

The interface is designed with a specific processing pattern in mind: its client is expected to process transactions in groups. Each group consists of a series of transactions that involve the same security, and the series is ordered accordingly to the dates of the transactions, with the earliest transaction appearing before later ones. The `position_initialize` function is intended to initialize the start of a new group of transactions involving a new security. It also updates the position with its transaction argument. There are only two kinds of transactions that can initialize a position: a buy transaction and a new_security transaction. Other transaction kinds trigger an error message. This check is useful for catching errors in user's inputs. An initialized position can subsequently be updated by calling `position_update` with a transaction.

Since `position_update` carries out fairly complicated processing, its specification is large. It is, however, highly structured. It requires the implementor to check that the given position and transaction have the same security. The ensures clause is a nested conditional; there is a case for each kind of transaction PM is required to handle. For some transaction kinds, additional checks need to be performed. For example, if a buy transaction is given, the condition `validMatch(p^, t)` must be false, and the transaction must have a single lot. Hence, a position can only be updated by a buy transaction if the new buy lot does not match any of the open lots of the position. This check establishes the invariant that the lot of a `trans` in the open lots of a position uniquely determines the `trans`.[5]

The details of how each position should be updated are described in the various position traits. The specification of `position_update` takes a layered approach to describing the behavior of the function. It highlights the main checks the function must perform, makes clear that its processing is dependent on the transaction kind, and leaves out details of how the transaction is processed. If the reader is interested in the details, they can be obtained from the `position` traits. Highlighting the details in the interface level clutters the specification.

A different design of the `position` interface could have exported one function for each transaction kind. Such a design has the advantage of breaking a big specification into smaller pieces. The design, however, simply pushes the case analysis of transaction kinds to clients. For the convenience of the client, the latter approach is not used.

---

[5] The claims clause in the `position_update` function states some properties that must logically follow from the specification of the function. Claims are discussed in the next chapter.

```
imports trans_set;
typedef struct {double capGain, dividends, totalInterest, ltCG_CY, stCG_CY,
                       dividendsCY, taxInterestCY, muniInterestCY,
                       govtInterestCY;} income;
mutable type position;
constant nat maxTaxLen;
spec nat cur_year, holding_period;
spec bool seenError;
uses position (cstring, income for income);

bool position_initMod (nat year, nat hp) nat cur_year, holding_period;
                                         bool seenError; {
  modifies cur_year, holding_period;
  ensures result ∧ ¬ seenError′ ∧ cur_year′ = year ∧ holding_period′ = hp;
}
position position_create (cstring name) {
  ensures fresh(result) ∧ result′ = create(getString(name^));
}
void position_reset (position p, cstring name) {
  modifies p;
  ensures p′ = create(getString(name^));
}
void position_free (position p) {
  modifies p;
  ensures trashed(p);
}
bool position_is_uninitialized (position p) {
  ensures result = ¬ isInitialized(p^);
}
void position_initialize (position p, trans t) FILE *stderr; bool seenError; {
  let fileObj be *stderr^;
  modifies p, seenError, fileObj;
  ensures p′ = (if t.kind = buy
                then update_buy(create(t.security.sym), t)
                else create(t.security.sym))
       ∧ if t.kind = buy ∨ t.kind = new_security
          then unchanged(fileObj, seenError)
          else seenError′
               ∧ ∃ errm: cstring (appendedMsg(fileObj′, fileObj^, errm));
}
security position_security (position p) {
  ensures result = p^.security;
}
double position_amt (position p) {
  ensures result = p^.amt;
}
trans_set position_open_lots (position p) {
  ensures fresh(result) ∧ result′ = [p^.openLots, 0];
}
```

Figure 4-24: position.lcl, part 1

```
void position_update (position p, trans t) nat cur_year, holding_period;
                                           bool seenError; FILE *stderr; {
  let fileObj be *stderr^,
      report be seenError′
                ∧ ∃ errm: cstring (appendedMsg(fileObj′, fileObj^, errm)),
      ok be unchanged(seenError, fileObj);
  checks p^.security = t.security;
  modifies p, seenError, fileObj;
  ensures
   if p^.lastTransDate > t.date
   then report
   else if t.kind = buy ∧ ¬ validMatch(p^, t) ∧ length(t.lots) = 1
        then p′ = update_buy(p^, t) ∧ ok
        else if t.kind = cash_div
             then p′ = update_dividends(p^, t, cur_year^) ∧ ok
        else if isInterestKind(t.kind)
             then p′ = update_interest(p^, t, cur_year^) ∧ ok
        else if validMatchWithBuy(p^, t)
             then if t.kind = cap_dist
                  then p′ = update_cap_dist(p^, t, cur_year^) ∧ ok
                  else if t.kind = tbill_mat
                       then p′ = update_tbill_mat(p^, t, cur_year^) ∧ ok
                  else if t.kind = exchange
                       then p′ = update_exchange(p^, t) ∧ ok
                  else if t.kind = sell
                       then p′ = update_sell(p^, t, cur_year^, holding_period^,
                                             maxTaxLen) ∧ ok
                       else report
             else report;
  claims ¬ (seenError′) ⇒
          ((t.kind = cap_dist ⇒ (p′.dividends ≥ p^.dividends
                                  ∧ p′.totalInterest = p^.totalInterest
                                  ∧ p′.capGain = p^.capGain))
           ∧ (t.kind = sell ⇒
               ((p′.ltCG_CY - p^.ltCG_CY) + (p′.stCG_CY - p^.stCG_CY))
                = (p′.capGain - p^.capGain)));
}
void position_write (position p, FILE *pos_file) {
  modifies *pos_file;
  ensures (*pos_file)′ = (*pos_file)^ || position2string(p^);
}
void position_write_tax (position p, FILE *pos_file) {
  modifies *pos_file;
  ensures (*pos_file)′ = (*pos_file)^ || position2taxString(p^);
}
double position_write_olots (position p, FILE *olot_file) {
  modifies *olot_file;
  ensures (*olot_file)′ = (*olot_file)^ || position2olotsString(p^)
          ∧ result = totalCost(p^);
}
```

Figure 4-24: position.lcl, part 2.

```
income position_income (position p) {
  ensures result = p^.income;
}
income income_create (void) {
  ensures result = emptyIncome;
}
void income_sum (income *i1, income i2) {
  modifies *i1;
  ensures (*i1)' = sum_incomes((*i1)^, i2);
}
void income_write (income i, FILE *pos_file) {
  modifies *pos_file;
  ensures (*pos_file)' = (*pos_file)^ || income2string(i);
}
void income_write_tax (income i, FILE *pos_file) {
  modifies *pos_file;
  ensures (*pos_file)' = (*pos_file)^ || income2taxString(i);
}
```

Figure 4-24: position.lcl, part 3.

4.10 Summary

We documented some interfaces that were used to build a useful program. We used the interface specifications to illustrate some specification techniques used to document the interfaces. Our techniques are complementary to those discussed in [39].

By reusing specifications from [15] in the specification of `trans_set`, we illustrated reusing existing LCL specifications. Our specification of the `dateBasics` trait, however, is not as reusable as we would have liked. The trait included special indeterminate dates that are needed in specifying our program. It suggests that specification reuse is often rather difficult. Existing specifications may be useful as conceptual models for future specification needs, but some customizations of existing specifications are likely to be necessary before they could be reused.

We also illustrated ways to achieve more compact and easier to understand specifications through the use of checks clauses, constraints associated with exposed types, and a flat style of defining LSL operators. We pointed out some common errors in writing specifications. In particular, the distinction between the theories in the two tiers in Larch specifications needs to be kept in mind to avoid a common class of specification errors. Many of the techniques we described are general; they are specific neither to LCL nor Larch.

Chapter 5

Using Redundancy in Specifications

Most uses of a formal specification assume that the specification is consistent and *appropriate*, in the sense that it states what the specifier has in mind. However, both are often false, especially when large specifications are involved. This chapter describes a technique for testing specifications using redundant information in specifications, called *claims*.

Besides using claims to help test and validate formal specifications, claims can be used in other ways. The author of a specification can use claims to highlight important or interesting properties of the specification. Claims can serve as useful lemmas in program verification. They can also be used to suggest useful test cases for implementations. Our research can also be viewed as exploring the use of formal specification to detect design errors [13, 23, 32, 10].

We study claims in the context of LCL specifications. Our focus is on checking Larch interface specifications; complementary work has been done on checking Larch Shared Language specifications [8].

In the next section, we describe our approach to testing specifications. In Section 5.2, we introduce three kinds of claims expressible in LCL and their semantics. In subsequent sections, we describe other ways claims can be useful. We draw upon the specifications described in Chapter 4 for examples. In Section 5.9, we describe some practical experience we have had with verifying LCL claims. In Section 5.10, we explain why we prefer to derive a desired property of a specification as a claim rather than specify it as an axiom. In the last section, we summarize the chapter.

5.1 Specification Testing Approach

Like programs, specifications can contain errors. We consider two related kinds of specification problems. First, the specification *aptness* problem: given a formal specification, does the specification say what is intended? Second, the specification *regression testing* problem: when a specification is changed, what can be done to minimize inadvertent consequences?

Executable specification languages are designed to address these problems by allowing specifiers to run specifications [42]. Larch specification languages are designed to be simple and expressive, rather than executable. In place of executing specifications as a means of testing them, logical conjectures about the specification can be stated and checked at specification time. A Larch specification defines a logical theory. The conjectures about Larch specifications are called *claims*. Claims contain redundant information that does not add to the logical content of the specification.

Our general approach of tackling the specification aptness problem is: given a formal specification, a specifier attempts to prove some conjectures that the specifier believes should follow from the specification. Success in the proof attempt provides the specifier with more confidence that the specification is appropriate. Failures can lead to a better understanding of the specification and can identify sources of errors.

Our methodology addresses the specification regression testing problem as follows: we attempt to re-prove the conjectures that were true before the specification changed. Success in the proof attempt reassures us that the properties expressed by the verified claims are still valid in the new specification. Failure can help uncover unwanted consequences.

While this idea is not new [13], our work provides specifiers specific guidance on how to find conjectures that are useful for testing specifications than earlier work. We also strengthen this methodology by adding facilities in a specification language so that a specifier can make claims about specifications. A tool can be built to translate such claims, together with the specifications, into inputs suitable for a proof checker. The specifier can then verify the claims using the proof checker.

Research in the past looked at generic properties of formal specifications. Two interesting and useful properties are consistency and completeness. Checking these properties is, however, impossible in general and very difficult in practice. While we know what it means for a logical specification to be consistent, what constitutes a complete specification is not clear [41]. While checking for the consistency and completeness of formal specifications is possible, it does not address the original problem we have: how do we know that the specification describes our intent?

We take a different but complementary approach: we focus on problem-specific claims which are frequently easier to state and prove.

5.2 Semantics of Claims

A claim in an LCL specification defines a conjecture about the logical theory of the specification. There are three kinds of LCL claims.

First, a *procedure* claim expresses a conjecture that is associated with an individual function of a module. For this purpose, LCL supports a `claims` clause with a function specification, which has the same syntax as the ensures clause. An example is shown in Figure 5-1.

The semantics of a procedure claim is given by the following schema:

$$(\texttt{RequiresP} \wedge \texttt{ChecksP} \wedge \texttt{ModifiesP} \wedge \texttt{EnsuresP}) \Rightarrow \texttt{ClaimsP}$$

where `RequiresP` stands for the requires clause of the function, `ChecksP`, the checks clause, `ModifiesP`, the translation of the modifies clause, `EnsuresP`, the ensures clause, and `ClaimsP`, the claims clause.

```
nat date_year (date d) {
  checks isNormalDate(d);
  ensures result = year(d);
  claims result ≤ 99;
}
```

Figure 5-1: An example of a procedure claim.

Sometimes, there may be a number of procedure claims associated with a single function. To avoid cluttering the specification of a function, a procedure claim may be given in a different syntactic form. An example is given in Figure 5-2. The semantics of the `date_yearRange` claim is identical to the procedure claim shown in Figure 5-1. The identifier `result` refers to the result returned by the `date_year` function call in the body of the claim.

```
claims date_yearRange (date d) {
  body { date_year(d); }
  ensures result ≤ 99;
}
```

Figure 5-2: Alternate syntax of a procedure claim.

The second kind of claims is a *module claim*. A module claim of an interface is a conjecture that an invariant holds. An invariant is a property that is maintained by the functions of the module. Module claims can be used to make claims about invariants about abstract types, or about properties that must hold for the private state of the module. An example of a module claim is the `lotsInvariant` claim shown in Figure 5-3. The claim is from the `lot_list` module (in Appendix D.30) which defines `lot_list` to be a mutable type.

```
claims lotsInvariant (lot_list x) {
  ensures ∀ e: lot (count(e, x~) ≤ 1);
}
```

Figure 5-3: An example of a module claim.

The `lotsInvariant` claim is equivalent to the interface invariant on `lot_list` objects:

```
∀ x: lot_list_Obj, e: lot (count(e, x~) ≤ 1)
```

The symbol ~ is a state operator; it is analogous to the pre state operator ^ and the post state operator ′, but it stands for any state. The module claim in Figure 5-3 says that the lots in a `lot_list` form a set.

Data type induction principles can be used to show that a module claim holds for an interface. One data type principle is described in detail in Section 7.5. The gist of the principle is as follows: First, show that all constructors in the interface produce instances of the type that satisfy the invariant. Second, we show that all mutators of the type in the interface preserve the invariant. Since observers do not modify the value of instances of the type (though they may modify the representation), they cannot affect the invariant, and hence they are not needed in the inductive proof.

A module claim can be translated into several procedure claims about the functions of the module. Figure 5-4 shows the translation of the `lotsInvariant` module claim into procedure claims involving the constructor and mutators of the `lot_list` module. We support such shorthands because module claims highlight properties that are preserved by the functions of the module. They are also more modular and robust than separate procedure claims: adding a new constructor or mutator to the module does not require adding the corresponding procedure claim.

```
lot_list lot_list_create (void) {
  ensures fresh(result) ∧ result′ = nil;
  claims ∀ e: lot (count(e, result′) ≤ 1);
}
bool lot_list_add (lot_list s, lot x) {
  requires ∀ e: lot (count(e, s^) ≤ 1);
  modifies s;
  ensures result = x ∈ s^ ∧ if result then unchanged(s) else s′ = cons(x, s^);
  claims ∀ e: lot (count(e, s′) ≤ 1);
}
bool lot_list_remove (lot_list s, lot x) {
  requires ∀ e: lot (count(e, s^) ≤ 1);
  modifies s;
  ensures (result = x ∈ s^) ∧ ¬(x ∈ s′)
         ∧ ∀ y: lot ((y ∈ s^ ∧ y ≠ x) ⇒ y ∈ s′);
  claims ∀ e: lot (count(e, s′) ≤ 1);
}
```

Figure 5-4: Translating a module claim into procedure claims.

Module claims that are properties about the private state of a module can be similarly proved by module induction. The basis case of the induction involves the initialization function of the module, and the inductive cases involve the functions that modify the private state.

The third kind of claim is an *output type consistency* claim, or an *output* claim for short. An output claim is a procedure claim about an output result, with abstract type T, of a function F in a module that imports the module defining T. An output claim states that an output result of a function satisfies the invariant associated with the type of the output result.[1] Output claims are necessary consistency conditions associated with the types of output results of a function. Figure 5-5 shows an example of an output claim about the `trans_lots` function of the `trans` module.

---

[1] In Chapter 2, we define the output results of a function to include its returned value (if any), and any object, be it an input, a spec or a global object, listed in the modifies clause of the function.

```
claims trans_lots lotsInvariant;
```

Figure 5-5: An example of an output type consistency claim.

The claim says that the output of `trans_lots`, a `lot_list` value in the post state, must satisfy the `lotsInvariant` module invariant. If we expand the claim into a procedure claim, it would look like:

```
lot_list trans_lots (trans t) {
  ensures result′ = t.lots ∧ fresh(result);
  claims ∀ e: lot (count(e, result′) ≤ 1);
}
```

A key difference between the output claim and the above procedure claim is that the former relies on the implicit definition of the `lotsInvariant` module claim. If the module claim changes, the meaning of the output claim changes in tandem[2].

In the next few sections, we show the different ways LCL claims can be used.

5.3 Claims Help Test Specifications

To study how claims can be used to help test interface specifications, we carried out a small verification exercise. We manually translated the position interface specifications and claims into inputs suitable for LP, a proof checker [7]. We formally verified parts of the `amountConsistency` and `noShortPositions` claim in Figure 5-6 using LP, and in the process, discovered several errors in the specification. In this section, we characterize the kinds of errors we found and describe a few of the errors to illustrate how claims verification helped us uncover errors in specifications. The specifications in Appendix D give other useful claims.

```
claims amountConsistency (position p) bool seenError; {
  ensures ¬ (seenError~) ⇒ p~.amt = sum_amt(p~.openLots);
}
claims noShortPositions (position p) bool seenError; {
  ensures ¬ (seenError~) ⇒ p~.amt ≥ 0;
}
```

Figure 5-6: The amountConsistency and noShortPositions module claims.

The `amountConsistency` claim shown in Figure 5-6 states that the amount of a position is the total of the amounts of the transactions in its open lots. The proof of the claim requires an induction. The basis step consists of showing that the position object created by the `position_create` function satisfies the invariant. The inductive steps consist of showing that the following mutators of position objects preserve the invariant: `position_reset`, `position_initialize`, and `position_update`. The `noShortPositions` claim states that a position must not have negative amounts.

---

[2] We assume that the module claim specified in an output claim expresses an invariant of the type of the output of the function.

The errors we encountered in our specification fall into two broad kinds. First, there are errors in stating the claims themselves. This kind of error often results from an inadequate understanding of the specification. Some conjectures we stated as claims were false. When they were discovered, they led us to reformulate the claims, and sometimes to change our specification. Second, there are errors in the specification itself. There are three classes in this kind of errors. The first class are generic modeling errors in which inappropriate common data types are chosen to model application domain objects. The second class are omission errors where we did not specify some necessary conditions that were intended. The third class are commission errors where we simply stated the wrong axioms.

The various kinds of errors we uncovered can also occur in informal program specifications and other specification languages. They occur as a result of human errors, especially in the presence of specification evolution.

Next, we describe a number of errors we encountered in our proofs of claims to illustrate the different kinds of errors, and the contexts in which the errors arose.

5.3.1 Examples of Specification Errors

One of the first errors we uncovered lies in the statement of the claim itself. In the faulty specification, we did not introduce the spec variable `seenError`, and our initial claim was

```
claims amountConsistencyOld (position p) bool seenError; {
  ensures p~.amt = sum_amt(p~.openLots);
}
```

The proof failed in the cases where an error occurred and the value of the position object in the post state was not guaranteed by the specification. To correct the error, we introduced the spec variable `seenError` to restrict our claim to non-erroneous states. An alternative is to strengthen the specifications so that the position object is unchanged if an error occurs. The latter solution is likely to force the implementor to check for input errors before modifying the position object. Since the user of the PM program does not rely on the results of the program if an error occurs, this approach is considered inefficient. The error in the `amountConsistency` claim helps highlight the program requirement that we care about the invariant only if no input errors occur. Errors in the claim statement itself often help us better understand the impact of our specifications.

A modeling issue in our specification is highlighted by the failure to prove the amount consistency claim. We initially used the handbook trait `FloatingPoint` to model the C `double` type. The proof of the claim requires the commutative and associative laws of floating point numbers. The addition and multiplication of floating point numbers, however, may not be associative. The lack of such numeric properties make formal reasoning difficult. A careful modeling of `double` using floating point arithmetic is appropriate if we are interested in the exact precision it offers us. Our intent is, however, more modest. The precision requirements of our program are sufficiently met by the double precision of most computers today. Hence, the `Rational` handbook trait is used to model the `double` type instead.

```
void position_update (position p, trans t) nat cur_year, holding_period;
                                           bool seenError; FILE *stderr; {
  let fileObj be *stderr^,
      printErrorMsg be ∃ errm: cstring (appendedMsg(fileObj′, fileObj^, errm)),
      report be seenError′ ∧ printErrorMsg,
      ok be unchanged(seenError, fileObj);
  checks p^.security = t.security;
  modifies p, seenError, fileObj;
  ensures
   if p^.lastTransDate > t.date
   then report
   else if t.kind = buy ∧ ¬validMatch(p^, t)
        then p′ = update_buy(p^, t) ∧ ok
        else if t.kind = cash_div
             then p′ = update_dividends(p^, t, cur_year^) ∧ ok
        else if isInterestKind(t.kind)
             then p′ = update_interest(p^, t, cur_year^) ∧ ok
        else if validMatchWithBuy(p^, t)
        then if t.kind = cap_dist
             then p′ = update_cap_dist(p^, t, cur_year^) ∧ ok
        else if t.kind = tbill_mat
             then p′ = update_tbill_mat(p^, t, cur_year^) ∧ ok
             else if t.kind = exchange
                  then p′ = update_exchange(p^, t) ∧ ok
             else if t.kind = sell
                  then p′ = update_sell(p^, t, cur_year^, holding_period^,
                                        maxTaxLen) ∧ ok
                  else report
             else report;
}
```

Figure 5-7: A faulty version of position_update specification.

An example of an omission error appears in the specification of `position_update` in Figure 5-7. In the ensures clause, a buy transaction must have a lot that does not match the open lots of the position. In our formal proof, we discovered that an additional check is necessary: the transaction must have a single lot. The correction consists of adding the check `length(t.lots) = 1` to the buy case of `position_update` as sketched below:

```
  ...
   if p^.lastTransDate ≤ t.date
   then if t.kind = buy ∧ ¬validMatch(p^, t) ∧ length(t.lots) = 1
        then p′ = update_buy(p^, t) ∧ ok
        ...
```

The discovery of the length check omission prompted us to re-examine the specification of the `trans_set` interface where a similar matching check was made in the `trans_set_insert` function. We found a similar omission there, and corrected it. This example illustrates how the discovery of one error can have the desirable cascading effect of helping us find other similar errors.

A source of common specification errors occurs in copying and editing specifications that are similar. We often start the specification of a transaction by copying the specification of a similar kind of transaction, and editing the copy. Sometimes, the similarities are misconceived. We encountered a case in point in the specification of the exchange transaction. A valid exchange transaction must pass a check: its amount must not exceed the matching transaction in the open lots of the position. This check was originally omitted in our specification because we originally specified the exchange transaction by copying the specification of the sell transaction and editing it. A sell transaction handles multiple lots and hence the amount of the transaction can exceed a single matching transaction in the open lots of the position. Unlike a sell transaction, an exchange transaction handles only a single lot, and hence the check on the amount of the transaction is needed.

We discovered a subtle logical error during the proof process. It illustrates how subtle logical errors can sometimes escape human inspection of specifications but can be detected in a machine-checked proof. The exact details of the context in which the specification error occurred are complicated to describe because it occurred in a previous version of the `position` trait which is quite different from the current version. The specific details are, however, unimportant. The gist of the specification error was the mis-statement of an axiom. We stated an axiom of the form:

```
∀ p:position, t:trans (Q(p, t) ⇔ R(p, t, 0))
```

when what was intended was:

```
∀ p:position, t:trans (Q(p, t) ⇔ ∀ hp:int (R(p, t, hp))
```

The mistake was discovered during the proof when an instantiation for the `hp` variable was needed, but the given definition of `Q` could only work with `hp` equals to 0.

The various kinds of errors we encountered were uncovered when we verified claims. It was often not difficult to figure out where and what the error was. Their discoveries suggest that claims verification can help uncover common classes of specification errors.

5.4 Claims Help Specification Regression Testing

With changes in program requirements and improved understanding of the problem domain, specifications change over time. Some changes have little impact on the central properties of a specification, some may introduce unintentional changes, and others introduce intended changes. The exact impact of a specification change is, however, often difficult to appreciate because different aspects of a specification may be intricately related. The meaning of a specification can change dramatically and subtly if the specification is modified slightly. It is desirable for specifiers to have a better appreciation of the impact of specification changes. Our approach for testing specifications extends naturally into regression testing of specifications, as described in Section 5.1.

Errors get introduced as specifications evolve. Many careless errors occur because local changes are not propagated throughout the specification. We encountered a simple example in the specification of `position_update`. The specification of the `validMatchWithBuy` operator used in an old version of the specification of `position_update` is given in Figure 5-8. The relevant part of the old specification of `position_update` is given in Figure 5-9. When the specification of `position_update` was modified so that the `validMatchWithBuy` operator was used to guard capital distribution transactions in addition to sell, exchange and T bill maturity transactions as given in Figure 5-10, the specification of `validMatchWithBuy` in Figure 5-8 was, unfortunately, not updated. The mistake, however, was discovered easily during the proof of the `noShortPositions` claim (shown earlier in Figure 5-6). Such careless mistakes are common as specifications evolve. The corrected definition of `validMatchWithBuy` operator is given in Figure 5-11.

```
validMatchWithBuy(p, t) ==
 if t.kind = sell then validMatches(p, t, false)
 else if t.kind = tbill_mat
      then validMatches(p, t, true) ∧ tbillInterestOk(p, t)
      else t.kind = exchange ∧ validMatch(p, t) ∧ (findMatch(p, t).amt ≥ t.amt)
```

Figure 5-8: An old definition of validMatchWithBuy trait operator.

```
void position_update (position p, trans t) nat cur_year, holding_period;
 ...
  ensures
   if p^.lastTransDate > t.date
   then report
   else if t.kind = buy ∧ ¬ validMatch(p^, t)
        then p′ = update_buy(p^, t) ∧ ok
        else if t.kind = cash_div
             then p′ = update_dividends(p^, t, cur_year^) ∧ ok
        else if isInterestKind(t.kind)
             then p′ = update_interest(p^, t, cur_year^) ∧ ok
        else if t.kind = cap_dist
             then p′ = update_cap_dist(p^, t, cur_year^) ∧ ok
        else if validMatchWithBuy(p^, t)
             then if t.kind = tbill_mat
                  then p′ = update_tbill_mat(p^, t, cur_year^) ∧ ok
             else if t.kind = exchange
                  then p′ = update_exchange(p^, t) ∧ ok
             else if t.kind = sell
                  then p′ = update_sell(p^, t, cur_year^, holding_period^,
                                        maxTaxLen) ∧ ok
                  else report
             else report;
}
```

Figure 5-9: An old version of position_update specification.

```
void position_update (position p, trans t) nat cur_year, holding_period;
 ...
  ensures
   if p^.lastTransDate > t.date
   then report
   else if t.kind = buy ∧ ¬ validMatch(p^, t) ∧ length(t.lots) = 1
        then p′ = update_buy(p^, t) ∧ ok
        else if t.kind = cash_div
             then p′ = update_dividends(p^, t, cur_year^) ∧ ok
        else if isInterestKind(t.kind)
             then p′ = update_interest(p^, t, cur_year^) ∧ ok
        else if validMatchWithBuy(p^, t)
             then if t.kind = cap_dist
                  then p′ = update_cap_dist(p^, t, cur_year^) ∧ ok
             else if t.kind = tbill_mat
                  then p′ = update_tbill_mat(p^, t, cur_year^) ∧ ok
             else if t.kind = exchange
                  then p′ = update_exchange(p^, t) ∧ ok
             else if t.kind = sell
                  then p′ = update_sell(p^, t, cur_year^, holding_period^,
                                        maxTaxLen) ∧ ok
                  else report
             else report;
}
```

Figure 5-10: The corrected version of position_update specification.

We believe that output type consistency claims are useful for detecting ramifications of specification changes across modules. For example, the output claim about `trans_lots` in Figure 5-5 relies on the `lotsInvariant` module claim in Figure 5-3. The module claim is, in turn, dependent on the specifications of the functions exported by the `lot_list` module since its proof requires a data type induction on the `lot_list` type. If any specification in the `lot_list` module changes in a way as to strengthen the `lot_list` invariant, and if the change is reflected in the definition of the `lotsInvariant` module claim, the specification of `trans_lots` may not be strong enough to guarantee the new invariant. For example, if

```
validMatchWithBuy(p, t) ==
 if t.kind = sell then validMatches(p, t, false)
 else if t.kind = tbill_mat
      then validMatches(p, t, true) ∧ tbillInterestOk(p, t)
      else if t.kind = exchange
           then validMatch(p, t) ∧ (findMatch(p, t).amt ≥ t.amt)
           else t.kind = cap_dist ∧ validMatch(p, t);
```

Figure 5-11: The corrected definition of validMatchWithBuy trait operator.

the `lotsInvariant` module claim is strengthened to require that `lot_list`'s be sorted[3], as in Figure 5-12, the proof of the output claim will fail, and alert the specifier to the inconsistency. It signals to the specifier that some changes in the `trans` module may need to accompany the changes in the `lot_list` module.

```
claims lotsInvariant (lot_list x) {
  ensures sorted(x~) ∧ ∀ e: lot (count(e, x~) = 1);
}
```

Figure 5-12: A new invariant on the lot_list module.

5.5 Claims Highlight Specification Properties

Claims are a useful design documentation tool. Claims can highlight important, interesting, or unusual properties of specifications. There are infinitely many consequences in a logical theory. Most of them are neither interesting nor useful. It can be difficult for readers of a specification to pick up the important or useful properties of the specification. Specifiers can use claims to highlight these properties. Readers of a specification can use them to check their understanding of the specification.

There are many detailed requirements about the behavior of a program. Some are more important than others. For example, in the design of the PM program, a key requirement is to check for errors in the input of the user. This is embodied in two central program constraints. First, the program should not allow the user to sell short, that is, to sell securities that the user does not own. Second, the cost basis of a security should not go below zero. The second constraint is useful because the amount of a capital distribution may exceed the cost basis of the security it is reducing. The excess should be recorded as dividends for tax purposes. These properties are expressed as the `noShortPositions` and `okCostBasis` claims in the `position` module, shown in Figure 5-13. The `noShortPositions` claim is the same as that given in Figure 5-6; it is reproduced here for convenience.

```
claims noShortPositions (position p) bool seenError; {
  ensures ¬ (seenError~) ⇒ p~.amt ≥ 0;
}
claims okCostBasis (position p) bool seenError; {
  ensures ¬ (seenError~) ⇒ (∀ t: trans (t ∈ p~.openLots ⇒ t.price ≥ 0));
}
```

Figure 5-13: Claims express key program constraints.

Claims can also be used to highlight unusual properties in the design of a module. For example, the `date` module adopts a special interpretation of a two-digit representation of a year: it interprets any number over fifty as the corresponding year in the current century, and any positive number under fifty as the corresponding year in the next century. This unusual interpretation is expressed in the `assumeCentury` module claim shown in Figure 5-14. The claim says that for normal dates, the date `"0/0/50"` represents the smallest date, and the date `"12/31/49"` represents the largest date.

---

[3] This requires the specifications of `lot_list_insert` and `lot_list_delete` be strengthened to maintain the new invariant.

```
claims assumeCentury (date d) {
  ensures isNormalDate(d) ⇒ ((string2date("0/0/50") ≤ d)
                              ∧ (d ≤ string2date("12/31/49")));
}
```

Figure 5-14: The assumeCentury module claim in the date interface.

The boundary conditions of a specification are an important class of specification properties. They are useful for highlighting the limits of the intent of a specification. For example, for a property about an indexing structure, such as an array, we consider if there is an off-by-one error in indexing. For collection types, we ask if a delete operation removes one element or all matching elements. For functions that produce output, we make claims about the specific length of the output produced to catch mistakes of not counting newlines and spaces needed for formatting.

Claims can highlight the essence of a specification without giving too much detail. For example, a capital distribution transaction updates a position in a complicated way. The salient feature of the transaction, however, is simple: *a distribution reduces the cost basis unless the cost basis is already zero.* The feature is expressed as the `distributionEffect` claim in Figure 5-15.

```
claims distributionEffect (position p, trans t) bool seenError; {
  requires p^.security = t.security ∧ t.kind = cap_dist;
  body { position_update(p, t);}
  ensures ((findMatch(p^, t).net ≠ 0) ∧ ¬(seenError')) ⇒
           (findMatch(p', t).net < findMatch(p^, t).net);
}
```

Figure 5-15: The claim about distribution effects in the position interface.

Claims can be used to offer different views of the implications of a specification. For example, the specification of the `position` module describes how a position is changed by the different transaction kinds. The `openLotsUnchanged` claim in Figure 5-16 tells of the situations under which the open lots of a position do not change. Instead of focusing on how a position is changed by a transaction kind, the claim is about how a specific property of the position is left unchanged by different transaction kinds. By giving the reader of the specification different views of the same specification, the intent and implications of the specification can be reinforced.

```
claims openLotsUnchanged (position p, trans t) bool seenError; {
  requires t.kind = cash_div ∨ isInterestKind(t.kind) ∨ t.kind = new_security;
  body { position_update(p, t); }
  ensures ¬ (seenError′) ⇒ p′.openLots = p^.openLots;
}
```

Figure 5-16: The openLotsUnchanged claim in the position module.

5.6 Claims Promote Module Coherence

A well-designed module is not an arbitrary collection of functions needed to support some clients. There are often invariants that are maintained by the functions in the module. Such invariants can be stated as claims, and proved to hold from the interfaces of the exported functions. Organizing a module around some useful or interesting claims promotes the design coherence of the module. It makes the designer focus more on overall module properties, less on specific operations to meet particular client needs.

An important constraint the PM program enforces is expressed as an invariant in the `trans` interface. It appears as the `buyConsistency` claim in Figure 5-17. It ensures that the net, amount, price of a buy transaction are non-negative, and that the amount must also be non-zero. It also requires that there is a single lot in a buy transaction, and that the product of its price and its amount is sufficiently close to its net.

```
claims buyConsistency (trans t) {
  requires t.kind = buy;
  ensures t.net ≥ 0 ∧ t.amt > 0 ∧ t.price ≥ 0 ∧ length(t.lots) = 1
          ∧ within1(t.amt * t.price, t.net);
}
```

Figure 5-17: The buyConsistency module claim in trans interface.

The transaction module of the original PM program provided a `trans_set_net` function that sets the net field of a transaction to a given value and a `trans_set_amt` function that changes the amount field of a transaction.[4] Since the net and the amount fields of a transaction can be set to arbitrary values using these functions, the invariant expressed by `buyConsistency` could not be maintained. It turned out that the operations supported by the original transaction module were too general: all actual uses of these two functions in the original PM program allowed the desired invariant to be maintained. In fact, one of them maintained it explicitly, with calls to adjust the net and the amount fields of a transaction in tandem.

The new PM program uses a more coherent design, which replaces the two functions in the module by a `trans_adjust_net` function that adjusts the net of the transaction together with its price, and a `trans_adjust_net_and_amt` function that adjusts the net and amount of a transaction together, both maintaining the intended invariants.

---

[4] As indicated in Chapter 4, our specifications were developed in the process of reengineering an existing program.

Claims also helped us improve the design coherence of the `position` module. In the module, we specified the constraints that must hold between the different fields of a position as claims, such as the `amountConsistency` claim shown earlier in Figure 5-6. The claim led us to learn that the original program was using a position in two different ways. First, a position was used to record the incomes due to the transactions of a single financial security. In this use, a position had book-keeping information such as the total number of shares of the security currently owned by the PM program user. Second, a position was used to accumulate the sums of the different kinds of incomes of all securities owned by the user. The original `position` module supports this second use by exporting a function named `position_sum` which takes two positions and added the different kinds of income from the second position into the respective kinds of income in the first position. In this second use, the book-keeping properties of a position were irrelevant. These properties include the name of the security, the last transaction date, the number of outstanding shares, and the open lots of a position.

To ensure that the `position_sum` function maintained the `amountConsistency` claim would force us to over-specify the behavior of `position_sum`. For example, we would require that in the post state, `position_sum` ensures that the amount and the open lots of the first position remained unchanged. Another arbitrary choice is to sum up the amounts and to union the open lots of the two positions. Either choice is arbitrary because the clients of `position_sum` do not rely on these properties of positions. Furthermore, we intended the claim to apply only to positions that represent individual securities. It was clear then that a better design is to separate the two uses of the `position` module. The new program codified the `position` module for the first use, and added a separate `income` abstraction to capture the second use.

5.7 Claims Support Program Reasoning

If a claim about a specification has been verified, it states a property that must be true of any valid implementation of the specification, since the specification is an abstraction of all its valid implementations. As such, claims can sometimes serve as useful lemmas in program verification. In particular, claims about a module can help the implementor of the module exploit special properties of the design.

```
claims trans_setUID (trans_set s) {
  ensures ∀ t1: trans, t2: trans
           ((t1 ∈ s~.val ∧ t2 ∈ s~.val ∧ t1.security = t2.security
             ∧ t1.lots = t2.lots) ⇒ t1 = t2);
}
```

Figure 5-18: The trans_setUID module claim.

For example, the specification of `trans_set_delete_match` in the `trans_set` module requires that all matching transaction be removed from the input set. An implementation strategy can rely on the `trans_setUID` module invariant maintained by the interface: there can only be one matching transaction in a transaction set. The claim is shown in Figure 5-18. This means that we can stop searching for more matching transactions as soon as the first one is found. The program optimization strategy is applicable to different implementations, but each of them will have to rely on a lemma that is derived from the specification of `trans_set` interface: the `trans_setUID` module claim. If the module claim has already been proved, the verifier of the delete function can simply use it as a lemma. The example also indicates that claims can help suggest optimizations to the implementor of an interface.

Claims that would be useful for formal program verification can be used for informal program reasoning. Our description of an optimizing implementation of the set delete function in the previous example is informal and uses the `trans_setUID` claim. Another example is found in the implementation of the `position_write` function, which writes the fields of a position to an output file. In the original PM program, the function printed the position only after it checked that the position had a strictly positive amount. Otherwise, it printed an error message indicating that the position was short. In the new implementation, the check is redundant because the `amountConsistency` claim in the `position` module guarantees the amount of a position to be strictly positive. The new design has moved the check to the place where the position is modified rather than where it is observed. Reasoning about the invariant helps assure us that the check is redundant, and leads to the removal of the check.

5.8 Claims Support Test Case Generation

When claims specify important properties of a specification, these are likely to be properties that should be checked in an implementation. Hence, claims can be used to motivate test cases. For example, the `assumeCentury` claim in Figure 5-14 suggests the creation of a test case that can detect when the special year interpretation is violated: each normal date should be after or equal to `"0/0/50"` and before or equal to `"12/31/49"`. Figure 5-19 shows some C code that codifies the test case. Similarly, the `date_formats` claim in Figure 5-20 explicitly lists some boundary cases for acceptable and invalid date formats.

```
bool testDate (date d) {
 date minD, maxD;
 date_parse("0/0/50", "", &minD);
 date_parse("12/31/49", "", &maxD);
 return (is_null_date(d) || date_is_LT(d) ||
         ((date_same(d, minD) || date_is_later(d, minD)) &&
          (date_same(d, maxD) || date_is_later(maxD, d))));
}
```

Figure 5-19: A test case motivated by the assumeCentury module claim in the date interface.

```
claims date_formats (void) {
  ensures okDateFormat("0/0/93") ∧ okDateFormat("1/0/93")
          ∧ ¬ (okDateFormat("13/2/92") ∨ okDateFormat("1/32/92")
                ∨ okDateFormat("1/2") ∨ okDateFormat("1/1/1993"));
}
```

Figure 5-20: The date_formats module claim in the date interface.

5.9 Experiences in Checking LCL Claims

Our methodology encourages specifiers to check claims. Claims, however, are also useful as design documentation. Even if claims are not checked, they help the readers of a specification understand the design of the specification. They also make it more likely that specification errors will be found and detected. If resources permit, however, claims should be checked so that specification errors can be uncovered and fixed earlier in the program development process.

In this section, we report our experiences in verifying LCL claims using LP [7]. We discuss how our claims methodology can be scaled up, and the kind of tools needed to better support claims verification.

5.9.1 Assessment

Formally verifying claims with a proof checker is tedious and difficult. We have found that while many proof steps are easy and quick, a few are difficult and slow. The difficulty often arises from our inadequate understanding of the implications of the specification. The key benefit of verifying claims seems to be in gaining a better understanding of our specification. As a result of the better understanding, we are able to uncover mistakes and propose corrections to the specification, thus enhancing its quality. We have given examples of the kinds of specification mistakes we found by formally verifying claims about specifications. They show that formally verifying claims can detect common classes of specification errors.

Given that formal verification of claims requires substantial effort, is it useful to check the claims informally, without the use of a proof checker? In a separate (informal) experiment, we checked two module claims of the specification of an employee data base example in [14] carefully by hand, without verification tool support. We found one error in the specification [35]. We believe that most of the mistakes we found in the specification of PM can also be found by meticulous informal claims checking. However, most of them are unlikely to be uncovered by casual inspection.

There is a spectrum of efforts in checking claims informally, ranging from casual inspection to meticulous hand proofs. Informal claims checking by casual inspection is unlikely to uncover the inappropriate choice of modeling floating point numbers using a floating point trait. A casual inspection is likely to wrongly assume that floating point numbers are commutative and associative in proofs. At the other end of the effort spectrum, meticulous informal claims checking is just as tedious and expensive as, if not more tedious and expensive than, claims verification with a proof checker.

We believe that the value of formal claims verification lies in the proof replay during regression testing of specifications. Specifications evolve, and when they do, we would want to re-check claims. Verification with the help of a proof checker reduces the effort needed to re-check claims. Re-checking of claims is very tedious and error-prone without a proof checker.

5.9.2 Modularity of Claims

An important issue about the claims methodology relates to its modularity and scalability. Claims are designed to be modular. They are conjectures about the local properties of a small group of modules. A procedure claim is about the local property of a function. A module claim asserts an invariant about a single module. The proof of a module claim depends only on the constructors and mutators of the module, which is often a small fraction of the functions of the module. Furthermore, we expect the size of the module to be small, on the order of tens of functions. The number of outputs produced by a function is relatively small. Hence, the number of modules an output claim can span is also small.

The size of the proof of a claim often depends on the size of the claim. We can often break a large claim into smaller parts. There is a limit to how useful a complicated claim will be as it reaches our limit to understand it. As such, we do not expect the size of a claim to pose a problem.

The most important factor with respect to proof feasibility, however, is the size of the axiomatization the proof of a claim needs. It is limited by the size of the axiomatization of the modules involved in the claim. The underlying LSL axiomatization of an interface can be large. For example, the `position` module in the case study corresponds to about nine hundred axioms. The size can pose a problem for some proof checkers.

While the size of a module may be big, often only a tiny fraction of the axiomatization is actually needed in a proof. For example, many of the axioms in the `position` module define operators on dates and how to parse a string into a transaction. They are not needed in the proof of the `amountConsistency` claim. An operator often appears in the proof of a claim as a condition in a case split. For example, the comparison operator on dates, `__ ≥ __: date, date → Bool`, appears in the specification of `position_update`, and hence in the proof of the `amountConsistency` claim. Its definition, however, is not needed in the proof. In the case split of the proof, we simply consider both when it is true and when it is false.

It is often useful to remove axioms not needed in a proof because the performance of some theorem provers may degrade in their presence. The human verifier often has a reasonably good idea about which sets of axioms are not likely to be useful in a proof. For example, in the proof of `amountConsistency`, axioms from the `date`, `string`, `security`, `char`, and `lot` traits are not needed. The removal of these traits and other traits supporting them reduced the number of axioms by a quarter and quickened the proof by an eighth. There are still, however, many axioms that are useless for the proof. To cut down the size of the axiom set further requires substantially more effort on the part of the human verifier. It would be far better if the verifier does not have to be concerned with this level of detail. We will return to this issue in a later discussion on the kind of mechanical prover support our methodology requires.

An attendant issue in claims verification is whether verified claims can be useful in subsequent proofs of other claims. Since LCL specifications are translated into a logical theory, they enjoy the nice properties of a logical theory: conjectures that have been proved can be used in other proofs, and proofs can be structured modularly through the use of lemmas.

5.9.3 Support for Proving Claims

There are two tools that can help in claims verification. The first is a verification condition generator. For example, a program that translates LCL specifications and claims into LP proof obligations, called *lcl2lp*, could remove the tedium of hand translation and reduce human translation errors. In the interest of research focus, we did not build such a translator since it is clear how it can be done.[5]

The second tool that can help in claims verification is a proof checker. The requirements imposed on a proof checker for verifying claims in realistic specifications are more demanding than those intended for smaller axiom sets. The following facilities in a proof checker are important to support the checking of claims.

Proof Control: A claims verifier needs good facilities to control the proof process. Since the key goal of verifying claims is testing specifications, the user must have control over the proof steps. A proof checker that takes short proof steps each time is important because proof failures are common. An automated prover often tries too hard and fails after a long search. In this regard, LP is good for verifying claims because each LP proof step tends to terminate quickly.

Theory Management: As we have discussed earlier in this section, a realistic specification often has a large number of axioms, but the proof of a claim often requires a small fraction of them. Unfortunately, we cannot predict automatically the operators whose definitions are needed in the proof of a claim until the proof is done. This means that a proof checker must be able to handle a large database of axioms, many of which are passive throughout a proof. Their presence must not degrade the performance of the prover.

It is also essential for the user to have control over the expansion of axioms. For example, in a rewrite-based prover, axioms may be expanded into forms that are unreadable and difficult to use. For small specifications, the axiom readability problem seldom arises. For realistic specifications, they are the norm.

Common Theories: We believe that a large part of the specifications of many realistic programs is not complicated. They use simple data structures such as tuples, enumerations, sets, and mappings to model their application domains. They build on well-known arithmetic theories. A proof checker should be able to reason about such data structures efficiently and without assistance from the user. A proof checker should have support for reasoning about simple arithmetic. Substantial proof effort and tedium can be reduced with special facilities for reasoning about such common theories.

---

[5] Works in traditional program verification requires a verification condition generator [17] that is more complicated than what is needed in checking specification claims.

The translation of the `position` interface specification produces an axiomatization in which a significant number of axioms are about basic facts of arithmetic and sets. Most of the domain-specific data structures are modeled using tuples, enumerations, and sets. In the position specification, they amount to more than a third of the LP axioms. Our proofs in LP are burdened by the need to supply and prove simple lemmas about tuples, sets, and linear arithmetic. About one-eighth of the position specification are axioms related to linear arithmetics, partial orders, and transitivity. Built-in specialized decision procedures for these theories will help in claims verification.

Proof Abstractions: The proofs of different invariants about a single module have a similar outer proof structure because the sub-parts of the proof obligation corresponding to an invariant are derived from the constructors and mutators of the module. They are the same for a given module, no matter what the invariant is. After completing the proof of a module invariant, it is useful to reuse the proof structure for proving other invariants of the module. Similarly, some changes in a specification may be captured by a change in the abstraction of the proof structure, rather than individual proofs.

While the reuse of the outer proof structure can be achieved by a program like *lcl2lp* which generates the proof obligations corresponding to a claim, the proof steps within the outer structure may benefit from proof procedure abstractions too. Some theorem provers provide proof methods, or *tactics* [11], to allow proof procedures to be abstracted and reused. Such facilities can help capture the inner structures of the proofs of module invariants and support their reuse.

Proof Robustness: Unlike well-known mathematical theorems and theories, claims and specifications evolve. Support for regression testing of proofs is hence more important for claims checking than it is for checking well-known mathematical theorems. A robust proof is one whose successful replay does not depend on incidental aspects of the axiomatization. For example, a robust proof should not depend on the order of presentation of the axioms to the prover, and it should not depend on prover-generated names that are not guaranteed to be context-independent. The order of axioms has no semantic logical consequences, and prover-generated names are implementation details that should have no semantic impact. A robust proof is stable, and its performance is predictable.

5.10 Claims or Axioms?

There is an alternate style of LCL specifications in which the desired invariants of a module are stated as axioms of the module. Our LCL language does not support a construct that allows invariants be given as LCL axioms of a module. If it did, the property stated by a module claim could have been given as an LCL axiom, rather than derived from the specifications of the module. Even without a means of stating axioms directly at the LCL level, a type invariant can often be given via an LSL trait by stating it as a property of the sort that represents the type. For example, the `lotsInvariant` module claim in the `lot_list` interface can be stated as the following LSL axiom in the `lot_list` trait:

```
∀ x: lot_list, e: lot (count(e, x) ≤ 1)
```

Stating the invariant as an axiom, whether at the LSL level or the LCL level, however, can easily lead to inconsistencies. For example, if the `lot_list_add` function in the `lot_list` interface were to omit the membership check, the specification would be inconsistent. The inconsistency cannot be easily detected, producing a specification which is less robust. Consideration about specification robustness led us to derive the property about the `lot_list` type as a data type invariant. The module claim is also useful for identifying unintended logical consequences when changes are made to the module.

5.11 Summary

We have introduced the concept of claims to support semantic analysis of formal specifications. Claims are logical assertions about a specification that must follow semantically from the specification.

Using LP to verify claims has helped to uncover errors in specifications. Some of these errors are not easily detected by inspection. Redundant information in formal specifications is useful for removing errors in the specifications.

The claims in the case study illustrated the use of claims as a documentation tool for highlighting interesting and unusual properties of specifications. Claims can also be used to support program reasoning, and help generate test cases. These uses also suggest specific sources for motivating problem-specific claims for a given specification.

Through examples in the case study, we provided specifiers some practical guidelines for using claims to improve specifications.

Chapter 6

Reengineering Using LCL

Many existing programs are written in programming languages that do not support data abstraction. As a result, they often lack modularity. It is difficult and expensive to maintain or extend such legacy programs. One strategy is to abandon them and build new ones, but this strategy is often not cost-effective. An alternative is to improve the existing programs in ways that make their maintenance easier.

Chapter 3 described how LCL is designed to support a style of C programming based on abstract types. Chapter 5 describes how claims can be used to highlight important design properties, support program reasoning, and promote design coherence of software modules.

In this chapter, we discuss how we apply the ideas in Chapter 3 and 5 to reengineer the original PM program into the version described in Chapter 4. The process of improving an existing program while keeping its essential functionality unchanged is termed *reengineering.* The kinds of program improvement we consider here are primarily aimed at making programs easier to maintain and reuse.

In the next section, we describe a specification-centered reengineering process model. In Section 6.2, we describe the effects of applying the process to reengineer the PM program using LCL. In Section 6.3, we categorize the impact of the reengineering process on the quality of the PM program. In Section 6.4, we discuss the importance of various specification tools we have used in writing and checking formal specifications. In the last section, we summarize this chapter.

6.1 Software Reengineering Process Model

The high-level goal of our reengineering process is to improve an existing program in ways that make its maintenance and reuse easier, without changing its essential functionality. Our process addresses three aspects of program improvement. First, we aim to improve the modularity of the existing program. This means re-structuring the existing modules of the program so that they are more independent of each other. By module independence, we mean that the implementation of a module can be changed without affecting its clients as long as the specification of the module remains unchanged.

Second, we formally document the behaviors of program modules. The specifications serve as precise documentation for the modules. Specifications play two crucial roles in

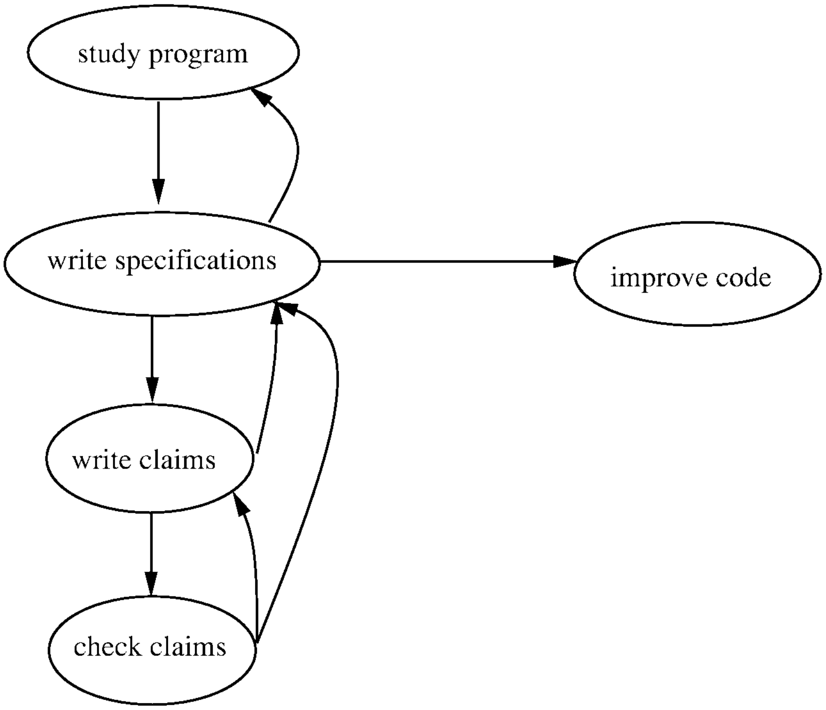

Figure 6-1: Specification-centered software reengineering process model.

software maintenance. One, the specification of a module forces the clients of the module to use the module without relying on its implementation details. Two, it clearly defines the kinds of program changes that must be considered when a module is modified. If the modification causes the specification of the module to change, then the impact of the specification change on all the clients of the module must be considered, and the effects of the change may propagate to indirect clients. On the other hand, if the modification to a module does not affect the specification of the module, then the clients of the module need not be considered since they can only rely on the specification of the module. Without specifications, we must consider the effects of the modification on all the clients of the module.

Third, we highlight important properties of various program modules. Such information can help the implementor of a module reason about the correctness of the implementation of the module. It can also guide the designer of the module towards more coherent module designs. Furthermore, it aids in the reuse of the module by highlighting some implicit design information of the module.

Our reengineering process model is depicted in Figure 6-1. An oval in the figure is a step in the process, and an arrow shows the next step one may take after the completion of a step. We outline the steps of the process below.

1. Study the existing program: First, we learn about the requirements of the program and its application domain. In this step, we also study the program to extract the structure of the program in terms of its constituent modules, and to understand the

intended roles and functionalities of these modules.

2. Write specifications for the modules of the program: In this step, we write LCL specifications for the modules of the program. This step is the most significant step of the reengineering process. It involves studying the functions exported by each module carefully, and specifying the essential behavior of most functions. Not every function in a module needs to be specified. Furthermore, it is often necessary to abstract from the specific details of the chosen implementation. The major activities in this step include choosing to make some existing types abstract, identifying new procedural and data abstractions, and uncovering implicit preconditions of functions.

3. Improve code: This step is driven by the previous specification step. While the overall requirements of the program do not change, how the requirements are met by the modules of the program can change. The specifications of the modules of the program may suggest a different division of labor among the different modules. Each time the specification of a module changes, the code has to be updated. Each change in the program is accompanied by appropriate testing to ensure that the code meets its specification. LCLint is a useful testing tool; it performs some consistency checks between the specification of a module and its implementation.

4. Write claims about the specifications of the program modules: In this step, we analyze the specification of each module and its clients to extract properties about the design of the module. We codify some of these properties as LCL claims. This step may lead to changes in the specification of a module that make it more coherent. It may suggest splitting an existing abstraction into different abstractions, performing new checks to weaken the preconditions of functions, or removing unused information kept in the implementation. Some of these specification changes may affect its clients. If a specification changes, its implementation and its client may have to be modified.

5. Check claims: We check that the claims we wrote about a module in the previous step are met by the specification of the module. Depending on the desired level of rigor, this step may range from an informal argument of why a claim should hold, to a formal proof of the claim with the help of a mechanical proof checker. This step is intended to ensure that the specifications we wrote are consistent with the understanding of the module design we have in mind. If this step leads to specification changes, the clients and the implementation of the changed specification must be updated accordingly. As indicated in Section 5.3, checking a claim can also lead to changes in the claim statement itself. This explains the arrow from the "check claims" step to the "write claims" step in Figure 6-1.

6.2 A Reengineering Exercise

We used LCL to specify the PM program, and we wrote claims to improve the specification. In the process, we improved the program in many ways. In this section, we briefly describe the specific changes we made to the program as we followed the steps of our reengineering process outlined in the previous section.

6.2.1 Study Program

In this step, we learned about the intended application of the PM program: keeping a portfolio of financial securities. The application domain includes some knowledge about how different kinds of incomes from financial transactions are treated for tax purposes.

The other major activity in this step is to extract the structure of the PM program. This is done with the help of a module dependency diagram which shows the relationships among the major modules of the program. The module dependency diagram of the original program is shown in Figure 6-2. Our interpretation of a module dependency diagram is adapted from [27]. A module dependency diagram consists of labeled nodes and arcs. The nodes represent the modules of the program. We draw an arc from module M1 to module M2 if the implementation of some function in M1 uses some function in M2. A node with a horizontal bar near its top represents an abstract type. For example, the nodes labeled `lot` and `lot_list` are abstract types. An LCL module that exports more than one type is illustrated as a node with internal nodes representing its constituent types, called an *aggregate node*. In Figure 6-2, the node containing the `lot` and `lot_list` nodes is an aggregate node. The `pm` node represents the top-level routine of the PM program.[1] The diagram captures the coarse-grained structure of the program design. It shows the modules we must consider whenever a change is made to a module. It formalizes the change propagation we describe in the previous section.

6.2.2 Write Specifications

Given that this step is the most significant step of the reengineering process, we describe the major activities in this step in more detail.

Making Some Exposed Types Abstract

The first thing we did in this step was to convert some exposed types into abstract types. An abstract type offers us more modularity: we can choose to change its implementation type without affecting its clients. An exposed type, however, can be more convenient because its interfaces are pre-defined by the programming language, and hence, we do not have to specify or implement them. In the PM program, we chose to make all the major types abstract except for two types: the `kind` type, which is a C enumeration type of different transaction kinds, and the `income` type, which is a C struct holding the different types of income from security transactions. We prefer abstract types for the others because an abstract type limits the modifications its clients can make. This allows our design of the type to maintain module invariants that may be useful to the clients and the implementor of the module. The `kind` and `income` types were not made abstract because we did not find useful invariants about the types and because making them abstract would have forced us to export many simple interfaces.

---

[1] We leave out the `genlib` module in the module dependency diagram because it is used by most modules. Including it would clutter up the diagram without giving new insights into the program structure. The module should be considered as a general utility, like a standard C library.

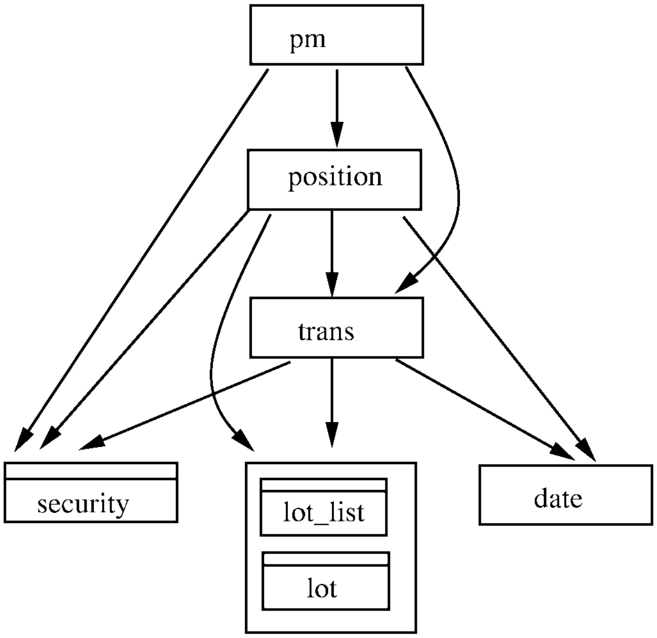

Figure 6-2: Module dependency diagram of the original PM program.

Specifying Appropriate Abstractions

By studying the functions exported by a module and its clients, it is often easy to identify functions that are auxiliary. Such functions need not be exported or specified in the interfaces. For example, in the `date` module, we found that the `day_of_year` and `days_to_end` functions in the original program were there to support the exported `is_long_term` date function.

For those functions that must be exported, it is important to specify only their essential behaviors, and not incidental implementation details. For example, in the original program, the open lots of a position was represented as an array of transactions in a field of the position type. While specifying the behavior of the `position` module, it became clear that the use of an array to represent the open lots was incidental, and the open lots should be modeled more abstractly, as a set of buy transactions. Hence, a new module, the `trans_set` module, was created to codify this abstraction.

Highlighting Implicit Preconditions

One of the more difficult activities in the reengineering process is identifying the assumptions a function or a module makes about its clients. These assumptions are often not written down, and are difficult to infer from the code without careful analysis. The effort, however, is useful because future program maintenance is facilitated if the assumptions are made explicit. Once the assumptions are made explicit, we can often weaken them to make the function or module more robust.

For example, the old `position` module was designed to be used in a rather specific way:

it processed batches of transactions about one security that were sorted by their dates. Unfortunately, the key function in the module, the `position_update` function, did not check that the security of the input position and the security of the input transaction were the same. The function assumed that its caller would guarantee the same security precondition. We first made explicit the precondition in the specification of the `position_update`. As a further improvement, we weakened the precondition by adding explicit checks in the `position_update` function so that the function is more robust.

Other examples of identifying and weakening implicit preconditions include adding bounds checks on the input arrays in the `get_line` function of the `genlib` module, adding length checks on date formats in the `date_parse` function of the `date` module, and identifying the constraint that the date of a sell transaction must not be `"LT"` in the `trans` module.

6.2.3 Improve Code

The modifications to the PM program were driven by changes made to the specification of its constituent modules. We used the LCL specifications of the modules to improve the program with the help of the LCLint tool. LCLint uncovered some inconsistencies between the implementations of the modules and their specifications. The errors included type barrier breaches, modifying objects that should not be changed, and accessing globals not sanctioned by the specification. The LCLint tool improved the quality of the PM program by uncovering flaws in the code.

6.2.4 Write Claims

The process of writing claims about the modules of the PM program led to a number of program improvements. In the `trans` module, we specified the constraints that must hold for the different fields of a transaction as claims. Our claims led us to add new checks on the user's input transactions. For example, for a Treasury bill maturity transaction, its interest payment is checked for correctness, and partial lots are flagged as errors. In addition, a buy transaction must have a strictly positive amount and a single lot.

The use of claims in specifications helped to promote design coherence in software modules. The effect has already been discussed in Section 5.6. In the `trans` module, arbitrary changes to the net and amount fields of a transaction were replaced by controlled changes that respect module invariants. In the `position` module, the second distinct use of the old `position` type was extracted and codified as a separate `income` type.

6.2.5 Check Claims

This step is needed to ensure that the claims we made in the previous step follow from our specifications. In our exercise, the checking of claims led to uncovering some of the new checks in the PM program. For example, the `amountConsistency` claim in the `position` module states that in the absence of input errors, the amount of a position should be the sum of the amounts of the transactions in the open lots of the position, and the `noShortPositions` claim says that in the absence of input errors, the amount of a position should be non-negative. When we tried to prove these claims, we realized that buy transactions must have non-negative amounts in order for the claims to hold. On further analysis, we decided that the amount of a buy transaction should be strictly positive because it does not make sense to buy zero shares of a security. This constraint was not enforced in the original program. Adding this check to the `trans` module changes the specification of the `trans` module, and led us to consider input checks that could be performed on other kinds of transactions.

The other improvements in the specifications of the PM program that resulted from the checking of claims have already been described in Section 5.3. Since the specifications of the program are an integral part of its design, the claims checking step contributed significantly to the quality of the products of our reengineering process.

6.3 Effects of Reengineering

In this section, we summarize the effects our reengineering process had on the functionality, structure, performance, and robustness of the program. While most of the effects to be mentioned in this section have been attributed to specific steps in the reengineering process in the previous section, some are not easily attributed to any specific step. We also discuss the important role the formal specifications of the main modules of the PM program play in the maintenance and reuse of the program.

6.3.1 Effects on Program Functionality

The functionalities of the original and the new programs are essentially the same. This was one of the goals of our reengineering process. We believe that the service provided to the user of the PM program is improved as a result of our reengineering process. The service provided by the new program improved because the process helped to identify useful checks on the user's inputs to the program. For example, many of the checks on transaction and date formats were new. The new program also ensures that dividend and interest transactions do not initialize a position. These checks help catch a class of data entry errors in which the name of a security is misspelled.

6.3.2 Effects on Program Structure

Our reengineering process had significant impact on the structure of the PM program. We observed three kinds of effects. Two of these effects can be illustrated by comparing the module dependency diagrams of the two programs. The module dependency diagram of the new PM program is shown in Figure 6-3.

The most significant effect our reengineering process had on the PM program was to improve the modularity of the program. The original program was already designed in a structured manner. There were clear module boundaries; for example, definitions for different conceptual types named by `typedef` were kept in different modules. There were no global variables across modules. There was, however, no separation between the representation type and the conceptual type. As a result, if the representation of a transaction was modified, clients such as the `position` module might have to change.

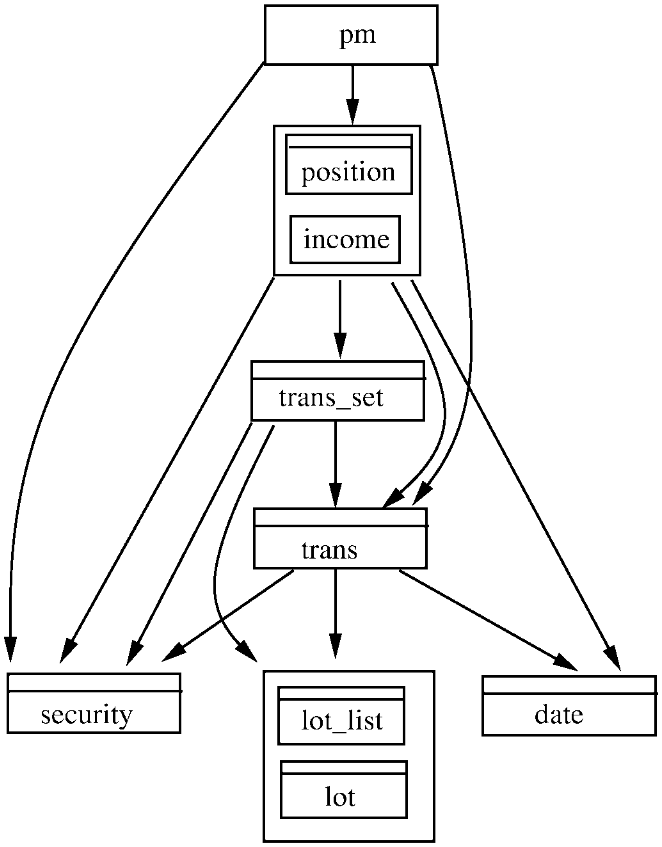

Figure 6-3: Module dependency diagram of the new PM program.

Abstract types create protective barriers, preventing arbitrary changes to instances of the types. The implementation of abstract types can be modified without affecting their clients. As the new diagram in Figure 6-3 shows, the following exposed types were made abstract: `date`, `trans`, and `position`.

We observed an adverse effect of using abstract types in programming. An abstract type necessitates the specification and implementation of more interfaces than if the type were exposed. For example, the original PM program relied on C built-in operators of exposed types; it resulted in a more compact source program.[2] For example, a transaction is represented as a C struct in the original program, so there is no need to export functions to return the respective fields of a transaction. Such functions, however, must be explicitly specified and implemented in the new program after a transaction is made into an abstract type. While more interfaces are needed, program efficiency is not sacrificed because many of these extra interfaces needed in the new program are implemented as macros.

The second beneficial effect we observed in the reengineering process is that it helped to suggest new abstractions that were overlooked in the original program. As described in the previous section, the `trans_set` module is a new module; it did not exist in the original program. This change is clearly reflected in the extra node labeled `trans_set` in Figure 6-3. In the old program, it was part of the `position` module.

---

[2] Several other factors contributed to a larger new source program: we added new checks on the inputs of the program, and checks to weaken the preconditions of some functions. We estimate that the use of abstract types in the PM program caused it to increase its size by about 10%.

While the new diagram looks more cluttered than the old one, it is in fact a more modular design than the old one. The new program is more modular because changes in the choice of rep types of the abstract types exported by the modules do not affect their clients. The module dependency diagrams in Figure 6-2 and Figure 6-3 show that the structure of the original PM program was retained. The new program did not introduce new dependencies between the program modules other than those due to the additional `trans_set` module.

The third beneficial effect we observed is that the use of claims in specifications helped to promote design coherence in software modules. The effect has already been discussed in Section 5.6. The subtle improvement does not show up in the module dependency diagrams of the two programs since the diagrams only capture the coarse-grained structure of the respective designs.

6.3.3 Effects on Program Performance

While we have not carried out careful controlled experiments on the execution time performance of the two programs, tests indicate that they are comparable. Since we have not changed the basic algorithms used in the program, we do not expect any significant difference in the execution time performance of the two programs.

6.3.4 Effects on Program Robustness

We believe that the reengineering process improved the robustness of the program. The new program is more robust because the process helped to remove some potential errors in the program. For example, the interpretation of the two-digit year code of a date relied on the assumption that no dates go beyond the year 2000. The turn of the century, however, is only a few years away. The new program changed the interpretation to handle this problem while still retaining the convenience of a two-digit code.

Another error detected during the reengineering process was in the implementation of the `get_line` function in the `genlib` module alluded to in Section 6.2.2. The function read a line from an input character stream into a caller-allocated character array. The original program did not check the bounds of the input array. If a line in the stream was longer than the input array, the program could crash. The potential problem showed up easily in the specification because the rigors of formally specifying the function forced us to make explicit the assumed preconditions. In the new program, we made `get_line` take an additional parameter indicating the length of the input array so that the bounds check could be done.

Similarly, the specification of `date_parse` in the `date` module forced us to delineate the acceptable format of transaction dates. The original program did few checks on the string representation of a date so that bad dates with months or days that were more than two digits could cause it to crash.

6.3.5 Documentation of Program Modules

A major product of the reengineering process is the formal specifications of the main modules of the program. This documentation was used to improve the implementation with the help of the LCLint tool. With modules that are documented, future changes to the program will be easier and parts of the program are more likely to be reused than if the formal documentation were absent.

In addition, claims are used to document properties of the modules of the program. For example, the `trans_setUID` module claim in the `trans_set` module described in Section 5.7 illustrates how a module claim can aid in reasoning about the implementation of the module. The `noShortPositions` claim and the `okCostBasis` claim in the `position` module described in Section 5.5 illustrate how module claims can be used to document some central properties of the PM program. Highlighting these properties improves the quality of the documentation of program modules in PM.

6.4 Specification Tool Support

The reengineering exercise would have been much harder and much less useful if there had been no tools to support the writing and checking of specifications. Five kinds of specification tools were used.

The first kind of tool checks the syntax and type correctness of specifications. They are the LSL and LCL checkers.[3] They caught many careless typos and type errors in our specifications.

The second kind of tool checks formal proofs. The proof checker we used was LP, a first-order term-rewriting proof checker. It was used to verify LCL claims. It was instrumental in catching the mistakes we found in the specification. As indicated in Section 5.9.1, many of the errors were not likely to be detected by inspection, or even by informal proofs. The chief benefit of using a proof checker lies in the regression testing of specifications.

The third kind of tool translates specifications. The LSL checker has an option that translates an LSL trait into an LP theory. This facility lessens the errors made by hand translation, and makes it easier to do regression testing. As indicated in Section 5.9.3, a similar program that translates LCL specifications and claims into LP inputs would have been useful.

The fourth kind of tool is previous Larch specifications. The Larch handbook contains many commonly used abstractions. Reusing the handbook traits saved us much specification effort. The traits also served as models of the LSL specifications we wrote for the case study. Similarly, we were able to adapt some LCL specifications from Chapter 5 of [15]. As the Larch handbook and the suite of Larch specifications grow larger, we expect specification reuse to increase.

The fifth kind of tool performs consistency checks between a specification and its implementation. As described in Section 6.2.3, the LCLint tool improved the quality of the PM program by uncovering flaws in our implementation. A pleasant effect of LCLint was that it also helped to uncover specification errors. An earlier specification of `date_year` did not list `stderr` as one of the objects it accessed. Its implementation, however, modified `stderr` when an error was encountered. The LCLint tool reported the inconsistency between the two, and led to a correction in the specification.

---

[3] The LCL checker is incorporated into the LCLint tool.

6.5 Summary

In this chapter, we gave a reengineering methodology centered around formal specifications. It is aimed at making an existing program easier to maintain and reuse while keeping its essential functionality unchanged.

We described the results of applying our methodology to reengineer the PM program. The most visible product of our reengineering exercise is the formal specification of the main modules of the program. The specifications serve as precise documentation for the modules. Besides the new specification product, the process helped to make the program more modular, uncovered some new abstractions, and contributed to a more coherent module design. In addition, the reengineering process improved the quality of the program by removing some potential errors in the program and improving the service provided by the program. We have achieved these effects without changing the essential functionality or performance of the program.

While some of the benefits of the reengineering process described in this chapter could be obtained with careful analysis and without specifications, we believe that our specification-centered reengineering process provides a methodology by which the benefits can be brought about systematically. Formal specifications have an edge over informal ones because of their precision. The precision sharpens the analysis process, and leaves no room for misinterpretation of the specification. Formal specifications are also more amenable to mechanical tool support, the use of which improved the program and the specification considerably.

Chapter 7

The Semantics of LCL

In this chapter, we formalize some informal concepts introduced in the earlier chapters, and describe interesting aspects of the semantics of LCL.

LCL is designed for formally specifying the behaviors of a class of sequential ANSI C [1] programs in which abstract types play a major role.[1]

The semantics of LCL described in this chapter is intended for reasoning about LCL specifications, not for program verification. The latter requires a formalization of the semantics of C which is beyond the scope of this work. As such, we do not provide a formal satisfaction relation that can be used to formally verify that a C program meets its LCL specification. Our approach of giving semantics to LCL builds on other works on Larch interface languages [38, 16, 3, 37].

The basic LCL semantic concepts are described in the next section. The storage model of LCL is formalized in Section 7.2. The type system of LCL is described in Section 7.3. The semantics of a function specification is given in Section 7.4, and that of a module specification is given in Section 7.5. A discussion of the assumptions made in the given semantics of LCL, and the technical issues that arise when the assumptions are violated, are given in Section 7.6. A summary of the chapter is given in the last section.

7.1 Basic LCL Concepts

In the semantic model of LCL, there are two disjoint domains: `objects` and `bvalues`, or basic values. Basic values are mathematical abstractions, such as integers, sets, and stacks. An object can contain another object as part of its value. It is an abstraction of a region of data storage. A third domain, `values`, is formed by the union of objects and basic values. The mapping of an object to its value is called a *state*.

Since an LCL specification is an abstraction of a class of C implementations, we call the states that the specification constrains the *abstract states*. In this chapter, whenever we refer to a state without qualification, we mean an abstract state. There is a corresponding state for the implementation of an LCL specification; it is called the *concrete state*.

There are two kinds of user-definable LCL abstract types: mutable types and immutable types. Instances of a mutable types are modeled by objects. Instances of immutable types are modeled by basic values. The semantics of C built-in types provide the semantics of LCL exposed types. The following gives the domain equations of our LCL semantic model.

---

[1] LCL does not support the specification of C programs that have procedure parameters.

`values` $\equiv$ `bvalues` $\cup$ `objects`
`states` $\equiv$ `objects` $\rightarrow$ `values`
`objects` $\equiv$ `mutable_objects` $\cup$ `exposed_objects`
`exposed_objects` $\equiv$ `locations` $\cup$ `structs` $\cup$ `unions` $\cup$ `arrays` $\cup$ `pointers`

The basic unit of an LCL specification is the specification of a C function. It states explicitly what relationship must hold between the state before the function is run (the pre state) and the state after the function completes (the post state). A key feature of LCL function specifications is that each of them can be understood independently of other function specifications.

A *module* specification consists of a number of global declarations. These include global constants, global variables, and function specifications.

An LCL module is useful in three ways. First, it supports the structuring of specifications. A module allows grouping of functions. One module can import another module. Second, it supports the specification of abstract types. Third, it allows private specification objects to be introduced in a module that are not exported to importing modules. Private objects and types are called *spec* variables and *spec* types respectively. The only place spec variables can be accessed is in the functions of the module in which they reside. Spec variables can be used to model C static variables. Spec variables are, however, more general, and they need not be implemented.

Each LCL specification implicitly or explicitly makes use of LSL specifications, called *traits*. An LSL trait introduces *sorts*, or named sets of basic values, and mathematical functions, called *operators*, on the sorts. Traits also contain axioms that constrain operators. The precise semantics of LSL traits is given in [15]. To understand this chapter, it suffices to know that a trait provides a multi-sorted first-order theory with equality for the operators and sorts of the trait.

The semantics of LCL is greatly simplified if we assume that implementations of abstract types do not *expose* the reps of the abstract types. If the reps are not exposed, a simple data type induction schema can be derived. This induction schema allows us to deduce inductive properties of the data type. It allows a property to be deduced locally from a module, and it enables the property to be applied globally, in all client contexts. Such stronger properties are often useful in reasoning about specifications and in program verification. The semantics of LCL given in this chapter assumes that the reps of abstract types are not exposed. Section 7.6 contains a discussion of this assumption.

7.2 LCL Storage Model

An LCL specification describes the behavior of a C function by the changes made to a *state*. A state is a function that maps LCL objects to their values.

In an LCL specification, there are three predefined state variables: `pre`, `post` and `any`. `pre` refers to the state before a C function is invoked, `post` refers to the state after the function returns. An LCL function specification constrains the relationship between the pre state and the post state of the function. In specifying invariants that are maintained across different states, there is a need for a generic state: `any` is used to refer to such a state.

Certain kinds of objects are immutable; their values do not change at all. We write them without their state retrieval functions (^, ′, or ~).

Our model of the abstract state and typed objects is similar to that of Larch/Generic [3].

7.2.1 LCL Abstract State

LCL objects are abstractions of computer memory locations. LCL and LSL constants are modeled as basic values.

We model LCL global variables as LCL objects; they are mutable objects whose values can change from state to state. Similarly, an LCL `spec` variable is also an LCL object. A global variable must be implemented but a spec variable need not be implemented.

The storage model of LCL is formalized as an LSL trait in Figure 7-1. An LCL abstract state is a mapping of untyped objects to their untyped values. We choose an untyped domain primarily for ease of explanation. A secondary reason is to remind ourselves that untyped objects correspond to bit patterns in memory locations.

The domain of a state is the universe of all existing objects in that state. A new state can be created by adding to an old state a binding of an object with the value of the object. A new state can also be created by only adding a new object without its corresponding value. In this case, such an object can be referenced but its value is not guaranteed to be valid. The state `nil` is the empty state. The value of an object in a state can be retrieved by the infix operator `#`. The value retrieved by `#` is the outermost binding, that is, the most recently stored one. The `trash` operator removes all copies of an object from a state if the object is present in the state.

7.2.2 Typed Objects

The state described so far is an untyped one: the objects and the values are both untyped. Like C, LCL is statically typed. Each LCL variable has a unique type. To relate typed objects with untyped ones, we have the `typedObj` trait shown in Figure 7-2.

The operator `widen` maps a typed object into an underlying untyped object. Its inverse operator is `narrow`. These operators are overloaded to handle typed and untyped values. For convenience, operators on untyped objects are overloaded to accept typed objects.

The `typedObj` trait is parameterized by two sorts: `TObject` and `TValue`. This trait is intended to be instantiated by specific sort names such as `set` and `set_Obj`.[2]

7.3 LCL Type System

We provide semantics only for LCL specifications that meet the static semantics of LCL. Fundamental to understanding this static semantics is the implicit mapping of LCL types to their corresponding unique LSL sorts. Through this type-to-sort mapping, each LCL variable

---

[2] A LCL mutable abstract type is modeled by a value sort and an object sort, see Section 7.3.2.

```
state: trait
 includes Set (object, objectSet)
 introduces
   nil: → state
   bind: object, value, state → state
   allocate, trash: object, state → state
   domain: state → objectSet
   __ ∈ __: object, state → Bool
   __ # __: object, state → value
 asserts
  state generated by nil, bind, allocate
  state partitioned by ∈, #
  ∀ st, st2: state, x, y: object, v: value
    domain(nil) == {};
    domain(bind(y, v, st)) == insert(y, domain(st));
    domain(allocate(y, st)) == insert(y, domain(st));
    x ∈ st == x ∈ domain(st);
    x # bind(y, v, st) == if (x = y) then v else x # st;
    x # allocate(y, st) == x # st;
    trash(x, nil) == nil;
    trash(x, bind(y, v, st)) == if x = y then trash(x, st)
                                  else bind(y, v, trash(x, st));
    trash(x, allocate(y, st)) == if x = y then trash(x, st)
                                  else allocate(y, trash(x, st));
 implies ∀ st: state, x, y: object, v1, v2: value
   not(x = y) ⇒ (bind(x, v1, bind(y, v2, st)) =
                  bind(y, v2, bind(x, v1, st)));
   converts domain, __ ∈ __: object, state → Bool, #, trash
    exempting ∀ x: object x # nil
```

Figure 7-1: Trait defining LCL storage model.

is given a sort according to the type of the variable and its LCL type. In this section the type system of LCL and the type-to-sort mapping are described.

Abstract Syntax

type	*::=*	*abstract* | *exposed*
abstract	*::=*	*[* **mutable** | **immutable** *]* **type** *id* **;**
exposed	*::=*	**typedef** *lclTypeSpec* { *declarator* *[* { *constraint* } *]* }$^+$, **;** | { **struct** | **union** } *id* **;**
constraint	*::=*	**constraint** *quantifierSym id* **:** *id* **(** *lclPredicate* **)** **;**

A *type* is either an abstract type or a synonym for an exposed type. Abstract types can either be immutable or mutable. The design of abstract types in LCL is inspired by CLU

```
typedObj (TValue, TObject): trait
 includes state
 introduces
  widen: TObject → object
  widen: TValue → value
  narrow: object → TObject
  narrow: value → TValue
  __ # __: TObject, state → TValue
  __ ∈ __: TObject, state → Bool
  bind: TObject, TValue, state → state
 asserts
  TObject generated by narrow
  TObject partitioned by widen
  ∀ to, to2: TObject, x: object, tv: TValue, v: value, st, st2: state
   narrow(widen(to)) = to;
   widen(narrow(x)) = x;
   narrow(widen(tv)) = tv;
   widen(narrow(v)) = v;
   to # st == narrow(widen(to) # st);
   bind(to, tv, st) == bind(widen(to), widen(tv), st);
   to ∈ st == widen(to) ∈ st;
   to = to2 == (widen(to) = widen(to2));
 implies ∀ to, to2: TObject, st: state
   widen(to) # st == widen(to # st);
```

Figure 7-2: Trait defining typed objects.

[27]. The description of LCL abstract types has already been given in Chapter 3.

An exposed type can be named using a `typedef` in the same way types are named in C. In addition, LCL allows a constraint to be associated with a type name introduced by `typedef`. It is useful for writing more compact specifications.

The detailed syntax for *declarators* and *lclTypeSpec* are given in Appendix A. It suffices to know that they generate the type system of C, supporting the primitive types of C, pointers, arrays, structs, unions, and enumeration types. They follow the same grammar and semantic rules as those of C. The only new feature is the addition of the `out` type qualifier for documenting outputs that are returned via an input pointer. This feature is discussed in Section 7.5.3 when its use in data type induction is explained.

Checking

- The identifier introduced by a type declaration must be new; it must not be already declared.
- The only state retrieval function that can appear in the *lclPredicate* in the *constraint* production is `any`.

7.3.1 LCL Exposed Types

The semantics of LCL exposed types is given by the semantics of C built-in types. The constraints on the type system of C are not described here; they can be found in the ANSI C standard [1] or [24]. The exceptions to the type compatibility rules of C are noted below:

- The following types are considered different types: `int`, `char`, and C enumeration types.
- An array of element type T and a pointer to T are considered distinct types.
- The following types are not distinguished: `float` and `double`.
- C type qualifiers, e.g., `volatile` and `const`, are not significant in LCL. If they appear in an LCL specification, they are ignored.

These differences are not fundamental to the design of LCL.

The semantics of C aggregate types provides the following non-aliasing information: Given a C array, each different index refers to a distinct object in the array, different from other objects in that array. That is, we have the axiom:

```
∀ a: array, i, j:int (0 ≤ i ≤ maxIndex(a) ∧ 0 ≤ j ≤ maxIndex(a)) ⇒
                          (i = j ⇔ a[i] = a[j])
```

Note that $a[i]$ refers to the object in the array `a` with offset `i`, not the value of this object. The latter value is obtained by applying a state function to $a[i]$, such as $a[i]^\wedge$.

Similarly, for identically typed fields in a C struct, there are corresponding object inequality assertions.

Exposed Types with Constraints

An exposed type with a non-empty constraint is *not* a new type; it is a type synonym. The constraint specification is useful as a shorthand for writing more compact specifications. For example, consider the following specifications:

```
typedef int nat {constraint ∀ n: nat (n ≥ 0)};
nat P (nat n) {
  ensures true;
}
int P2 (int n) {
  requires n ≥ 0;
  ensures result ≥ 0;
}
```

The meaning of the specification of `P` is the same as that of `P2`. When `nat` is the type of an input parameter of a function specification, the constraint associated with `nat` is assumed to hold for the input parameter in the pre state. Similarly, if `nat` is the type of an output parameter, then the corresponding constraint must implicitly hold in the post state for the output parameter.

uses traitName (typeName for sortName, ...)		
LCL type, T	*typeName = T*	*typeName = obj T*
immutable, I	I	I_Obj
mutable, M	M	M_Obj
primitive type T	T	T_Obj
enumerated type T	T	T_Obj
pointer to T	T_ObjPtr	T_ObjPtr_Obj
array of T	T_Vec	T_ObjArr
struct _tag	_tag_Tuple	_tag_Struct
union _tag	_tag_UnionVal	_tag_Union

Table 7.1: Mapping LCL Types to LSL sorts in the LCL uses construct.

7.3.2 Linking LCL Types to LSL Sorts

Each LCL type is modeled by one or two LSL sorts. A mutable abstract type M is modeled by two LSL sorts: `M` and `M_Obj`. The `M_Obj` sort is used to model LCL objects of type M, and it is called the *object sort* of M. The `M` sort is used to model the value of M objects in a state, and it is called the *value sort*. An immutable abstract type I is modeled by one LSL sort, `I`, its value sort, because the object identities of immutable objects are not expressible in LCL.

Since each LCL type can give rise to more than one LSL sort, we must define which unique sort an LCL type expression corresponds to. An LCL type expression often occurs together with an LCL variable such as when the variable is declared as a global variable or a formal parameter. The mapping of such LCL type expressions are given in Appendix B. In this subsection, we describe how an LCL type expression in the `uses` clause is mapped to its underlying LSL sort.

Table 7.1 shows the implicit LSL sorts generated to model LCL types. Each entry in the second column of a row is called the *value sort* of its corresponding first column, and the entry in the third column is called the *object sort* of its corresponding first column. By default, an LCL type expression is mapped to its value sort. If the corresponding object sort is desired, the `obj` qualifier can be used.

The following example illustrates how the `obj` type qualifier is used to model object identities. Consider the specifications of two C functions shown in Figure 7-3 and Figure 7-4. Both specifications use the same stack trait; the relevant part of the stack trait is shown in Figure 7-5. The two specifications are identical except in the two places indicated in the figures.

The first difference lies in the use of the type to sort renaming in the `uses` construct (in the third lines) of Figure 7-3 and Figure 7-4. In Figure 7-3, the value sort corresponding to the `mset` type is used in renaming, and in Figure 7-4, the object sort is used. The second difference lies in the use of $\in$ in the ensures clause in the figures.

The `member` function in Figure 7-4 returns true if a given set *object* is in the input stack; in Figure 7-3 it returns true if the *value* of the given set object in the pre state is in the input stack. The two membership relations have different signatures: the first $\in$ takes a set object and a stack whereas the second $\in$ takes a set value and a stack. The `member` function

```
mutable type mset;
immutable type stack;
uses Stack (mset for E, stack for C); /* __ ∈ __: mset, stack → Bool */
bool member (mset s, stack st) {
  ensures result = s^ ∈ st;          /* s^: mset */
}
```

Figure 7-3: Modeling a stack of set values in LCL.

```
mutable type mset;
immutable type stack;
uses Stack (obj mset for E, stack for C); /* __ ∈ __ : mset_Obj, stack → Bool */
bool member (mset s, stack st) {
  ensures result = s ∈ st;               /* s: mset_Obj */
}
```

Figure 7-4: Modeling a stack of set objects in LCL.

in Figure 7-3 returns true whenever `member` in Figure 7-4 returns true, but it does so even when the given set object does not belong to the stack but its value happens to be equal to the value of some set in the stack.

7.4 LCL Function Specification

The LCL specification of a C function specifies a type for the return value of the function, a name for the function, some formal parameters with their types, an optional list of global variables that the function accesses, and a function body. The function body can have a `let` clause to abbreviate common expressions, a `requires` clause, a `checks` clause, a `modifies` clause, and an `ensures` clause.

A function specification may be preceded by some type and global variable declarations in a module. They form the *scope* of the function specification. The scope includes the declarations imported by the module enclosing the function specification.

Abstract Syntax

fcn ::= *lclType fcnId* (**void** | {*lclType id* }$^+$,) { *global* }* { *fcnBody* }
global ::= *lclType id*$^+$, ;
fcnBody ::= [*letDecl*] [*requires*] [*checks*] [*modify*] [*ensures*] [*claims*]

```
Stack(E, C): trait
  introduces  __ ∈ __: E, C → bool
  ...
```

Figure 7-5: Part of a Stack Trait

letDecl ::= **let** { *id [* **:** *lclType]* **be** *term* }$^{+}$, ;
requires ::= **requires** *lclPredicate* ;
checks ::= **checks** *lclPredicate* ;
modify ::= **modifies** { **nothing** | *storeRef*$^{+}$, } ;
storeRef ::= *term* | *[* **obj** *]* *lclType*
ensures ::= **ensures** *lclPredicate* ;
claims ::= **claims** *lclPredicate* ;

Checking

- Every LCL type appearing in a function specification must already be declared in the scope of the function.
- In the body of a function specification, each identifier must either be an LCL variable, a variable bound by some quantifier, an operator in an used LSL trait, or a built-in LCL operator. Each LCL variable must appear either as a formal parameter of the function, or a listed global variable (in *global*) of the function, but not both. Identifiers introduced by the let clauses are macros.
- The sort of the term in the let clause must match the sort corresponding to the declared type.
- Every global variable (*global*) must already be declared in the scope of the function and accessible to the function.
- In an LCL module, no two type or function declarations can declare the same name.
- The only state retrieval function that can appear in the requires clause is the pre state.
- Every item in the modifies list is either a term or a type. If it is a term, it must denote a mutable object. If it is an LCL type, the type must correspond to a mutable type.

Meaning

The LCL specification of a C function is a predicate on two states, the pre state and the post state. This predicate indicates the relationship between the input arguments and the result. Furthermore, there is always an implicit requirement that the function terminates. If the function does not terminate, then there is no post state.

The requires clause specifies a relation among objects denoted by the formal parameters and the global variables in the pre state. An omitted requires clause is the same as `requires true`. The ensures clause is similar, except that it relates the values of these objects in the pre and the post states. An omitted ensures clause is the same as `ensures true`.

The checks clause is like the requires clause except that instead of the caller ensuring that the conditions specified are met before the procedure call is made, it is the implementor's job to check those conditions. It is a convenient shorthand for specifying conditions that the implementor must check. If the conditions are not met, the implementor must print an error message on `stderr` (the standard error stream of C), and halt the program. If the checks clause is present, `*stderr^` is implicitly added to the modifies clause and the global list.

The let clause does not add anything new to the specification. It is a syntactic sugar to make the specification more concise and easier to read. If a let clause introduces more than one abbreviation, the abbreviations nest, that is, a later abbreviation can use earlier ones in its definition.

7.4.1 Translation Schema

The meaning of an LCL function specification P is given schematically by:

```
RequiresP ⇒
 (ModifiesP
  ∧ if ChecksP then EnsuresP ∧ StdErrorChanges
     else halts ∧ ∃ errm: cstring (appendedMsg((*stderr^)', (*stderr^)^,
                                              FatalErrorMsg || errm)))
```

where `RequiresP` stands for the requires clause of the function, `ChecksP`, the checks clause, `ModifiesP`, the translation of the modifies clause, and `EnsuresP`, the ensures clause. The object `*stderr^` is implicitly added to the modifies clause and the list of globals accessible by the function. The `StdErrorChanges` is defined to be `true` if the specifier explicitly adds `*stderr^` to the modifies clause or if the checks clause is absent, and `unchanged(*stderr^)` otherwise. This semantics allows a specifier to override the default assumption that the standard error stream is unchanged if the checks clause holds by explicitly adding the standard error stream to the modifies clause. An omitted checks clause means `ChecksP = true`.

We define *specs(P, pre, post)* to be the following logical translation of a function specification P:

```
specs(P, pre, post) ≡ RequiresP ∧ ChecksP ∧ ModifiesP ∧ EnsuresP
```

specs(P, pre, post) is a predicate that must be true of the pre state and the post state of successfully executing P. This abbreviation will be used in the translation of claims.

The *input arguments* of a function consist of the formal parameters and the global variables that the function accesses. The set of objects that appear explicitly or implicitly in the modifies clause of a function specification is called its *modified set*. The *output results* of the function consist of `result` (which stands for the return value of the function) and the modified set of the function.

The set of fresh objects in a function specification is called its *fresh set*. Similarly, the set of trashed objects in a function specification is called its *trashed set*.

We view an instance of each C aggregate type as a collection of the underlying objects that comprise it, called *base objects*. The type of a base object is either one of the primitive types of C, or some abstract type. Given a function F, we define the *input base arguments* as the base objects corresponding to the input arguments of F. Similarly, the *output base results* are the base objects corresponding to the output results of F. Consider, for example, the type specifications given below.

```
typedef struct {int first; double second;} pair;
typedef struct _tri {int arr[10]; pair match; struct _tri *next;} tri;
```

If a function F takes a single formal parameter, `x`, of type `tri` and no global or spec variables, then its base input arguments are

$$\{\texttt{x.arr[i]} \mid \texttt{0} \leq \texttt{i} < \texttt{10}\} \cup \{\texttt{x.match.first},\ \texttt{x.match.second},\ \texttt{x.next}\}$$

The notion of base objects is useful in formalizing the meaning of the modifies clause in Section 7.4.3.

7.4.2 Implicit Pre and Post Conditions

There are two kinds of implicit pre and post conditions associated with a function specification. First, we require that objects in the pre state do not disappear without being trashed in the post state, or

$\texttt{(domain(pre) - trashedObjs)} \subseteq \texttt{domain(post)}$

where `trashedObjs` is the trashed set of the function.

Second, if a parameter of the function is an exposed type with a constraint, some conditions are implicitly added. If the constraint predicate associated with an exposed type T is named *constraint: T → Bool*, we conjoin to the requires clause the following conditions:

- `constraint(x)[^ for ~]` if `x` is a non-array and `x` is a formal parameter of type `T` of the function.
- `constraint(x^)[^ for ~]` if `x` is an array and `x` is a formal parameter of type `T` of the function.
- `constraint(x^)[^ for ~]` if `x` is a global parameter of type `T` of the function.

The first condition differs from the latter two conditions because C uses pass by value in procedure calls, except for arrays. Global variables are modeled by LCL objects. Similarly, we conjoin to the ensures clause the following conditions:

- `constraint(x)[' for ~]` if `x` is a non-array and `x` is an output result of type `T` of the function.
- `constraint(x')[' for ~]` if `x` is an array and `x` is an output result of type `T` of the function.
- `constraint(x')[' for ~]` if `x` is a global parameter of type `T` and is an output result of the function.

7.4.3 The Modifies Clause

The translation of the modifies clause is:

$\forall\ \texttt{i: AllObjects ((i} \in \texttt{domain(pre)} \wedge \texttt{i} \notin \texttt{modifiedObjs)} \Rightarrow \texttt{i'} = \texttt{i\^{})}$

where `modifiedObjs` denotes the modified set of the function. `AllObjects` is the disjoint sum of the object sorts of the mutable types.

We next explain how the modified set of a function specification is constructed. The modifies clause can take three kinds of arguments.

modifies term: Suppose the term `x1` of type `T1` is in the modifies clause. There are two cases here. T1 can be a mutable abstract type or an exposed type. If T1 is a mutable abstract type, then `x1` is a member of the modified set. If T1 is an exposed type and if `x1` denotes an object that has one of C's primitive types (such as `int`) as its value, then `x1` is a member of the modified set.

If `x1` denotes a struct object or an array object, the meaning of `modifies x1` is defined in terms of its base objects. For example, using the specifications of the `tri` type given in Section 7.4.1 and the specification of F below,

```
void F (tri *t) {
  modifies *t;
  ensures ...
}
```

the modifiedObjs of F is given by

$$\{(*\mathtt{t}).\mathtt{arr}[\mathtt{i}] \mid \mathtt{0} \leq \mathtt{i} < \mathtt{10}\} \cup \{(*\mathtt{t}).\mathtt{match.first},\ (*\mathtt{t}).\mathtt{match.second},\ (*\mathtt{t}).\mathtt{next}\}$$

An instance of a C aggregate type is simply viewed as a shorthand for its (transitive) constituents. Note that the term `(*t).next` above denotes a location that contains a pointer to a `tri`, not a `tri` itself.

modifies [obj] lclType: Suppose we have `modifies T1` where `T1` is an LCL type. A type is viewed as a set of objects. The meaning of the modifies clause is that all instances of `T1` may be modified. That is, we add the set {`x:T1`} to the modified set.

The `obj` qualifier is used to generate the object sort of immutable types. For example, `modifies obj int` adds the set {`x:int_Obj`} to the modified set.

modifies nothing: This assertion constrains the function to preserve the value of all objects accessible to the function. It defines the modified set of the function to be the empty set.

Note that the translation of the modifies clause allows benevolent side-effects on abstract types. The translation only constrains those instances of mutable types that are explicitly present in the given state. If the rep type of the abstract type is not exposed, the instance of the rep type used to represent a mutable type instance does not appear in the state. Hence, while the translation forbids the function from changing the instances of the rep type of the abstract type present *in the state*, it does not prevent benevolent side-effects.

7.4.4 Fresh and Trashed

In an ensures clause, the built-in LCL predicates `fresh` and `trashed` can be used; they are predicates on objects. Their semantics are given below:

$$\begin{aligned}&\forall\ \mathtt{x{:}\ T1\_Obj\ (fresh(x)} \equiv \mathtt{x} \notin \mathtt{domain(pre)} \wedge \mathtt{x} \in \mathtt{domain(post))}\\ \wedge\ &\forall\ \mathtt{x{:}\ T1\_Obj\ (trashed(x)} \equiv \mathtt{x} \in \mathtt{domain(pre)} \wedge \mathtt{x} \notin \mathtt{domain(post))}\end{aligned}$$

7.4.5 The Claims Clause

The claims clause does not add new logical content to the specification. It states a conjecture that is intended to supplement the specification, and to aid in the debugging of the function specification. If the function is named P, the conjecture the claims clause states is:

`specs(P, pre, post) ⇒ ClaimsP`

where `ClaimsP` is the condition given in the claims clause.

7.5 LCL Module

An LCL module, also called an *interface*, has imported LCL modules, explicitly imported LSL traits, and *export* and *private* declarations.

Abstract Syntax

interface ::= {*import* | *use*} * {*export* | *private* } *
import ::= **imports** *id*$^{+}$, ;
use ::= **uses** *traitRef*$^{+}$, ;
export ::= *constDeclaration* | *varDeclaration* | *type* | *fcn*
private ::= **spec** { *constDeclaration* | *varDeclaration* | *type* | *fcn* }
constDeclaration ::= **constant** *lclType* { *id* [= *term*] }$^{+}$, ;
varDeclaration ::= *lclType* { *id* [= *term*] }$^{+}$, ;
traitRef ::= *id* [(*renaming*)]
renaming ::= *replace*$^{+}$, | *typeName*$^{+}$, *replace**,
replace ::= *typeName* **for** *id*
typeName ::= [**obj**] *lclType*

An *export* declaration is either a global constant, a global variable, a type declaration, or a function specification. The *private* declarations are similar, except marked by the prefix keyword `spec`. A variable declaration may include initialization. The symbols in a used trait can be renamed in a similar manner as the renamings of included traits within LSL traits. Unlike LSL trait renamings, operator renamings are not supported at the LCL level.

Checking

- There must be no import cycles. The `imports` clause defines a transitive *importing* relation between module names.
- If a constant or variable is initialized, the sort of the initializing term must be the sort of the constant or variable.
- The number of *typeName*s in a *renaming* must be equal to the number of sort parameters in the used traits. Each *id* in *traitRef* names a used trait.

Meaning

uses: The uses clause associates some LSL traits with the LCL module. The meanings of the operators used in LCL specifications are defined in these traits. The uses clause supports sort renaming so that LCL types may be associated with LSL sorts. This renaming is carried out after an implicit mapping of LCL type names to LSL sort names is done, as explained in Section 7.3.2. An optional `obj` qualifier indicates that the object sort corresponding to a given LCL type is desired.

global constants: An LCL (global) constant is simply a constant in the logical semantics of LCL. Unlike an LSL constant, LCL constants must be implemented so that clients can use them. If an LCL constant is declared with an "initial" value (see *constDeclaration* in the grammar above), it is interpreted to be an equality constraint on the declared constant. This is often used to relate LCL constants to LSL constants.[3]

global variables: An LCL global variable in a module is modeled by an object in a *global state*. The global state is the state that is modified by the functions specified in the module. If a global variable is initialized, then the object is given the appropriate value in the initial global state. Otherwise, the object is allocated but not assigned an initial value. Global variables must be implemented since clients can use imported global variables in their code. If the specification of a global variable has an initial value, then the initialization must occur in the module initialization function.

spec constants, variables, types, and functions: These are private versions of global declarations. They are treated the same as global constants, variables, types and functions, except that they do not have to be implemented and are only accessible in the specifications within the module. They support data and procedure hiding. In addition, each spec variable is modeled as a *virtual object*. The set of all virtual objects in a module is called its *spec state*. We assume that if a spec variable is implemented, the locations representing the spec variable are disjoint from those used to represent all global variables. This is necessary to encapsulate them in the module.

We introduce an operator, *specState: moduleName, state* $\rightarrow$ *objectSet*, to help us model spec states. Each module encapsulates some virtual objects that are different from other modules. The value of *specState(M, st)* is defined to be the set of virtual objects modeling the spec variables declared in module M. We also introduce selection functions named after the spec variables of M for ease of reference. For example, if M contains a spec variable named SV of type T, we introduce the operator *SV: objectSet* $\rightarrow$ *T_Obj* so that we can refer to the object modeling SV using the term *SV(specState(M, st))*. These operators are handy in formulating a module induction principle that can help us deduce inductive properties of the objects in the spec state of a module, see Section 7.5.6.

imports: If a module M1 imports another module M2, then M1 and clients of M1 have access to the types, constants, variables, and functions exported by M2. The traits used by M2 are also part of the specification of M1.

initialization procedures: It is an LCL convention that for each module named `D`, if there is an initialization function for the module, it will be named `D_initMod`. It is an LCL convention that all modules with initialization procedures must be initialized before use, and each module initializes every module it explicitly imports. Since multiple clients may

---

[3] LSL constants are zeroary operators, and like all LSL operators, they do not have to be implemented. In contrast, LCL (non-spec) constants must be implemented.

Definition of contains: Type → AbstractTypeSet	
Type T	*contains(T)* ≡
an abstract type	{T}
int, char, double, enumerated types	{}
a pointer to type T2	contains(T2)
an array of type T2	contains(T2)
a struct or union	$\bigcup_{f \in fields\ of\ T} contains(TypeOf(f))$

Table 7.2: Definition of the contains operator.

initialize the same module in a single program context, module initializations should be idempotent so that their effects do not depend on the number of initialization calls.

The semantics of an LCL module is given by the logical translations of the specifications of all the functions in the module, and two induction principles. First, when a module defines an abstract data type, we have a data type induction principle for predicates on the abstract type. Second, when a module is used to encapsulate private data, we have a module induction principle for predicates on the private data. The two induction principles are orthogonal. If a module defines some abstract types and encapsulates some private data, then the two induction principles can both be applied. These induction principles are discussed further in the sections after the next one.

7.5.1 Type Containment

In order to provide a useful and simple induction rule for reasoning about LCL abstract types, a notion of type containment is needed. We say that a type T *contains* another type T2 if we can reach an instance of T2 from an instance of T by using C built-in operators (without type casting). For example, a struct with a `set *` field contains `set`.

To formalize the concept of type containment, we define an operator *contains*: Type → AbstractTypeSet*, which gives all the abstract types that may be contained in an instance of the type. Note that the range of contains* consists of abstract types because we are only interested in using the type containment information to help us derive data type invariants for abstract types. The operator contains* is defined below:

$$\begin{aligned} contains^*(T) &= \{\}, && \textit{if T is an abstract type} \\ &= contains(T), && \textit{otherwise.} \end{aligned}$$

The auxiliary operator *contains: Type → AbstractTypeSet* is defined in Table 7.2. In the table, the expression *TypeOf(f)* stands for the declared type of the field named *f* in a C struct or union.

For example, suppose we have the following type declarations:

```
mutable type set;
immutable type stack;
typedef struct {int first; set second[10]; stack *third;} triplet;
typedef triplet *tripletPtr;
```

then we have the following:

```
contains*(set) = contains*(stack) = {}
contains*(tripletPtr) = contains*(triplet) = {set, stack}
```

We note that in LCL, by forbidding `imports` cycles, we cannot have two (or more) abstract types defined in separate modules that mutually contain each other. A module, however, can define more than one abstract type. We term them as *jointly defined* abstract types and they are discussed in Section 7.5.5.

7.5.2 A Simple Type Induction Rule

There are three reasons for deriving invariants for an abstract type. First, invariants are useful documentation. They highlight important properties of the type. Second, proving an invariant helps specifiers debug specifications. Third, invariants are useful properties to help clients reason about programs that use the type. To achieve these goals, it is important to have type induction rules that are sound and simple. These two criteria motivate the type induction rules we provide in LCL.

The basic framework for LCL type induction is as follows: Suppose we have an abstract type T defined in module M, and the type invariant we want to prove is P: T, state → Bool. We first derive a type induction rule for a simple case by making the following assumptions:

- No Rep Type Exposure Assumption: An implementation of an abstract type exposes its representation type if it is possible for its client to change the value of an instance of the abstract type without calling the interfaces of the abstract type. We assume that the representation types of abstract LCL types are not exposed. This assumption is essential for all type induction rules and is discussed further in Section 7.6.

- Simple Input-Output Assumption: The functions exported by M contain input arguments and output results that have type T2 such that T2 = T or T $\notin$ contains*(T2).[4] This restriction is employed to simplify the process of finding instances of T in the input arguments and output results of functions in M. We will relax this restriction in Section 7.5.3.

- Simple Invariant Assumption: The predicate P does not depend on the values of non-T objects. For example, the following case is not covered by the simple induction rule: T is the stack type whose instances contain mutable set objects, and P is the predicate: every set object in a stack has size less than ten. We exclude this case in order to focus on the functions exported by M alone. We relax this assumption in Section 7.5.4.

To specify the induction rule more formally, we first define a few terms that describe the functions of an abstract data type T. Given a function F of T, let *ins(F, T)* be the input arguments of F of type T. Let *outs(F, T)* be the output results of F that are of type T.

---

[4] Note that under our definition of output result in Section 7.4.1, an object in the modifies clause of a function is considered to be an output result of the function.

A *simple basic constructor* for an abstract type T is a function with empty *ins(F, T)* and non-empty *outs(F, T)*.

A *simple inductive constructor* for T is a function that has non-empty *ins(F, T)* and non-empty *outs(F, T)*. Simple inductive constructors are functions that take some parameters of type T and produce results of type T. If T is a mutable type, they include mutators that modify some input instances of T.

Let *SBC* be the set of simple basic constructors for T, and *SIC* be the set of simple inductive constructors. We assume that *SBC* is non-empty. If *SBC* is empty, then there is no basis for induction, and the induction rule cannot be applied. The type induction rule for T and predicate P: T, state → Bool is:

$$\begin{array}{lll} (T1) & \forall C : SBC & \mathit{specs}(C, \mathit{pre}, \mathit{post}) \Rightarrow \\ & & \quad (\forall y : T \ y \in \mathit{outs}(C,T) \Rightarrow P(y, \mathit{post})) \\ (T2) & \forall D : SIC & (\mathit{specs}(D, \mathit{pre}, \mathit{post}) \\ & & \quad \wedge \ \forall x : T \ x \in \mathit{ins}(D,T) \Rightarrow P(x, \mathit{pre})) \Rightarrow \\ & & \forall y : T \ y \in \mathit{outs}(D,T) \Rightarrow P(y, \mathit{post}) \\ \hline (T3) & & \forall x : T, st : \mathit{state} \ \mathit{revealed}(x, st) \Rightarrow P(x, st) \end{array}$$

First, the conclusion of the induction rule needs some explanation. We say that an instance of an abstract type T is *revealed* if it can be reached from some initialized program variable in a program context outside the module of T using C built-in operators. For example, in a program context outside the module of T, the instance bound to an initialized program variable of type T and the instance that is pointed to by a program variable of type `T *` are both revealed. We are interested in such revealed values because they are the ones that are accessible by the interfaces exported by T and by other built-in operations (excluding type casting) of C. If an instance of T is hidden in some other abstract type, and is never passed out of the module of T, the type invariant we derived above should not be applied to it. Section 7.5.5 provides a more detailed discussion of revealed and hidden values.

We are mainly interested in reasoning about specifications in some abstract state. To use the induction rule to reason about specifications, we add the following implicit condition to the requires clause of each LCL function specification F:

- `revealed(x, pre)` if `x` is an input argument of F and is an instance of an abstract type.

We also add the following implicit conditions to the ensures clause of F:

- `revealed(x, post)` if `x` is an input argument of F, is an instance of an abstract type, and is not trashed in the specification of F
- `revealed(x, post)` if `x` is an output result of F and is an instance of an abstract type.

We argue that our type induction rule is sound as follows: Instances of an abstract type T can only be manipulated in a program context by calling functions given in the module defining T. Since the only way instances of T can be communicated out of the T module is through the functions of T, we only need to consider the input arguments and output results of the functions of T. The proof obligations for simple basic constructors ensure that all initial revealed T instances satisfy P. The proof obligations for simple inductive constructors ensure that their T output results also satisfy P, given that their input T instances satisfy P. The only other class of functions in the module are the observers of T. Since they do not produce output results of T and they do not modify T instances, they preserve the invariant P.

The soundness of this induction rule depends on the assumption that the rep of an abstract type is not exposed. If the rep is exposed, then an abstract object can be modified directly without going through the functions of the type. The key benefit of data induction is gained by restricting the access to the rep of an abstract type so that properties deduced locally in a type can be applied in arbitrary client contexts. If reps are exposed, local properties need not hold in general since arbitrary changes could be made independent of the type interfaces.

7.5.3 A Second Type Induction Rule

In this subsection, we relax the simple input-output assumption in which the functions exported by an abstract type can only contain input arguments and output results of type T, or have types that do not contain T. There are situations where the exported functions of T take or return pointers to T instead of T instances directly. Since C does not support multiple argument returns or call by reference, it is a C idiom to return values indirectly via input pointers. An example is given in Figure 7-6.

```
immutable type set;
uses Set (int, set);
bool set_init (int i, out set *s) {
  modifies *s;
  ensures result = (i = 0) ∧ if result then (*s)′ = {1} else unchanged(*s);
}
```

Figure 7-6: C idiom: returning a result via an input pointer.

The `out` type qualifier in the specification is useful in two ways. First, it highlights the C idiom being used. Second, it indicates to the implementor that the value of `(*s)`^ may not be initialized. The term `*s` stands for a C location that contains a `set` instance, or a *set location* for short. Since the set location may not be initialized, we should treat it as an output argument, like `result`, and for the purpose of data type induction, we should not assume the invariant on `(*s)`^.

We extend the type induction rule in the last subsection to handle one-level pointers to abstract types. There are two reasons why we choose to handle only one-level pointers rather than arbitrary types that may contain abstract types. First, the induction rule for handling arbitrary types is more complicated to use. Second, it is desirable to have simpler interfaces for abstract types. If an abstract type T exports functions that deal with complicated data structures that contain instances of T, it is often a sign that the design is not as modular as it could be. We consider it desirable for abstract types to have simple interfaces. We believe that there are few situations where complicated interfaces are needed for the design of abstract types.

We assume that the second induction rule will only be applied to an abstract type whose functions respect the following condition: every input argument or output result of the function has type T2 such that T2 = T, or T2 is the C location type containing T (`T_Obj`), or T ∉ contains*(T2). From Section 7.5.1, we know that we can compute contains* easily.

Now, we give the second type induction rule. Given a function F of T, let _ins*(F, T)_ be the input arguments of F of type T, or are T locations which are non-`out`. Let _outs*(F, T)_ be the output results of F of type T, or are T locations.

A _basic constructor_ for an abstract type T is a function with empty _ins*(F, T)_ and non-empty _outs*(F, T)_. An _inductive constructor_ for T is a function that has non-empty _ins*(F, T)_ and non-empty _outs*(F, T)_.

Let _BC_ be the set of basic constructors for T, and _IC_ be the set of inductive constructors. As before, we assume that _BC_ is non-empty. The type induction rule for T and predicate P: T, state → Bool is:

$$
\begin{array}{lll}
(U1) & \forall C : BC & \mathit{specs}(C, \mathit{pre}, \mathit{post}) \Rightarrow \\
 & & \quad ((\forall y : T \;\; y \in \mathit{outs}^*(C,T) \Rightarrow P(y, \mathit{post})) \\
 & & \quad\quad \wedge\, \forall y : T\_Obj \;\; y \in \mathit{outs}^*(C,T) \Rightarrow P(y', \mathit{post})) \\
(U2) & \forall D : IC & (\mathit{specs}(D, \mathit{pre}, \mathit{post}) \\
 & & \quad \wedge\, (\forall x : T \;\; x \in \mathit{ins}^*(D,T) \Rightarrow P(x, \mathit{pre})) \\
 & & \quad \wedge\, \forall x : T\_Obj \;\; x \in \mathit{ins}^*(D,T) \Rightarrow P(x^{\wedge}, \mathit{pre})) \Rightarrow \\
 & & ((\forall y : T \;\; y \in \mathit{outs}^*(D,T) \Rightarrow P(y, \mathit{post})) \\
 & & \quad \wedge\, \forall y : T\_Obj \;\; y \in \mathit{outs}^*(D,T) \Rightarrow P(y', \mathit{post})) \\
\hline
(U3) & & \forall x : T, st : \mathit{state} \;\; \mathit{revealed}(x, st) \Rightarrow P(x, st)
\end{array}
$$

The new rule is similar to the previous one. The only difference lies in the new way in which T instances may be passed into and out of the functions of T, via pointers. The new way is accounted for in the induction rule by checking that if an instance of T is passed out of the functions of T via a pointer, then the instance satisfies the predicate P.

To use this more general rule for reasoning about specifications, we add the following implicit conditions to the requires clause of the specification of each function F, in addition to those given in Section 7.5.2:

- `revealed((*x)^, pre)` if `x` is an input argument of F and is a non-`out` pointer to an abstract type.

We also add the following implicit conditions to the ensures clause of F, in addition to those given in Section 7.5.2:

- `revealed((*x)', post)` if `x` is an input argument of F, is a non-`out` pointer to an abstract type, and `*x` is not trashed in the specification of F.
- `revealed((*x)', post)` if `x` is an output result of F and is a pointer to an abstract type.

Note that the second induction rule also makes the simple invariant assumption in which the invariant we are trying to show does not depend on the value of non-T objects.

7.5.4 A Third Type Induction Rule

An instance of an abstract type T can have objects of another type T2 as its value. In such a case, an invariant of type T may depend on the value of T2 objects in some state. Hence, a mutator of the T2 type can affect the truth of the invariant. For example, suppose we have an immutable type stack containing mutable sets and the stack invariant that each set object in a stack contains only strictly positive integers. A set insert may destroy the invariant if it is allowed to insert zero or negative integers.

The next induction rule we provide handles the above problem and thus discharges the simple invariant assumption. Recall from Section 7.5.2 that the simple invariant assumption states that an invariant of type T does not depend on the values of non-T objects. Our induction rule discharges the assumption in one specific way: it allows the invariant P to depend on the value of objects that are mutable abstract types. It, however, assumes that P cannot depend on the value of locations that contain LCL exposed types. For example, we can define an abstract type, set of `int *`, but since exposed types have no interface boundaries, no sound induction rule is possible for such cases.

First, we define a *mutator* for a mutable abstract type T to be a function in module T whose modified set has at least one object of type T.

If invariant P contains a term `m#st` where `m` denotes an instance of a mutable type T2 different from T, and `st` is a state variable, we define a *mutator hypothesis* of P and T2 as the following proof obligation:[5]

$$(MH) \quad \forall D : MC(T2) \quad (spec(D, pre, post) \wedge \forall x : T \; P(x, pre)) \Rightarrow \forall y : T \; P(y, post)$$

where *MC(T2)* is the set of mutators of T2.

The third induction rule is the same as the last induction rule we gave in Section 7.5.3, except that we add a new class of hypothesis to the induction rule in Section 7.5.3: the mutator hypotheses of P and T2, for all such terms `m#st` in P.

To ensure that we can mechanically get hold of all instances of `m#st`, we require that the only way the state variable `st` appears in P is as `m#st`. This restriction makes it easier for us to mechanically collect all the required mutator hypotheses. For example, if the state variable `st` is passed as an argument to an LSL operator `K` (not equal to `#`) that is defined in a trait, it is not clear how to systematically collect all objects whose values `K` might look up in `st` from the definition of `K`.

Our soundness argument for the third rule rests on the soundness of the second rule and the following observation. Consider an instance of T, `x`, and the value of the term `P(x, pre)`. The only way the value of the term can change when a function is invoked is if P depends on some object in `x` that gets modified by the function. Since we restrict our attention to terms that denote objects of mutable types, we only need to check that every possible mutator of such objects preserves the invariant. This is the statement of the above proof obligation *MH*. All other functions cannot possibly modify such objects, for if they did, they would be mutators of some abstract type, and would be covered by *MH*.

It is instructive to consider how the *MH* hypothesis can be proved in an example. Suppose we have the following set and stack interfaces shown in Figure 7-7 and Figure 7-8.

---

[5] Recall from Section 7.2.1 that the infix operator `#` returns the value of an object in a state.

Note that in Figure 7-8, the `push` function pushes a `mset` *object* onto the stack, not its value. The traits `Stack` and `Set` are Larch handbook traits [15] which define familiar set and stack operators.

```
mutable type mset;
uses Set (mset for C, int for E);
mset create (void) {
  ensures result′ = {} ∧ fresh(result);
}
bool member (mset s, int i) {
  ensures result = i ∈ s^;
}
void insert (mset s, int i) {
  requires i > 0;
  modifies s;
  ensures s′ = insert(i, s^);
}
void delete (mset s, int i) {
  modifies s;
  ensures s′ = delete(i, s^);
}
```

Figure 7-7: mset.lcl

```
imports mset;
immutable type stack;
uses Stack (obj mset for E, stack for C);
stack stackCreate (void) {
  ensures result = empty;
}
stack push (mset s, stack stk) {
  ensures if ∀ i: int (i ∈ s^ ⇒ i > 0)
          then result = push(s, stk) else result = stk;
}
```

Figure 7-8: stack.lcl

Suppose the invariant we want to prove for stacks is

$$P(stk, st) \equiv \forall so : set\_Obj \ (so \in stk \Rightarrow \forall i : int \ (i \in so\#st \Rightarrow i > 0))$$

The invariant says that a set object in a stack has strictly positive members. The specifications in the stack module allow us to discharge the *U1* and *U2* proof obligations of Section 7.5.3 easily. The invariant, however, also depends on the value of `mset` objects that have been pushed onto stacks, but may still be modified by `mset`'s `insert` and `delete` function calls. The mutator hypothesis *MH* for P is designed to check that they also respect P. The mutator hypothesis for P and `mset` is given in Figure 7-9.

We will sketch how *MHa* can be discharged. Consider a stack `stk` before the execution of `insert`; it obeys the stack invariant by assumption. Now, `insert` modifies a set, `s`, which

$$\begin{array}{ll} (MHa) & (specs(insert, pre, post) \wedge \forall x : stack\ P(x, pre)) \Rightarrow \forall y : stack\ P(y, post) \\ (MHb) & (specs(delete, pre, post) \wedge \forall x : stack\ P(x, pre)) \Rightarrow \forall y : stack\ P(y, post) \end{array}$$

Figure 7-9: A mutator hypothesis.

may or may not be in `stk`. There are two cases here. If `s` $\in$ `stk`, we use the specification of `insert` to make sure that the invariant still holds for `stk` in the post state. In this case, it does, since `insert` only inserts elements that are strictly positive. If the requires clause fails, then the function does not terminate, and the conclusion is trivially true. The second case is for `s` $\notin$ `stk`, where the invariant from the pre state can be invoked on `stk`.

7.5.5 Jointly Defined Abstract Types

The induction rules we have given above are also valid for jointly defined abstract types. In many ways, jointly defined types behave much the same way as distinct abstract types defined in separate modules. The only extra freedom two jointly defined abstract types T1 and T2 enjoy is that they have access to each other's rep type. The specification of object modification, however, is independent of where the abstract types are defined. If a function F in the module modifies an abstract object, `x` of type T1, the object must be listed in the modifies clause, even if `x` is contained in some instance of T2. Therefore, the freedom to access each other's rep does not affect the induction rule.

We consider one example that illustrates how our concept of revealed values avoids a source of potential unsoundness in the rules we have given. Suppose the types `stack` and `mset` are jointly defined in a single module as shown in Figure 7-10. The specifications of the functions in Figure 7-10 are identical to those in Figure 7-7 and Figure 7-8 except that in place of `push`, we have `funnyPush`, which pushes a new set object with value {`0`} onto the stack.

Instead of proving the stack invariant we give in Section 7.5.4, suppose we want to show an invariant about `mset`'s. Suppose the invariant is: P(s) $\equiv$ $\forall$ i: int (i $\in$ s $\Rightarrow$ i > 0). The functions in the `mset` module easily met the invariant. Using the induction rule given in Section 7.5.4, no other hypotheses need to be checked for this invariant since the two stack functions do not modify `mset`'s and there are no output results that are `mset`'s. The `funnyPush` function, however, creates a stack that contains an `mset` object that does not satisfy the invariant. This is not a problem in our induction rule because the conclusion is still sound: the `mset`'s revealed via the interfaces of the module satisfy the invariant. There are no interfaces in the module that reveal those `mset`'s hidden under stacks.

Consider, however, a variant of the above module in which an additional stack function, `top`, is exported. The specification of `top` is given in Figure 7-11. In this case, the hidden `mset`'s in a stack are revealed by `top`. Since `top` returns an output result that is an `mset`, `top` is considered to be a `mset` constructor, and it contributes a proof obligation to the proof of the invariant P. Its proof will fail because the supporting lemma that the top of any non-empty stack contains positive integers is false.

```
mutable type mset;
immutable type stack;
uses Set (mset for C, int for E), Stack (stack for C, obj mset for E);
mset create (void) {
  ensures result′ = {} ∧ fresh(result);
}
bool member (mset s, int i) {
  ensures result = i ∈ s^;
}
void insert (mset s, int i) {
  requires i > 0;
  modifies s;
  ensures s′ = insert(i, s^);
}
void delete (mset s, int i) {
  modifies s;
  ensures s′ = delete(i, s^);
}
stack stackCreate (void) {
  ensures result = empty;
}
stack funnyPush (stack stk) {
  ensures ∃ so: obj mset (result = push(so, stk) ∧ fresh(so) ∧ so′ = {0});
}
```

Figure 7-10: setAndStack.lcl

```
mset top (stack stk) {
  requires not(stk = empty);
  ensures result = top(stk);
}
```

Figure 7-11: Adding the top function to setAndStack.lcl

7.5.6 Module Induction Principle

The module induction rule can be viewed as a special case of the type induction rules we have given in the previous sections, where the invariant is independent of the abstract type exported by module D. Below, we give the module induction rule because it is simpler than the other rules.

An LCL module supports data hiding using spec variables. An induction principle for predicates on the virtual object set of a module can be derived using computational induction. Suppose we have a module named D, and *Exports* is the set of functions of D, excluding its initialization function. Suppose the invariant we want to prove is P0 such that P0: objectSet, state → Bool, where the first argument is a subset of the spec state of D. We define an auxiliary operator P: state → Bool to be P(st) ≡ P0(specState(D, st), st). The operator P merges the two arguments of P0 into one so we can apply the following module induction rule.

$$\begin{array}{ll}
(M1) & specs(D\_initMod, pre, post) \Rightarrow P(post) \\
(M2) & \forall F : Exports\ (specs(F, pre, post) \wedge P(pre)) \Rightarrow P(post) \\
\hline
(M3) & \forall st : state\ initializedState(D, st) \Rightarrow P(st)
\end{array}$$

In the conclusion of the rule, the term `initializedState(D, st)` is true if `st` is a state in which the module initialization function `D_initMod` of D has already been called. The predicate is false before `D_initMod` is called, and remains true after it has been called.

The induction rule says that once the invariant P is established over the virtual objects of a module by the initialization function, it is preserved by all other functions of the module. The soundness of this rule rests on the assumption that the spec state of a module can only be modified by the functions of the module.

7.5.7 Module Claims

A module claim does not add new logical content to an LCL specification. It states a conjecture that the condition given in the module claim is an invariant of the module. A module claim is proved using one of our type induction rules given in the previous subsections. If the invariant is independent of the abstract type exported by the module, the simpler module induction rule in Section 7.5.6 can be used in its place. In general, a module claim may involve both an abstract type T and some spec variable SV of type T2 in the module defining T; for example, the predicate corresponding to the module claim may have the form of P0: T, T2_Obj, state → Bool. In such a case, we express P0 using a corresponding predicate P: T, state → Bool as P(x, st) ≡ P0(x, SV(specState(D, st)), st). The form of P is suitable for use in our type induction rules.

Consider the `amountConsistency` module claim we have seen in Chapter 5, which is reproduced below:

```
claims amountConsistency (position p) bool seenError; {
  ensures ¬ (seenError~) ⇒ p~.amt = sum_amt(p~.openLots);
}
```

The predicate corresponding to the above module claim, P0, is

```
∀ p: position_Obj, st: state, seenError: bool_Obj
 P0(p, seenError, st) ≡ (¬ (seenError#st) ⇒
                          ((p#st).amt = sum_amt((p#st).openLots)))
```

The sorts given to the parameters of the module claim need some explanation. Recall from Section 4.9 that `position` is a mutable abstract type and that an instance of a mutable type is modeled by the object sort of the type. Hence, the parameter `p` in the module claim is given the object sort of the position type, `position_Obj`. Next, recall from the description of spec variables in Section 7.5 that spec variables are modeled as virtual objects, the parameter `seenError` is hence given the sort of a `bool` location, `bool_Obj`.

To use the type induction rules given in the previous subsections, we express P0 as follows:

```
  P(p, st) = P0(p, seenError(specState(position, st)), st)
```

where `seenError(specState(position, st))` is the object modeling the spec variable, `seenError`, in the `position` module.

7.6 Type Safety of Abstract Types

An implementation of an abstract type exposes its representation type if the value of an instance of the abstract type can be changed by a client of the abstract type without calling the interfaces of the type. Whether an implementation of an abstract type exposes its rep type or not is an implementation property, not a specification property. As such, the notion of rep type exposure can only be defined formally with respect to concrete program states, which is beyond the scope of this work. Since our goal is focused on reasoning about specifications, we discuss the problem informally and describe the typical ways in which an implementation violates type safety in this section. Vandevoorde [36] gives a more detailed and formal discussion of the type safety problem.

We illustrate the type safety problem with an example. Suppose we implement a mutable abstract type, set, using an `int` array in C, and that there is an exported function in the set type, `intArray2set`, which takes an `int` array and returns a set containing the elements in the input array. An implementation of `intArray2set` that returns the input array directly exposes the rep of the set type. This is because a caller of `intArray2set` may still hold a reference to the input array, for example, in the following program context:

```
int a[10];
set s;
s = intArray2set(a);
```

In the program state after the execution of `intArray2set`, both `s` and `a` point to the same object, that is, there is an alias across types. If the array `a` is modified by some array operation after this point, the change will also appear in `s`. This violates the central requirement of an abstract type: that the only way to change an abstract instance is through the interfaces of its type. Since our data type induction principle depends on this requirement, its soundness is compromised.

Note that we restrict our attention to program contexts that are outside of the implementation of abstract types. Within the implementation of an abstract type, type aliasing is fundamental to data abstraction and cannot be avoided.

Another way type safety can be violated is when an object of a rep type is cast into an abstract object. This allows a rep instance to masquerade as an abstract instance. Since a rep object need not obey the invariants maintained by an abstract type, the soundness of the type induction principle can be broken. One design goal of LCL is to add a stronger typing discipline to C programs. The LCLint tool checks that no rep objects are explicitly cast from and into abstract ones. It provides guarantees similar to that of the CLU compiler [26].

7.7 Summary

We have formalized several informal concepts introduced in the previous chapters and described interesting aspects of the semantics of LCL. Our semantics is designed primarily for reasoning about LCL specifications and claims.

The domain of LCL values consists of basic values and objects. Objects are containers of values. A state is a mapping of objects to values. LCL exposed types are modeled according to the semantics of C's built-in types. LCL immutable types are modeled by basic values, and LCL mutable types are modeled by objects. We showed how LCL types are implicitly mapped to LSL sorts. Each LCL specification makes use of LSL traits which introduce sorts and operators, and provide axioms that constrain the operators.

An LCL function specification is a predicate on two states, the pre state and the post state. This predicate constrains the input-output behavior of the function. The checks clause and exposed types with constraints play a role in translating a function specification into its corresponding predicate.

An LCL module can specify abstract types and encapsulate data. We provided induction rules for deriving inductive properties of abstract types and invariants of encapsulated data. Our induction rules relied on a notion of *revealed values*, which are instances of abstract types made accessible through the interfaces of the types. This is different from a previous approach [38] that relied on fresh objects.

Chapter 8

Further Work and Summary

In this chapter, we suggest areas where further work might be useful and summarize the thesis.

8.1 Further Work

There are four directions to further our work. We suggest more specifications checks, a few plausible code checks, directions to extend the LCL specification language to specify a wider class of programs, and further experimentation on using formal specifications to reengineer existing programs.

8.1.1 More Checks on Specifications

Our work on checking formal specifications has two components: a syntactic component and a semantic component. The LSL and LCL checkers perform syntactic and type checking on specifications. They help catch many careless errors, and improve the quality of the specification. The semantic component is to analyze the specification by proving claims. Below we suggest other useful checks in the two components.

More Expressive Claims: Our work focuses only on a few kinds of claims. They were chosen for simplicity and utility. A module claim asserts properties about a single state. It is useful to explore how claims that assert properties across several states can be proved and used. For example, such claims can be used to sketch the intended prototypical usage of the procedures in a module. Sometimes, the procedures in a module are designed to be used in a cooperative manner. For example, since C does not support iterators, multiple procedures are needed to support iteration abstraction in C. A more expressive claim language would allow the relationships between the iteration procedures to be stated.

In the extreme, the claim language may be the C language itself. This will allow full-fledged verification of C programs. On the other hand, it may be useful to explore a less expressive language that still adequately expresses many relationships that specifiers desire. For example, the claim language could be a trace-like language [2, 18] in which properties of various combinations of procedure calls can be stated and checked. Procedure calls could be combined using simple regular expressions for expressing procedure call sequencing and loops [35].

Efficient Specification Checks Enabled by Conventions: Because semantic analysis of formal specifications is expensive, a more efficient means is desirable. Syntactic checks are often efficient and can detect specification errors. There are, however, few sound syntactic checks. A sound check is one that does not produce spurious error messages. One kind of sound check is type checking of terms in Larch specifications.

Like the lint approach to checking programs, it is useful to devise efficient but not necessarily sound checks on specifications. We can rely on specification conventions to aid in the checking. While this approach can produce spurious error messages, it can efficiently catch a number of careless mistakes beyond type-checking. The following are some examples of syntactic checks on LCL function specifications.

First, if a result of a function is non-`void`, the specification must say what the result should be. We can syntactically detect if the result appears in the ensures clause. We can extend the check to handle conditionals: the results of a function must be *set* in every branch of the conditional.

Second, if both `trashed(x)` and `x`$'$ appear in a function specification, it is likely to be an error.

Third, if the post state value of an object is referred to, the object must be in the modifies clause. The check can detect a common error where the specifier inadvertently leaves out the changed object in the modifies clause. The check is not useful unless the specifier adopts the convention of using `x^` (rather than `x`$'$) to refer to the value of an object that is not changed by the procedure.

8.1.2 More Checks on Implementations

A formal specification contains information that can be used to check its implementation. Much of the information, however, is not easily extracted or used. This prompts research to simplify the specification language so that more useful checks can be performed on the implementation of a specification. For example, a user of the Aspect system [19] specifies dependencies among selected aspects of data items, and the system uses efficient data flow analysis to check an implementation of the specification for missing dependencies. The advantage of Aspect is that checks are efficient and specifications are simpler to write. This, however, comes at the expense of good documentation. Dependencies do not capture much of the intended behavior or design of procedures and modules.

Combining Aspect techniques with Larch specifications is desirable. It will reap the benefits of both: efficient analysis allows more program errors to be found, and complete behavioral description supports program documentation. We give an example of how the two can be combined. The checks clause of LCL is designed with a specific purpose in mind: the implementor of a function specification with a checks clause should ensure that the condition specified in the checks clause holds. The condition is often simple and involves some input arguments of the function. A kind of information that can be abstracted out of the condition is the set of input objects that are mentioned in the condition. If an implementation of the function never refers to an input argument in the set, it is likely that the check has been unintentionally omitted.

8.1.3 LCL and Larch Extensions

LCL can currently be used to specify a large class of C programs. It does not, however, support the specification of functions that take in procedure parameters. While C does not support type parameterization, many specifications are naturally parameterized by types. Supporting the new features can also suggest new kinds of useful claims to be specified and checked.

Parameterized Modules: Type parameterization is a complementary technique to procedural and data abstractions. Many programs and specifications are identical except for the types of arguments they take. A natural class of parameterized functions are functions that operate on sequences of data, but are independent of the types of data in the sequences.

LCL specifiers can benefit by specifying a common interface that is parameterized by types. Since C does not support type parameterization, a specifier can instantiate type parameters with ground types to generate a non-parameterized module which can then be used in the same way as other non-parameterized modules.

Procedure Parameters: Even though procedure parameters are not essential to expressing computation, they are useful for capturing common processing patterns that can lead to simpler, more modular, and more concise code. Procedures that take other procedures as parameters are often termed *higher-order*. Specification of higher-order functions is an area of research that has not yet been successfully addressed in Larch. Since C supports a very limited form of higher-order functions, a better avenue to study the problem in its fullness may be in an interface language for a programming language that supports higher-order functions, such as in Larch/ML [40].

8.1.4 Reengineering Case Studies

Our reengineering exercise revealed some beneficial effects of using LCL specification to help reengineer existing programs. We have not, however, done careful control experiments to measure its detailed effects. For example, we observed that using abstract types in the new PM program increased the size of the source code. It is difficult to study the actual increase in code size due solely to the use of abstract types because we made other changes to the program. For example, we added new checks on the inputs of the program, and checks needed to weaken the preconditions of some functions. Careful control experiments are needed to measure the effects of the various code changes.

8.2 Summary

Our work is motivated by the difficulty of developing, maintaining, and reusing software. We believe that through the design of more modular software, and by improving the quality of software documentation, we can alleviate the problem. In this thesis, we presented techniques for encouraging software modularity, improving software documentation, and testing specifications.

In Chapters 2 and 3, we presented a novel use of formal specification to promote a programming style based on specified interfaces and data abstraction in a programming language that lacks such supports. The Larch/C Interface Language (LCL) is a language for documenting the interfaces of ANSI C program modules. Even though C does not support abstract data types, LCL supports the specifications of abstract data types, and provides guidelines on how abstract types can be implemented in C. A lint-like program checks for some conformance of C code to its LCL specification [5].

Within the framework of LCL, we introduced the concept of *claims*, logically redundant, problem-specific information about a specification. In Chapter 5, we illustrated how claims can enhance the role of specifications as a software documentation tool. Claims are used to highlight important or unusual specification properties, promote design coherence of modules, and support program reasoning. In addition, we also showed how claims about a specification can be used to test the specification. We took the approach of proving that claims follow semantically from the specification. We provided a semantics of LCL suitable for reasoning about claims in Chapter 7.

In Chapter 4, we described the LCL specifications of the main modules of an existing 1800-line C program, named PM. The main ideas of this thesis are exercised in the specification case study. The case study showed that LCL is adequate for specifying a class of medium-sized C programs. It also motivated techniques for writing more compact and easier to understand specifications. Some LCL features were introduced to codify some of these techniques.

We verified parts of the claims in the case study using the LP proof checker [7] and, in the process, learned about some common classes of specification errors that claims can help catch. Formally verifying claims with a proof checker was tedious and difficult. In return, however, the process helped us gain a better understanding of our specification. Our experience verifying claims suggests that specification errors cannot be easily found without meticulous proof efforts. While a proof checker is not essential, it is a big bookkeeping aid in the verification process. In particular, verification with the help of a proof checker reduces the effort needed to re-check claims. We described the features a proof checker suitable for claims verification should provide in Chapter 5.

In Chapter 6, we gave a software reengineering process model for improving existing programs. The process is aimed at making existing programs easier to maintain and reuse while keeping their essential functionalities unchanged. Our process model is distinguished by the central role formal specifications play in driving code improvement. We described the effects of applying the process to the PM program using LCL.

Besides the new specification product, the specification process improved the modularity of the program, helped to uncover some new abstractions, and contributed to a more coherent module design. In addition, the process made the program more robust by removing some potential errors in the program. The service provided by the reengineered program also improved because the process helped us identify new useful checks on the user's inputs to the program. We have achieved these effects without changing the essential functionality or performance of the program.

We found tool support to be indispensable in writing formal specifications. We used tools to check the syntax and static semantics of our specifications, to check aspects of consistency between a specification and its implementation, to translate some of our specifications into inputs suitable for a proof checker, and to verify claims. These tools helped to uncover both simple and subtle errors. Our experience argues for the use of formal description techniques rather than informal ones because we can build better tools to support formal techniques.

Appendix A

LCL Reference Grammar

Interfaces

interface ::= {*import* | *use*}* {*export* | *private* | *claim* }*
import ::= **imports** *(* *id* | " *id* " | **<** *id* **>** *)*$^+$, ;
use ::= **uses** *traitRef*$^+$, ;
export ::= *constDeclaration* | *varDeclaration* | *type* | *fcn* | *claim*
private ::= **spec** { *constDeclaration* |*varDeclaration* | *type* | *fcn* }
constDeclaration ::= **constant** *typeSpecifier* { *varId* *[* = *term* *]* }$^+$, ;
varDeclaration ::= { *[* **const** | **volatile** *]* } *lclTypeSpec* { *declarator* *[* = *term* *]* }$^+$, ;
traitRef ::= *id* *[* (*renaming*) *]*
renaming ::= *replace*$^+$, | *typeName*$^+$, *replace**,
replace ::= *typeName* **for** { *opId* *[***:** *sortId**, *mapSym* *sortId* *]* | *CType* }
typeName ::= *[* **obj** *]* *lclTypeSpec* *[abstDeclarator]*

Functions

fcn ::= *lclTypeSpec* *declarator* { *global* }* { *fcnBody* }
global ::= *lclTypeSpec* *declarator*$^+$, ;
fcnBody ::= *[* *letDecl* *]* *[* *checks* *]* *[* *requires* *]* *[* *modify* *]*
[*ensures* *]* *[* *claims* *]*
letDecl ::= **let** { *varId* *[* **:** *sortSpec* *]* **be** *term* }$^+$, ;
sortSpec ::= *lclTypeSpec*
requires ::= **requires** *lclPredicate* ;
checks ::= **checks** *lclPredicate* ;
modify ::= **modifies** { **nothing** | *storeRef*$^+$, } ;
storeRef ::= *term* | *[* **obj** *]* *lclTypeSpec* * *
ensures ::= **ensures** *lclPredicate* ;
claims ::= **claims** *lclPredicate* ;

Types

type ::= *abstract* | *exposed*

abstract ::= [**mutable** | **immutable**] **type** *id* ;

exposed ::= **typedef** *lclTypeSpec* { *declarator* [{ *constraint* }] }$^{+}$, ;

| { **struct** | **union** } *id* ;

constraint ::= **constraint** *quantifierSym* *id* **:** *id* (*lclPredicate*) ;

lclTypeSpec ::= *typeSpecifier* | *structSpec* | *enumSpec*

structSpec ::= [**struct** | **union**] [*id*] { *structDecl*$^{+}$ }

| [**struct** | **union**] *id*

structDecl ::= *lclTypeSpec* *declarator*$^{+}$, ;

enumSpec ::= **enum** [*id*] { *id*$^{+}$, } | **enum** *id*

typeSpecifier ::= *id* | *CType*$^{+}$

CType ::= **void** | **char** | **double** | **float** | **int**

| **long** | **short** | **signed** | **unsigned**

absDeclarator ::= (*abstDeclarator*)

| * [*abstDeclarator*]

| [*abstDeclarator*] *arrayQual*

| *abstDeclarator* ( )

| [*abstDeclarator*] (*param**,)

param ::= [**out**] *lclTypeSpec* *parameterDecl*

| [**out**] *lclTypeSpec* *declarator*

| [**out**] *lclTypeSpec* [*abstDeclarator*]

declarator ::= *varId* | * *declarator*

| (*declarator*)

| *declarator* *arrayQual* | *declarator* (*param**,)

parameterDecl ::= *varId* | * *parameterDecl*

| *parameterDecl* *arrayQual* | *parameterDecl* (*param**,)

arrayQual ::= [[*term*]]

Predicates

lclPredicate ::= *term*

term ::= **if** *term* **then** *term* **else** *term* | *equalityTerm*

| *term* *logicalOp* *term*

equalityTerm ::= *simpleOpTerm* [{ *eqOp* | = } *simpleOpTerm*]

| *quantifier+* (*term*)

simpleOpTerm ::= *simpleOp2*$^{+}$ *secondary* | *secondary* *simpleOp2*$^{+}$

| *secondary* { *simpleOp2* *secondary* }*

simpleOp2 ::= *simpleOp* | *

secondary ::= *primary* | [*primary*] *bracketed* [**:** *sortId*] [*primary*]

| *sqBracketed* [**:** *sortId*] [*primary*]

bracketed ::= *open* [*term* { { *sepSym* | , } *term* }*] *close*

sqBracketed ::= [[*term* { { *sepSym* | , } *term* }*]]

open ::= { | *openSym*

close ::= } | *closeSym*
primary ::= (*term*) | *varId* | *opId* (*term*$^+$,) | *lclPrimary*
| *primary* { *preSym* | *postSym* | *anySym* }
| *primary* { *selectSym* | *mapSym* } *id*
| *primary* [*term*$^+$,]
| *primary* **:** *sortId*
lclPrimary ::= *cLiteral* | **result** | **fresh** (*term*)
| **trashed** (*storeRef*)
| **unchanged** ({ **all** | *storeRef*$^+$, })
| **sizeof** ({ *lclTypeSpec* | *term* })
| **minIndex** (*term*)
| **maxIndex** (*term*)
| **isSub** (*term* , *term*)
cLiteral ::= *intLiteral* | *stringLiteral* | *singleQuoteLiteral* | *floatLiteral*
quantifier ::= *quantifierSym* { *varId* **:** *[* **obj** *]* *sortSpec* }$^+$,
varId ::= *id*
fcnId ::= *id*
sortId ::= *id*
opId ::= *id*

Claims

claim ::= **claims** *id* (*param**,) { *global* }*
{ *[letDecl] [requires] [body] ensures* }
| **claims** *fcnId id* **;**
body ::= **body** { *fcnId* (*value**,) **;** }
value ::= *cLiteral* | *varId* | (*value*)
| *[value] simpleOp [value]* | *fcnId* (*value**,)

Appendix B

Relating LCL Types and LSL Sorts

This appendix supplements the description in Chapter 7. The following sections describe how LCL exposed types are modeled by LSL sorts, and how different kinds of LCL variables are given appropriate LSL sorts.

B.1 Modeling LCL Exposed Types with LSL Sorts

In this section, the sorts used to model C built-in types are described. For each of the exposed types, we define what its *value sort* and its *object sort* are.

Primitive types of C: The primitive types of C are modeled as if they are immutable abstract types; there is one identically named LSL sort to model each of them. It is also defined to be the *value sort* of the type. For instance, the sort `int` models the `int` type. The differences between the type compatibility rules of C and LCL described in Section 7.3.1 show up here. For example, LCL considers `int`, `char`, and enumerated types as different types. Hence, different sorts are generated for them. Since C type qualifiers are not significant, they are dropped in the mapping process. Finally, C `float` and `double` types are both mapped to the `double` sort.

It is useful to define the object sort of C primitive types and LCL immutable types. They are used to model memory locations that can contain such values. Such locations arise when pointers to them are dereferenced by the * operator. Their object sorts allow us to describe properties about these locations. If T is a C primitive type or an immutable abstract type, then we define the sort `T_Obj` to be the object sort of T.

Pointer to type T: The type T is first mapped to its underlying sort, suppose it is `TS`. Two LSL sorts are used to model a pointer type. The `TS_ObjPtr` models the pointer that is passed to and from procedure calls, and the `TS_Obj` sort models the object that is obtained when the pointer is dereferenced by the LCL operator *. `TS_ObjPtr` is called a *pointer sort*, and is defined to be the value sort of the pointer type. Finally, when an object of sort `TS_Obj` (an object sort) is evaluated in a state, a value of sort `TS` (a value sort) is obtained. The object sort of the type `T *` is the `TS_ObjPtr_Obj` sort.

C does not make any distinction between arrays and pointers. LCL views them as distinct types.

Array of element type T: The type T is first mapped to its underlying sort; suppose it is `TS`. Two LSL sorts are used to model an array type. The `TS_Arr` sort models the array object, and it is the object sort of the array type. The `TS_Vec` sort models the value of the array in a state, and it is the value sort of the array type. `TS_Arr` is called an *array sort*, and `TS_Vec`, a *vector sort*.

Struct type: Suppose we have the following C struct: `struct _S { ... }`. Two uniquely generated LSL sorts are used to model a struct type. The `_S_Struct` sort, called a *struct sort*, models an object of the struct type; it is the object sort of the struct type. The `_S_Tuple` sort, called a *tuple sort*, models the value of a struct object in a state; it is the value sort of the struct type.

Union type: The mapping for union types is analogous to that for struct types. Suppose we have the following C union: `union _U { ... }`. The `_U_Union` sort, called a *union sort*, models an object of the union type; it is the object sort of the union type. The `_U_UnionVal` sort, called a *union value sort*, models the value of a union object in a state; it is the object sort of the union type.

Enumerated type: Unlike C, LCL views enumerated types as new types, different from `int`, `char`, and other separately generated enumerated types. They are modeled as immutable types. Suppose we have: `enum _E { ... }`. The unique corresponding LSL sort is `_E_Enum`; it is called an *enumeration sort*. It is the value sort of the enumerated type. As for other primitive C types or immutable types, the object sort of the `enum _E` type is the `_E_Enum_Obj` sort.

The object sorts of LCL types are *mutable* sorts. Instances of these sorts represent mutable objects. Other kinds of sorts are *immutable* sorts.

B.2 Assigning LSL Sorts to LCL Variables

An LCL variable is either a global variable, a formal parameter, or a quantified variable. A spec variable is considered to be a global variable for the purpose of assigning LSL sorts. Each variable is given an LSL sort according to its LCL type. This type-to-sort assignment is fundamental to defining the sort compatibility constraints a legal LCL specification must satisfy.

Table B.1 shows how LCL typed variables are assigned LSL sorts. LCL models formal parameters and global variables of the same type differently. Since C supports only one style of parameter passing, pass by value, each formal parameter is a copy of the argument passed. To simplify reasoning about LCL specifications, an explicit environment to map identifiers to objects is not introduced. LCL global variables are simply modeled as objects, that is, if a global variable changes its value, we model it as a change in the mapping between the name of the global variable and its underlying value. For example, if `x` is a global variable of type `int`, then `x` is assigned the sort `int_Obj`. Changes to `x` are modeled as changes in the state binding `x` to `ints`. C treats arrays differently from other types: an array is not copied; it is implicitly passed as a pointer to the first element of the array. LCL carries this exception too: a global variable and a formal parameter of the same array type are both mapped to the same array sort.

Let I be an immutable abstract type, M be a mutable abstract type *Let T be an LCL exposed type*			
LCL Types	*Formal Parameter*	*Global Variable*	*Quantified Variable*
I	I	I_Obj	I
M	M_Obj	M_Obj_Obj	M
primitive type T	T	T_Obj	T
enumerated type T	T	T_Obj	T
pointer to T	T_ObjPtr	T_ObjPtr_Obj	T_ObjPtr
array of T	T_ObjArr	T_ObjArr	T_Vec
struct _S	_S_Tuple	_S_Struct	_S_Tuple
union _U	_U_UnionVal	_U_Union	_U_UnionVal

Table B.1: Assigning sorts to LCL variables.

Kind of C Literal	*Assigned Sorts*
int	int
char	char
C string	char_Vec, char_ObjPtr
float, double	double

Table B.2: Assigning sorts to C literals.

A quantified variable is always given the value sort of its type unless the `obj` qualifier is used. If the `obj` qualifier is present, the object sort corresponding to the type of the quantified variable is used.

B.3 Assigning LSL Sorts to C Literals

C literals can be used in LCL specifications. The following kinds of literals are supported: integers, strings, character literals, and floating point numbers. Single precision floating point numbers are treated as double precision floating point numbers.

Abstract Syntax

cLiteral ::= *intLiteral* | *stringLiteral* | *singleQuoteLiteral* | *floatLiteral*

By C convention, C strings are arrays of C characters terminated by a null character. The built-in sort that models the value of C strings is `char_Vec`. The corresponding array sort is `char_Arr`. Since a C string can be viewed either as an array of characters or as a pointer to a character, C strings are overloaded accordingly. The sort assignments of C literals are shown in Table B.2.

Appendix C

LCL Built-in Operators

A number of LCL operators are automatically generated to model LCL types. They are generated based on the kinds of sorts used to model LCL types. Table C.1 shows the built-in operators corresponding to each kind of sort. A number of other built-in operators such as `fresh` and `trashed` are discussed in the semantics of a function specification in Section 7.4.

For a given kind of sort, the corresponding entry of Table C.1 shows how the built-in operators can be generated by using some LSL traits with the appropriate renamings. These traits are shown in Figures C-1 to C-5. For example, the row for a pointer sort (named `ps`) says that the following auxiliary sorts are expected: a sort corresponding to the value the pointer points to (value sort named `s`), and the object sort corresponding to `s` (named `os`). The last column of the table, for a pointer sort, indicates that the built-in operators can be generated by adding the following `uses` clause to the module: `use lclpointer (s for base, os for baseObj, ps for baseObjPtr)`. This corresponds to the following operators:

```
__ # __: baseObj, state → base
nil: → baseObjPtr
* __: baseObjPtr → baseObj
__ + __, __ - __: baseObjPtr, int → baseObjPtr
__ + __: int, baseObjPtr → baseObjPtr
__ - __: baseObjPtr, baseObjPtr → int
fresh, trashed, unchanged: baseObj → Bool
minIndex, maxIndex: baseObjPtr → int
sizeof: base → int
sizeof: baseObj → int
sizeof: baseObjPtr → int
```

The `nil` operator is the null pointer for the pointer type. There is an operator (`* __`) to dereference a pointer; its result is an object whose value can be retrieved from some state. There are operators for pointer arithmetic, and an auxiliary operator is introduced as a shorthand: `unchanged`. Its definition is given in Table C.2, together with the built-in operator for array types, `isSub`.

The terms in the left hand column of the first two rows of Table C.2 can only appear in the body of a function specification. Their equivalent assertion contains the operators, ^

Sort Kind	*Auxiliary Sorts*	*Implicitly Used Trait*
any sort (s)	none	lclsort (s for base)
object sort (os)	value sort (s)	lclobj (s for base, os for baseObj)
pointer sort (ps)	value sort (s) object sort (os)	lclpointer (s for base, os for baseObj, ps for baseObjPtr)
array sort (as)	value sort (s) object sort (os) vector sort (vs) pointer sort (ps)	lclarray (s for base, os for baseObj, as for baseObjArr, vs for baseVec, ps for baseObjPtr)

Table C.1: LCL built-in operators.

Term with Built-In Operator	*Equivalent Assertion*
unchanged(x)	x'= x^
unchanged(all)	∀ x:AllObjects x'= x^
isSub(a, i)	0 <= i <= maxIndex(a)

Table C.2: Semantics of some LCL built-in operators.

and ′, which are the same states as those implicitly available in the function specification. They can simply be viewed as macros within the body of a function specification.

While the bounds of C arrays are not dynamically kept, they are useful for reasoning about C programs and LCL specifications. A verification system can deduce such information from array declarations and pointer usage. For example, if the LCL variable declaration is given as `int xa[10]`, it provides the following assertion: `maxIndex(xa) = 9`.

The `sizeof` operator is a C built-in operator. It returns the number of bytes an object occupies in an implementation of C. Its semantics is compiler-implementation-dependent.

```
lclsort (base): trait
 introduces
  sizeof: base → int
```

Figure C-1: lclsort.lsl

A struct sort has a number of operators for selecting the components of the struct. Suppose we have the following: `struct _pair {int i; double d;}`. The corresponding LSL sorts are `_pair_Struct` and `_pair_Tuple`. Since the C `&` operator can be applied to the components of C structs, their object identities must be modeled. LCL does this by overloading the field selector operators to extract the objects representing the components of a struct object. For example, we have both `__.i: _pair_Tuple → int` and `__.i: _pair_Struct → int_Obj`. Other automatically generated operators are given in the `lclstruct2` trait shown in Figure C-5. In the pair struct example, the implicitly generated trait corresponds to one with the following renaming: `lclstruct2 (_pair_Tuple, _pair_Struct, int, int_Obj, double, double_Obj, i for field1, d for field2)`.

```
lclobj (base, baseObj): trait
 includes lclsort (base), typedObj (base, baseObj)
 introduces
   fresh, trashed, unchanged: baseObj → Bool
   sizeof: baseObj → int
```

Figure C-2: lclobj.lsl

```
lclpointer (base, baseObj, baseObjPtr): trait
 includes lclobj (base, baseObj)
 introduces
  nil: → baseObjPtr
  * __ : baseObjPtr → baseObj
  minIndex, maxIndex: baseObjPtr → int
  __+__, __-__: baseObjPtr, int → baseObjPtr
  __+__: int, baseObjPtr → baseObjPtr
  __-__: baseObjPtr, baseObjPtr → int
  sizeof: baseObjPtr → int
```

Figure C-3: lclpointer.lsl

```
lclarray (base, baseObj, baseObjArr, baseVec, baseObjPtr): trait
 includes lclpointer (base, baseObj, baseObjPtr)
 introduces
  __ # __: baseObjArr, state → baseVec
  __ [ __ ]: baseObjArr, int → baseObj
  maxIndex: baseObjArr → int
  __ []: baseObjPtr → baseObjArr
  unchanged, fresh, trashed: baseObjArr → Bool
  isSub: baseObjArr, int → Bool
  sizeof: baseObjArr → int
  __ [ __ ]: baseVec, int → base
  isSub: baseVec, int → Bool
  sizeof: baseVec → int
```

Figure C-4: lclarray.lsl

```
lclstruct2 (tup, struct, field1Sort, field1Obj, field2Sort, field2Obj): trait
 includes lclobj (tup, struct)
 introduces
  [ __, __ ]: field1Sort, field2Sort → tup
  __ . field1 : tup → field1Sort
  __ . field2 : tup → field2Sort
  __ . field1 : struct → field1Obj
  __ . field2 : struct → field2Obj
```

Figure C-5: lclstruct2.lsl

Union sorts are handled in exactly the same way as struct sorts. No additional operators are generated for enumeration sorts.

Appendix D

Specification Case Study

The case study consists of the LCL specifications for the following interfaces: genlib, date, security, lot_list, trans, trans_set, and position. The following are the traits supporting the interfaces: char, string, cstring, mystdio, genlib, dateBasics, dateFormat, date, security, lot, list, lot_list, kind, transBasics, transFormat, transParse, trans, trans_set, income, positionBasics, positionMatches, positionExchange, positionReduce, positionTbill, positionSell, and position. These traits use traits from the Larch LSL handbook traits described in [15].

The main features of some interfaces and traits have already been discussed in the body of the thesis. The complete specifications are given below. The specifications have been checked by the LSL checker and LCLint for syntax and type correctness. While no code is shown here, PM is a program in regular use. The program has been also checked by LCLint against the specifications given here.

D.1 The char Trait

```
char (char): trait
 includes Integer
 introduces
  isNumberp, isBlankChar: char → Bool
  char2int: char → Int
  tolower: char → char
    % should be an enumeration of char's but that would cause the output
    % of lsl2lp to blow up.
  'null', 'newline', 'tab', 'escape', 'space', 'slash', 'period': → char
  'minus', 'EOF', 'comma', 'leftParen', 'rightParen', '_': → char
  '0', '1', '2', '3', '4', '5', '6', '7', '8', '9': → char
  'A', 'B', 'C', 'D', 'E', 'F', 'G', 'H', 'I', 'J', 'K', 'L', 'M', 'N': → char
  'O', 'P', 'Q', 'R', 'S', 'T', 'U', 'V', 'W', 'X', 'Y', 'Z': → char
  'a', 'b', 'c', 'd', 'e', 'f', 'g', 'h', 'i', 'j', 'k', 'l', 'm', 'n': → char
  'o', 'p', 'q', 'r', 's', 't', 'u', 'v', 'w', 'x', 'y', 'z': → char
  __ < __ : char, char → Bool % ASCII collating order
 asserts
  ∀ c: char
   isNumberp(c) == (c = '0' ∨ c = '1' ∨ c = '2' ∨ c = '3' ∨ c = '4'
                    ∨ c = '5' ∨ c = '6' ∨ c = '7' ∨ c = '8' ∨ c = '9');
   isBlankChar(c) == (c = 'null' ∨ c = 'newline' ∨ c = 'tab' ∨ c = 'escape'
                      ∨ c = 'space' ∨ c = 'EOF');
   char2int('0') == 0;
    % ...
   char2int('9') == 9;
   tolower('A') == 'a';
    % ...
   tolower('Z') == 'z';
    % for non-alphabet, tolower(c) = c
   tolower('null') == 'null';
    % ...
   tolower('_') == '_';
 implies converts isNumberp, isBlankChar
```

D.2 The cstring Trait

```
cstring: trait
  includes String (char, String), Integer (int for Int)
  introduces
    null: → char
    nullTerminated: String → Bool
    throughNull: String → String
    sameStr: String, String → Bool
    lenStr: String → int
  asserts
    ∀ s, s1, s2: String, c: char
      ¬ nullTerminated(empty);
      nullTerminated(s ⊢ c) ==
        c = null ∨ nullTerminated(s);
      nullTerminated(s)
        ⇒ throughNull(s ⊢ c) = throughNull(s);
      ¬ nullTerminated(s)
        ⇒ throughNull(s ⊢ null) = s ⊢ null;
      sameStr(s1, s2) ==
          throughNull(s1) = throughNull(s2);
      lenStr(s) == len(throughNull(s)) - 1
```

D.3 The string Trait

```
string (C): trait
 includes cstring (C for String), char
 introduces
  isNumeric, nonBlank: C → Bool
  tolower, getString: C → C
  __ < __ : C, C → Bool
  lastElement: C → char
  countChars: C, char → Int
  NthField, NthFieldRest: C, int, char → C
 asserts ∀ s, s1, s2: C, c, cc, sc: char, i: Int
```

```
   null == 'null'; % relate null in cstring.lsl and 'null' in char.lsl
   isNumeric(empty);
   isNumeric(s ⊢ c) == isNumberp(c) ∧ isNumeric(s);
   ¬ nonBlank(empty);
   nonBlank(c ⊣ s) == ¬ isBlankChar(c) ∨ nonBlank(s);
   tolower(empty) == empty;
   tolower(c ⊣ s) == tolower(c) ⊣ tolower(s);
   nullTerminated(s) ⇒
    getString(c ⊣ s) =
     (if c = 'null' then empty else c ⊣ getString(s));
   ¬ ((cc ⊣ s2) < empty);
   empty < (cc ⊣ s2);
   (c ⊣ s) < (cc ⊣ s2) == (c < cc) ∨ (c = cc ∧ (s < s2));
   lastElement(s ⊢ c) = c;
   countChars(empty, cc) == 0;
   countChars(c ⊣ s, cc) ==
    if c = cc then succ(countChars(s, cc)) else countChars(s, cc);
     % NthField returns the string that is between (i-1)th and i-th sc
     % character not including the sc characters.  Start and end of string
     % is viewed as having implicit sc characters.  i starts at 1.
   NthField(empty, i, sc) == empty;
   NthField(c ⊣ s, 1, sc) ==
    (if c = sc then empty else c ⊣ NthField(s, 1, sc));
   i ≥ 1 ⇒ NthField(c ⊣ s, succ(i), sc) =
              (if c = sc then NthField(s, i, sc) else NthField(s, succ(i), sc));
     % NthFieldRest returns the string after the i-th sc
     % character (the returned string does not include the leading sc char).
     % Start of string is viewed as having an implicit sc character,
     % the 0th sc character.  i starts at 0.
   NthFieldRest(empty, i, sc) = empty;
   NthFieldRest(s, 0, sc) == s;
   i ≥ 0 ⇒ NthFieldRest(c ⊣ s, succ(i), sc) =
              (if c = sc then NthFieldRest(s, i, sc)
               else NthFieldRest(s, succ(i), sc));
  implies ∀ s: C, sc: char
   s ≠ empty ⇒
    s = NthField(s, 1, sc) || (sc ⊣ NthFieldRest(s, 1, sc));
   converts isNumeric, nonBlank, tolower: C → C,
    __ < __: C, C → Bool, countChars, lastElement exempting lastElement(empty)
```

D.4 The mystdio Trait

```
mystdio (String): trait
 includes string (String)
 introduces
  peekChar: FILE → char
  canRead: FILE → Bool
  __ || __: FILE, String → FILE
  getLine: FILE → String
  getLineNChars: FILE, Int → String % read up to newline or at most N chars,
  removeLine: FILE, String → FILE
   % String should be a prefix of FILE, remove it from FILE
  replaceNewlineByNull: String → String
  appendedMsg: FILE, FILE, String → Bool
 asserts ∀ f, f2: FILE, s: String, c: char
   replaceNewlineByNull(empty) == empty;
   replaceNewlineByNull(c ⊣ s) ==
    (if c = 'newline' then 'null' else c) ⊣ replaceNewlineByNull(s);
   appendedMsg(f, f2, s) == (f = f2 || s) ∧ nonBlank(s);
 implies converts replaceNewlineByNull, appendedMsg
```

D.5 The genlib Trait

```
genlib (String, Int): trait
 includes mystdio, string, DecimalLiterals (double for N),
          Rational (double for Q), Exponentiation (double for T, nat for N),
          Exponentiation (Int for T, nat for N)
 introduces
  sep_char: → char
  within1: double, double → Bool
  near0: double → Bool
  okFloatString, okNatString, isNumberOrPeriodOrComma: String → Bool
  string2double: String → double
  double2string2: double → String % 2 decimal points
  string2int: String → Int
  string2intExponent: String, Int → Int
  wholePart, decimalPart: String → double
  keepNumbers, leftOfDecimal, rightOfDecimal: String → String
   % skip primitive format conversion routines
  int2double: Int → double
  int2string: Int → String
  int: double → Int
  nat: Int → nat
 asserts ∀ d, d1, d2, eps: double, c, c2: char, s: String, i, n, p: Int
   within1(d1, d2) == abs(d1 - d2) < 1;
   near0(d) == (100 * abs(d)) < 1;
   okFloatString('minus' ⊣ s) == countChars(s, 'period') ≤ 1
    ∧ len(s) > 0 ∧ isNumberOrPeriodOrComma(s);
   okFloatString('period' ⊣ s) == countChars(s, 'period') = 0
    ∧ len(s) > 0 ∧ isNumberOrPeriodOrComma(s);
   isNumberp(c) ⇒
    okFloatString(c ⊣ s) == countChars(s, 'period') = 1
     ∧ isNumberOrPeriodOrComma(s);
   ¬ okFloatString(empty);
   okFloatString(c ⊣ s) ⇒ (c = 'minus' ∨ c = 'period' ∨ isNumberp(c));
   okNatString(empty);
   okNatString(c ⊣ s) == isNumberp(c) ∧ okNatString(s);
   isNumberOrPeriodOrComma(empty);
   isNumberOrPeriodOrComma(c ⊣ s) ==
    (c = 'period' ∨ isNumberp(c) ∨ c = 'comma') ∧ isNumberOrPeriodOrComma(s);
   okFloatString(s) ⇒
    string2double(s) = wholePart(keepNumbers(leftOfDecimal(s))) +
                        decimalPart(keepNumbers(rightOfDecimal(s)));
   double2string2(d) ==     % do truncation
    (if d ≥ 0 then empty else 'minus' ⊣ empty) ||
     (int2string(int(abs(d))) ⊢ 'period') ||
      int2string(int((abs(d) - int2double(int(abs(d)))) * 100));
   okNatString(s) ⇒ string2int(s) = string2intExponent(s, len(s) - 1);
   string2intExponent(empty,  p) == 0;
   (okNatString(c ⊣ s) ∧ p ≥ 0) ⇒
    string2intExponent(c ⊣ s, p) =
     (char2int(c) * (10**nat(p))) + string2intExponent(s, p - 1);
   leftOfDecimal(empty) == empty;
   leftOfDecimal(c ⊣ s) ==
    if c = 'period' then empty else c ⊣ leftOfDecimal(s);
   rightOfDecimal(empty) == empty;
   rightOfDecimal(c ⊣ s) == if c = 'period' then s else rightOfDecimal(s);
   keepNumbers(empty) == empty;
   keepNumbers(c ⊣ s) ==
    if isNumberp(c) then c ⊣ keepNumbers(s) else keepNumbers(s);
   wholePart(empty) == 0;
   isNumeric(s) ⇒ wholePart('minus' ⊣ s) = - int2double(string2int(s));
   isNumeric(c ⊣ s) ⇒
    wholePart(c ⊣ s) = int2double(string2int(c ⊣ s));
   decimalPart(empty) == 0;
   isNumeric(s) ⇒
    decimalPart(s) = int2double(string2int(s)) / (10**nat(len(s)));
 implies
  converts within1, near0, okFloatString, okNatString, keepNumbers,
   isNumberOrPeriodOrComma, double2string2, leftOfDecimal, rightOfDecimal
```

D.6 The dateBasics Trait

```
dateBasics: trait
 includes Integer, TotalOrder (date)
 date tuple of month, day, year: Int % unknown month is 0, jan 1, ... dec 12.
 introduces
  isInLeapYear: date → Bool
  isLeapYear: Int → Bool
  validMonth: Int → Bool
  __ - __ : date, date → Int
  daysBetween: Int, Int → Int
  dayOfYear, daysToEnd: date → Int
  dayOfYear2: Int, Int, Int, Int → Int
  daysInMonth: Int, Int → Int
 asserts ∀ d, d2: date, k, m, yr, yr2: Int, mth, mth2: Int
   isInLeapYear(d) == isLeapYear(d.year);
   isLeapYear(yr) == mod(yr, 400) = 0 ∨ (mod(yr, 4) = 0 ∧ mod(yr, 100) ≠ 0);
   validMonth(mth) == mth ≥ 0 ∧ mth ≤ 12;
   d < d2 == d.year < d2.year
             ∨ (d.year = d2.year ∧ dayOfYear(d) < dayOfYear(d2));
   d ≥ d2 ⇒
    d - d2 = (if d.year = d2.year then dayOfYear(d) - dayOfYear(d2)
              else daysToEnd(d2) + dayOfYear(d) +
                    daysBetween(succ(d2.year), d.year));
   yr ≤ yr2 ⇒
    daysBetween(yr, yr2) =
     (if yr = yr2 then 0
      else (if isLeapYear(yr) then 366 else 365) + daysBetween(succ(yr), yr2));
   (validMonth(d.month) ∧ (d.month ≠ 0 ∨ d.day = 0)) ⇒
    dayOfYear(d) = (if d.month = 0 then 0
                    else dayOfYear2(d.month, 1, d.day, d.year));
   (validMonth(mth) ∧ validMonth(mth2)) ⇒
    dayOfYear2(mth, mth2, k, yr) =
     (if mth = mth2 then k
      else dayOfYear2(mth, succ(mth2), k + daysInMonth(mth2, yr), yr));
   validMonth(d.month) ⇒
    daysToEnd(d) = (if isInLeapYear(d) then 366 else 365) - dayOfYear(d);
   (validMonth(mth) ∧ mth ≠ 0) ⇒
    daysInMonth(mth, yr) =
     (if mth = 2 then if isLeapYear(yr) then 29 else 28
      else if mth = 1 ∨ mth = 3 ∨ mth = 5 ∨ mth = 7 ∨ mth = 8 ∨ mth = 10
              ∨ mth = 12
          then 31 else 30);
 implies
  converts isInLeapYear, isLeapYear
```

D.7 The dateFormat Trait

```
dateFormat: trait
 includes genlib, dateBasics
 introduces
  okDateFormat, isNormalDateFormat: String → Bool
  validDay: Int, Int, Int → Bool
 asserts ∀ s: String, i, m, yr: Int
   okDateFormat(s) == (len(s) = 2 ∧ s[0] = 'L' ∧ s[1] = 'T')
                      ∨ isNormalDateFormat(s);
   isNormalDateFormat(s) == (len(s) ≥ 5) ∧ (len(s) ≤ 8)
     ∧ countChars(s, 'slash') = 2 ∧ NthField(s, 1, 'slash') != empty
     ∧ isNumeric(NthField(s, 1, 'slash'))
     ∧ validMonth(string2int(NthField(s, 1, 'slash')))
     ∧ NthField(s, 2, 'slash') != empty
     ∧ isNumeric(NthField(s, 2, 'slash'))
     ∧ NthField(s, 3, 'slash') != empty
     ∧ isNumeric(NthField(s, 3, 'slash'))
     ∧ validDay(string2int(NthField(s, 2, 'slash')),
                string2int(NthField(s, 1, 'slash')),
                string2int(NthField(s, 3, 'slash')));
   validDay(i, m, yr) == (i ≥ 0) ∧ (i ≤ 31)
```

```
    ∧ ((m = 0 ∧ i = 0) ∨ % reject 0/non-0-day/yr format
        (m > 0 ∧ m ≤ 12 ∧ i ≤ daysInMonth(m, yr)));
  implies converts okDateFormat, isNormalDateFormat, validDay
```

D.8 The date Trait

```
date: trait
 includes dateFormat (ndate for date), TotalOrder (date)
 date union of normal: ndate, special: Bool
 introduces
  null_date: → date % serves as an uninitialized date.
  isLT, isNullDate, isNormalDate, isInLeapYear: date → Bool
  year: date → Int
  __ - __ : date, date → Int
  is_long_term: date, date, Int → Bool
  string2date: String → date
  date2string: date → String
  fixUpYear: Int → Int
 asserts ∀ d, d2: date, nd: ndate, s: String, i, day, yr: Int
   null_date == special(false);
   isNullDate(d) == d = null_date;
   isLT(d) == tag(d) = special ∧ d.special;
   isNormalDate(d) == tag(d) = normal;
   isNormalDate(d) ⇒ isInLeapYear(d) = isInLeapYear(d.normal);
   isNormalDate(d) ⇒ year(d) = d.normal.year;
   (isNormalDate(d) ∧ isNormalDate(d2)) ⇒ (d - d2 = d.normal - d2.normal);
   (isNormalDate(d) ∧ isNormalDate(d2)) ⇒
    is_long_term(d, d2, i) = ((d.normal - d2.normal) > i);
   (isNormalDate(d) ∧ isNormalDate(d2)) ⇒ (d < d2 = d.normal < d2.normal);
   (isLT(d) ∧ isNormalDate(d2)) ⇒ (d < d2);
   null_date < d == not(d = null_date); % non-reflexive
   okDateFormat(s) ⇒
    string2date(s) =
     (if (len(s) = 2 ∧ s[0] = 'L' ∧ s[1] = 'T') then special(true)
      else normal([string2int(NthField(s, 1, 'slash')),
                   string2int(NthField(s, 2, 'slash')),
                   fixUpYear(string2int(NthField(s, 3, 'slash')))]));
   yr ≥ 0 ⇒ fixUpYear(yr) = (if yr < 50 then 2000 + yr else 1900 + yr);
   isNormalDate(d) ⇒ string2date(date2string(d)) = d;
  implies
    ∀ d: date
     isNormalDate(d) ⇒ dayOfYear(d.normal) + daysToEnd(d.normal) =
                        (if isInLeapYear(d) then 366 else 365)
```

D.9 The security Trait

```
security (String, Int): trait
 includes string (String for C)
 security tuple of sym: String % current model. future: add other attributes
 introduces
```

```
  __ < __: security, security → Bool
  hasNamePrefix: security, String → Bool
  isCashSecurity: security → Bool
   % these strings can obviously be defined using chars, ⊣, and empty
  'AmExGvt', 'Cash', 'LehmanBrosDaily', 'SmBarShGvt', 'USTrM',
    'USTrS': → String
 asserts ∀ s, s2: security, str: String
   s < s2 == tolower(s.sym) < tolower(s2.sym);
   hasNamePrefix(s, str) == prefix(s.sym, len(str)) = str;
   isCashSecurity(s) == hasNamePrefix(s, 'AmExGvt') ∨ hasNamePrefix(s, 'Cash')
    ∨ hasNamePrefix(s, 'LehmanBrosDaily') ∨ hasNamePrefix(s, 'SmBarShGvt')
    ∨ hasNamePrefix(s, 'USTrM') ∨ hasNamePrefix(s, 'USTrS');
 implies converts hasNamePrefix, __ < __: security, security→ Bool,
                  isCashSecurity
```

D.10 The lot Trait

```
lot (String, Int): trait
 includes string (String)
 introduces
  string2lot: String → lot
  lot2string: lot → String
  __ < __: lot, lot → Bool
 asserts ∀ x, y: lot
   string2lot(lot2string(x)) == x;
   x < y == lot2string(x) < lot2string(y);
 implies lot partitioned by lot2string
  converts __ < __: lot, lot → Bool
```

D.11 The list Trait

```
list (E, list) :trait
 includes Integer
 introduces
  nil: → list
  cons: E, list → list
  car: list → E
  cdr: list → list
  __ ∈ __: E, list → Bool
  length: list → Int
  count: E, list → Int
 asserts
  list generated by nil, cons
  list partitioned by car, cdr
  ∀ x, y:list, e, f: E, i: Int
    car(cons(e, x)) = e;
    cdr(cons(e, x)) = x;
    ¬ (e ∈ nil);
    e ∈ cons(f, x) == (e = f) ∨ e ∈ x;
    length(nil) == 0;
    length(cons(e, x)) == succ(length(x));
    count(e, nil) == 0;
    count(e, cons(f, x)) == if e = f then succ(count(e, x)) else count(e, x);
 implies
  converts car, cdr exempting ∀ i: int car(nil), cdr(nil)
  converts __ ∈ __, length, count
```

D.12 The lot_list Trait

```
lot_list (String, Int): trait
 includes lot (String, Int), list (lot, lot_list)
 introduces
  __ < __ : lot_list, lot_list → Bool
  sorted, uniqueLots: lot_list → Bool
 asserts ∀ e, f: lot, x, y: lot_list, s: String, c: char
   nil < cons(e, y);
   ¬ (cons(e, x) < nil);
   cons(e, x) < cons(f, y) == (e < f) ∨ (e = f ∧ (x < y));
   sorted(nil);
   sorted(cons(e, nil));
   sorted(cons(e, cons(f, x))) == e < f ∧ sorted(x);
   uniqueLots(nil);
   uniqueLots(cons(e, x)) == ¬ (e ∈ x) ∧ uniqueLots(x);
 implies converts __ < __: lot_list, lot_list → Bool, sorted, uniqueLots
```

D.13 The kind Trait

```
kind (String, kind): trait
 includes string (String)
 kind enumeration of buy, sell, cash_div, cap_dist, tbill_mat, exchange, interest,
                     muni_interest, govt_interest, new_security, other
 introduces
  validKindFormat: String → Bool
  string2kind: String → kind
  needsLot, isInterestKind: kind → Bool
 asserts ∀ k: kind, s: String, sc, c: char
   validKindFormat(s) ==
    (len(s) = 1
     ∧ (s[0] = 'B' ∨ s[0] = 'S' ∨ s[0] = 'E' ∨ s[0] = 'D' ∨ s[0] = 'I'
          ∨ s[0] = 'C' ∨ s[0] = 'M' ∨ s[0] = 'N'))
    ∨ (len(s) = 2 ∧ (s[0] = 'I' ∧ (s[1] = 'M' ∨ s[1] = 'G')));
   string2kind('B' ⊣ empty) == buy;
   string2kind('S' ⊣ empty) == sell;
   string2kind('E' ⊣ empty) == exchange;
   string2kind('D' ⊣ empty) == cash_div;
   string2kind('I' ⊣ empty) == interest;
   string2kind('C' ⊣ empty) == cap_dist;
   string2kind('M' ⊣ empty) == tbill_mat;
   string2kind('N' ⊣ empty) == new_security;
   string2kind('I' ⊣ ('M' ⊣ empty)) == muni_interest;
   string2kind('I' ⊣ ('G' ⊣ empty)) == govt_interest;
   ¬ validKindFormat(s) ⇒ string2kind(s) = other;
   needsLot(k) == (k = buy ∨ k = sell ∨ k = exchange ∨ k = cap_dist
                    ∨ k = tbill_mat);
   isInterestKind(k) == (k = interest ∨ k = muni_interest ∨ k = govt_interest);
 implies converts validKindFormat, string2kind, needsLot, isInterestKind
```

D.14 The transBasics Trait

```
transBasics: trait
 includes genlib, date, kind, security, lot_list
 trans tuple of security: security, kind: kind, amt, price, net: double,
               date: date, lots: lot_list, input: String, comment: String
```

D.15 The transFormat Trait

```
transFormat: trait
 includes transBasics
 fields enumeration of security, kind, amt, price, net, date, lots, comment
 introduces
  okTransFormat: String → Bool
  okTransFormatByKind: kind, String → Bool
  hasAllFields, noPriceLotsFields, noPriceField: String → Bool
  okLotsFormat, areNumbersOrCommas: String → Bool
  getField: String, fields → String
  getComment: String, kind, double → String
 asserts ∀ s: String, sc: char, k: kind, amt: double
   okTransFormat(s) == len(s) > 0 ∧ getField(s, security) != empty
    ∧ getField(s, kind) != empty
    ∧ okTransFormatByKind(string2kind(getField(s, kind)), s);
   okTransFormatByKind(buy, s) == hasAllFields(s);
   okTransFormatByKind(sell, s) == hasAllFields(s);
   okTransFormatByKind(exchange, s) == hasAllFields(s);
   okTransFormatByKind(interest, s) == noPriceLotsFields(s);
   okTransFormatByKind(muni_interest, s) == noPriceLotsFields(s);
   okTransFormatByKind(govt_interest, s) == noPriceLotsFields(s);
   okTransFormatByKind(cap_dist, s) == noPriceField(s);
   okTransFormatByKind(tbill_mat, s) == noPriceField(s);
   okTransFormatByKind(new_security, s) == okDateFormat(getField(s, date));
   ¬ okTransFormatByKind(other, s);
   hasAllFields(s) == okFloatString(getField(s, amt))
    ∧ okFloatString(getField(s, price)) ∧ okFloatString(getField(s, net))
    ∧ okDateFormat(getField(s, date)) ∧ okLotsFormat(getField(s, lots));
   noPriceLotsFields(s) == okFloatString(getField(s, amt))
    ∧ getField(s, price) = empty ∧ okFloatString(getField(s, net))
    ∧ okDateFormat(getField(s, date));
   noPriceField(s) == okFloatString(getField(s, amt))
    ∧ getField(s, price) = empty ∧ okFloatString(getField(s, net))
    ∧ okDateFormat(getField(s, date)) ∧ okLotsFormat(getField(s, lots));
    % entry ::= security kind amt price net date [lots] [comment]
   getField(s, security) == NthField(s, 1, sep_char);
   getField(s, kind) == NthField(s, 2, sep_char);
   getField(s, amt) == NthField(s, 3, sep_char);
   getField(s, price) == NthField(s, 4, sep_char);
   getField(s, net) == NthField(s, 5, sep_char);
   getField(s, date) == NthField(s, 6, sep_char);
   getField(s, lots) == NthField(s, 7, sep_char);
   getComment(s, k, amt) == if ¬ needsLot(k) ∨ amt = 0
                            then NthFieldRest(s, 6, sep_char)
                            else NthFieldRest(s, 7, sep_char);
   okLotsFormat(s) == len(s) > 0 ∧ areNumbersOrCommas(s)
                      ∧ lastElement(s) ≠ 'comma';
   areNumbersOrCommas(empty);
   areNumbersOrCommas(sc ⊣ s) ==
    (sc = 'comma' ∨ isNumberp(sc)) ∧ areNumbersOrCommas(s);
```

D.16 The transParse Trait

```
transParse: trait
 includes transFormat
 introduces
  string2trans: String → trans
  string2transByKind: kind, String → trans
  withAllFields, withNoPriceLotsFields, withNoPriceField: String → trans
  string2lot_list: String → lot_list
 asserts ∀ s: String
   okTransFormat(s) ⇒
    string2trans(s) = string2transByKind(string2kind(getField(s, kind)), s);
   string2transByKind(buy, s) == withAllFields(s);
   string2transByKind(sell, s) == withAllFields(s);
   string2transByKind(exchange, s) == withAllFields(s);
   string2transByKind(interest, s) == withNoPriceLotsFields(s);
   string2transByKind(muni_interest, s) == withNoPriceLotsFields(s);
   string2transByKind(govt_interest, s) == withNoPriceLotsFields(s);
   string2transByKind(cap_dist, s) == withNoPriceField(s);
   string2transByKind(tbill_mat, s) == withNoPriceField(s);
   string2transByKind(new_security, s) ==
    [ [getField(s, security)], string2kind(getField(s, kind)), 0, 0, 0,
      string2date(getField(s, date)), nil, s, NthField(s, 7, sep_char)];
   withAllFields(s) ==
    [ [getField(s, security)], string2kind(getField(s, kind)),
      string2double(getField(s, amt)), string2double(getField(s, price)),
      string2double(getField(s, net)), string2date(getField(s, date)),
      string2lot_list(getField(s, lots)), s,
      getComment(s, string2kind(getField(s, kind)),
                 string2double(getField(s, amt)))];
   withNoPriceLotsFields(s) ==
    [ [getField(s, security)], string2kind(getField(s, kind)),
      string2double(getField(s, amt)), string2double(getField(s, price)),
      string2double(getField(s, net)), string2date(getField(s, date)),
      nil, s, NthFieldRest(s, 6, sep_char)];
   withNoPriceField(s) ==
    [ [getField(s, security)], string2kind(getField(s, kind)),
      string2double(getField(s, amt)), 0, string2double(getField(s, net)),
      string2date(getField(s, date)), string2lot_list(getField(s, lots)),
      s, getComment(s, string2kind(getField(s, kind)),
                    string2double(getField(s, amt)))];
   okLotsFormat(s) ⇒
    string2lot_list(s) =
     (if s = empty then nil
      else cons(string2lot(NthField(s, 1, 'comma')),
                string2lot_list(NthFieldRest(s, 1, 'comma'))));
```

D.17 The trans Trait

```
trans (String): trait
 includes transParse
 introduces
  transIsConsistent: trans, kind → Bool
  __ ≤ __ : trans, trans → Bool
 asserts ∀ t, t2: trans
   transIsConsistent(t, buy) == t.net ≥ 0 ∧ t.amt > 0 ∧ t.price ≥ 0
     ∧ length(t.lots) = 1 ∧ within1(t.amt * t.price, t.net);
    % sell amount may be 0 to handle special court-ordered settlements.
    % also cannot give away securities for free.
   transIsConsistent(t, sell) == t.net > 0 ∧ t.amt ≥ 0 ∧ t.price > 0
    ∧ isNormalDate(t.date) ∧ uniqueLots(t.lots)
    ∧ (t.amt > 0 ⇒ within1(t.amt * t.price, t.net));
   transIsConsistent(t, cash_div) == t.amt > 0;
   transIsConsistent(t, exchange) == t.amt > 0 ∧ length(t.lots) = 1;
   transIsConsistent(t, cap_dist) == t.net > 0 ∧ t.amt > 0
                                     ∧ length(t.lots) = 1;
   transIsConsistent(t, tbill_mat) == t.net > 0 ∧ t.amt > 0
                                      ∧ uniqueLots(t.lots);
    % negative interests arise when bonds are purchased between their interest
    %  payment periods.
   transIsConsistent(t, interest);
   transIsConsistent(t, muni_interest);
   transIsConsistent(t, govt_interest);
   transIsConsistent(t, new_security);
   ¬ transIsConsistent(t, other);
   t ≤ t2 == (t.security < t2.security)
               ∨ (t.security = t2.security ∧ t.date ≤ t2.date);
 implies converts transIsConsistent, __ ≤ __: trans, trans → Bool
```

D.18 The trans_set Trait

```
trans_set (String, trans_set_obj): trait
```

```
 includes trans, Set (trans, tset)
 trans_set tuple of val: tset, activeIters: Int
 trans_set_iter tuple of toYield: tset, setObj: trans_set_obj
 introduces
  yielded: trans, trans_set_iter, trans_set_iter → Bool
  startIter: trans_set → trans_set
  endIter: trans_set → trans_set
  matchKey: security, lot, tset → Bool
  findTrans: security, lot, tset → trans
  sum_net, sum_amt: tset → double
 asserts ∀ t: trans, ts: tset, s: security, e: lot, trs: trans_set,
          it, it2: trans_set_iter
   yielded(t, it, it2) ==
    (t ∈ it.toYield) ∧ it2 = [delete(t, it.toYield), it.setObj];
   startIter(trs) == [trs.val, trs.activeIters + 1];
   endIter(trs) == [trs.val, trs.activeIters - 1];
   ¬ matchKey(s, e, {});
   matchKey(s, e, insert(t, ts)) ==
    (s = t.security ∧ length(t.lots) = 1 ∧ e = car(t.lots))
     ∨ matchKey(s, e, ts);
   matchKey(s, e, ts) ⇒ (findTrans(s, e, ts) ∈ ts
    ∧ car(findTrans(s, e, ts).lots) = e ∧ findTrans(s, e, ts).security = s
    % buy trans has single lots, only interested in matching buy trans
    ∧ length(findTrans(s, e, ts).lots) = 1);
   sum_net({}) == 0;
   t ∈ ts ⇒ sum_net(insert(t, ts)) = sum_net(ts);
   ¬ (t ∈ ts) ⇒ sum_net(insert(t, ts)) = t.net + sum_net(ts);
   sum_amt({}) == 0;
   t ∈ ts ⇒ sum_amt(insert(t, ts)) = sum_amt(ts);
   ¬ (t ∈ ts) ⇒ sum_amt(insert(t, ts)) = t.amt + sum_amt(ts);
 implies converts matchKey, sum_net, sum_amt
```

D.19 The income Trait

```
income (String, income): trait
 includes kind (String, kind), genlib (String, Int)
 income tuple of capGain, dividends, totalInterest, ltCG_CY, stCG_CY,
                 dividendsCY, taxInterestCY, muniInterestCY,
                 govtInterestCY: double
 introduces
  emptyIncome: → income
  sum_incomes: income, income → income
  incCYInterestKind: income, double, kind → income
  incInterestKind: income, double, kind, Int, Int → income
  incDividends: income, double, Int, Int → income
  incCapGain: income, double, double, Int, Int → income
   % formatting details, leave unspecified
  income2string, income2taxString: income → String
 asserts ∀ amt, lt, st: double, i, i2: income, yr, tyr: Int, k: kind
   emptyIncome == [0, 0, 0, 0, 0, 0, 0, 0, 0];
  sum_incomes(i, i2) ==
   [i.capGain + i2.capGain, i.dividends + i2.dividends,
    i.totalInterest + i2.totalInterest, i.ltCG_CY + i2.ltCG_CY,
    i.stCG_CY + i2.stCG_CY, i.dividendsCY + i2.dividendsCY,
    i.taxInterestCY + i2.taxInterestCY, i.muniInterestCY + i2.muniInterestCY,
    i.govtInterestCY + i2.govtInterestCY];
  incCYInterestKind(i, amt, interest) ==
   set_taxInterestCY(i, i.taxInterestCY + amt);
  incCYInterestKind(i, amt, muni_interest) ==
   set_muniInterestCY(i, i.muniInterestCY + amt);
  incCYInterestKind(i, amt, govt_interest) ==
   set_govtInterestCY(i, i.govtInterestCY + amt);
  isInterestKind(k) ⇒
   incInterestKind(i, amt, k, yr, tyr) =
    (if yr = tyr
     then set_totalInterest(incCYInterestKind(i, amt, k),
                            i.totalInterest + amt)
     else incCYInterestKind(i, amt, k));
  incDividends(i, amt, tyr, yr) ==
   set_dividends(if tyr = yr then set_dividendsCY(i, i.dividendsCY + amt)
                 else i, i.dividends + amt);
  incCapGain(i, lt, st, tyr, yr) ==
   set_capGain(if tyr = yr
               then set_ltCG_CY(set_stCG_CY(i, i.stCG_CY + st),
                                i.ltCG_CY + lt)
               else i, st + lt);
 implies converts incDividends, incCapGain
```

D.20 The positionBasics Trait

```
positionBasics: trait
 includes trans_set, date, income
 position tuple of security: security, amt: double, income: income,
                   lastTransDate: date, openLots: tset, taxStr: String
 introduces
  set_amtOlotsDate: position, double, tset, date → position
  adjust_amt_and_net: trans, double → trans
  update_olots: tset, trans, double → tset
  __.capGain, __.dividends, __.totalInterest, __.ltCG_CY, __.stCG_CY,
   __.dividendsCY, __.taxInterestCY, __.muniInterestCY,
   __.govtInterestCY: position → double
 asserts ∀ amt: double, p: position, yr, tyr: Int, sd: date,
          t, mt: trans, ts: tset
  set_amtOlotsDate(p, amt, ts, sd) ==
   set_amt(set_openLots(set_lastTransDate(p, sd), ts), amt);
  adjust_amt_and_net(t, amt) =
   set_net(set_amt(t, t.amt - amt), t.net - ((t.net / t.amt) * amt));
  update_olots(ts, t, amt) =
   insert(adjust_amt_and_net(t, amt), delete(t, ts));
   % convenient abbreviations
  p.capGain == p.income.capGain;
  p.dividends == p.income.dividends;
  p.totalInterest == p.income.totalInterest;
  p.ltCG_CY == p.income.ltCG_CY;
  p.stCG_CY == p.income.stCG_CY;
  p.dividendsCY == p.income.dividendsCY;
  p.taxInterestCY == p.income.taxInterestCY;
  p.muniInterestCY == p.income.muniInterestCY;
  p.govtInterestCY == p.income.govtInterestCY;
 implies converts adjust_amt_and_net, set_amtOlotsDate
```

D.21 The positionMatches Trait

```
positionMatches: trait
 includes positionBasics
 introduces
  validMatch: position, trans → Bool
  validMatches: position, trans, Bool → Bool
  validAllMatches: tset, security, lot_list, double, Bool → Bool
  findMatch: position, trans → trans
 asserts ∀ amt: double, p: position, e: lot, y: lot_list, se: security,
          t: trans, ts: tset, completeLot: Bool
  validMatch(p, t) == matchKey(t.security, car(t.lots), p.openLots)
                      ∧ length(t.lots) = 1;
  validMatches(p, t, completeLot) == (t.kind = sell ∧ t.amt = 0)
   % above: selling zero shares is for special court-ordered settlements.
   ∨ (t.lots ≠ nil
       ∧ validAllMatches(p.openLots, t.security, t.lots, t.amt, completeLot));
  validAllMatches(ts, se, nil, amt, completeLot) ==
   if completeLot then amt = 0 else amt ≤ 0;
  validAllMatches(ts, se, cons(e, y), amt, completeLot) ==
   amt > 0 ∧ matchKey(se, e, ts)
   ∧ validAllMatches(ts, se, y, amt - findTrans(se, e, ts).amt, completeLot);
  validMatch(p, t) ⇒     % an abbreviation
   findMatch(p, t) = findTrans(t.security, car(t.lots), p.openLots);
 implies converts validMatch, validMatches, validAllMatches
```

D.22 The positionExchange Trait

```
positionExchange: trait
 includes positionMatches
 introduces
  match_exchange: position, trans → tset
  update_exchange: position, trans → position
 asserts ∀ p: position, t: trans
  validMatch(p, t) ⇒
   match_exchange(p, t) =
    (if t.amt ≥ findMatch(p, t).amt then delete(t, p.openLots)
     else update_olots(p.openLots, t, findMatch(p, t).amt - t.amt));
  (t.kind = exchange ∧ validMatch(p, t)) ⇒
   update_exchange(p, t) =
    set_amtOlotsDate(p, p.amt - t.amt, match_exchange(p, t), t.date);
```

D.23 The positionReduce Trait

```
positionReduce: trait
 includes positionMatches
 reduceMatch tuple of profit: double, olots: tset
 introduces
  reduce_cost_basis: position, trans → reduceMatch
  update_cap_dist: position, trans, Int → position % need cur_year
  adjust_net: trans, double → trans
  update_olots_net: tset, trans, double → tset
 asserts ∀ p: position, t: trans, yr: Int, amt: double, ts: tset
  validMatch(p, t) ⇒
   reduce_cost_basis(p, t) =
    [max(t.net - findMatch(p, t).net, 0),
     update_olots_net(p.openLots, findMatch(p, t),
                      max(findMatch(p, t).net - t.net, 0))];
  (t.kind = cap_dist ∧ validMatch(p, t)) ⇒
   update_cap_dist(p, t, yr) =
    set_amtOlotsDate(
     set_income(p, incDividends(p.income, reduce_cost_basis(p, t).profit,
                                year(t.date), yr)),
                     p.amt, reduce_cost_basis(p, t).olots, t.date);
  adjust_net(t, amt) == set_net(set_price(t, amt/t.amt), amt);
  update_olots_net(ts, t, amt) = insert(adjust_net(t, amt), delete(t, ts));
```

D.24 The positionSell Trait

```
positionSell: trait
 includes positionMatches
 sellMatch tuple of profits: allProfits, taxStr: String, olots: tset
 allProfits tuple of total, LT, ST: double
 introduces
  update_sell: position, trans, Int, Int, Int → position
  match_sell: position, trans, Int → sellMatch
  matchSells: position, trans, Int, lot_list, double, sellMatch → sellMatch
   % convenient abbreviations
  updateSellMatch: sellMatch, trans, Int, Bool, trans → sellMatch
  updateCapGain: position, allProfits, Int, Int, String, Int → position
  sellGain: trans, trans, double → double
  splitProfits: allProfits, date, date, double, Int → allProfits
  summarizeSell: trans, trans, Int, double → String
   % summarizeSell takes a matched buy trans, a sell trans, the holding period
   % and an amt, prints income from selling the lot, the cost of the lot,
   % the transaction dates, and whether they are LT or ST gains.
 asserts ∀ amt: double, p: position, taxl, hp, yr, tyr: Int, s: String,
          sd, bd: date, e: lot, y: lot_list, t, mt: trans, completeLot: Bool,
          sm: sellMatch, ap: allProfits
  (t.kind = sell ∧ validMatches(p, t, false)) ⇒
   update_sell(p, t, hp, yr, taxl) =
    set_amtOlotsDate(updateCapGain(p, match_sell(p, t, hp).profits,
                                   year(t.date), yr,
                                   match_sell(p, t, hp).taxStr, taxl),
                     p.amt - t.amt, match_sell(p, t, hp).olots, t.date);
  (t.kind = sell ∧ validMatches(p, t, false)) ⇒
   match_sell(p, t, hp) =
    (if t.amt = 0
     then [ [t.net, t.net, 0], empty, p.openLots] % assume LT gain, no buy date
     else matchSells(p, t, hp, t.lots, t.amt, [ [0, 0, 0], empty, p.openLots]));
  validMatches(p, t, false) ⇒
   matchSells(p, t, hp, nil, amt, sm) = sm;
  validMatches(p, t, false) ⇒
   matchSells(p, t, hp, cons(e, y), amt, sm) =
    (if amt ≥ findTrans(t.security, e, p.openLots).amt
     then matchSells(p, t, hp, y,
                     amt - findTrans(t.security, e, p.openLots).amt,
                     updateSellMatch(sm, t, hp, true,
                                     findTrans(t.security, e, p.openLots)))
     else matchSells(p, t, hp, y, amt - mt.amt,
                     updateSellMatch(sm, t, hp, false,
                                     findTrans(t.security, e, p.openLots))));
  updateSellMatch(sm, t, hp, completeLot, mt) ==
   if completeLot
   then [splitProfits(sm.profits, mt.date, t.date, sellGain(mt, t, amt), hp),
         sm.taxStr ‖ summarizeSell(mt, t, hp, mt.amt), delete(mt, sm.olots)]
   else [splitProfits(sm.profits, mt.date, t.date, sellGain(mt, t, amt), hp),
         sm.taxStr ‖ summarizeSell(mt, t, hp, mt.amt - amt),
         update_olots(sm.olots, mt, mt.amt - amt)];
  sellGain(t, mt, amt) ==
   if amt = 0 then t.net else min(amt, mt.amt)*((t.net/t.amt)-(mt.net/mt.amt));
  splitProfits(ap, bd, sd, amt, hp) ==
   if is_long_term(bd, sd, hp) then [ap.total + amt, ap.LT + amt, ap.ST]
                               else [ap.total + amt, ap.LT, ap.ST + amt];
  updateCapGain(p, ap, tyr, yr, s, taxl) ==
   set_taxStr(set_income(p, incCapGain(p.income, ap.ST, ap.LT, tyr, yr)),
              if tyr = yr ∧ ¬ near0(ap.total)
              then prefix(p.taxStr ‖ s, taxl)
              else p.taxStr);
 implies converts updateSellMatch, sellGain, splitProfits, updateCapGain
```

D.25 The positionTbill Trait

```
positionTbill: trait
 includes positionMatches
 TbillMatch tuple of net: double, olots: tset
 introduces
  update_tbill_mat: position, trans, Int → position
  tbillInterestOk: position, trans → Bool
  match_tbill: position, trans → TbillMatch
  matchTbillBuys: position, trans, lot_list, TbillMatch → TbillMatch
  updateTBillMatch: position, trans, lot, TbillMatch → TbillMatch
 asserts ∀ p: position, e: lot, y: lot_list, t: trans, tb: TbillMatch,
          yr: Int
  (t.kind = tbill_mat ∧ validMatches(p, t, true) ∧ tbillInterestOk(p, t)) ⇒
   update_tbill_mat(p, t, yr) =
    set_amtOlotsDate(set_income(p, incInterestKind(p.income, t.net, t.kind,
                                                   yr, year(t.date))),
                     p.amt - t.amt, match_tbill(p, t).olots, t.date);
  (t.kind = tbill_mat ∧ validMatches(p, t, true)) ⇒
   tbillInterestOk(p, t) = within1(t.net, (t.amt*100) - match_tbill(p, t).net);
  validMatches(p, t, true) ⇒
   match_tbill(p, t) = matchTbillBuys(p, t, t.lots, [0, p.openLots]);
  validMatches(p, t, true) ⇒
   matchTbillBuys(p, t, nil, tb) = tb;
  validMatches(p, t, true) ⇒
   matchTbillBuys(p, t, cons(e, y), tb) =
    matchTbillBuys(p, t, y, updateTBillMatch(p, t, e, tb));
  updateTBillMatch(p, t, e, tb) ==
   [tb.net + findTrans(t.security, e, p.openLots).net,
    delete(findTrans(t.security, e, p.openLots), tb.olots)];
```

D.26 The position Trait

```
position (String): trait
 includes positionExchange, positionReduce, positionSell, positionTbill
 introduces
  isInitialized: position → Bool
  create: String → position
  update_buy: position, trans → position
  update_dividends, update_interest, update_cap_dist, update_tbill_mat:
   position, trans, Int → position % need cur_year
  validMatchWithBuy: position, trans → Bool
  totalCost: position → double
   % formatting details, leave unspecified
  position2string, position2taxString, position2olotsString: position → String
 asserts ∀ p, p2: position, taxl, yr: Int, t: trans, s: String
  isInitialized(p) == ¬ (p.lastTransDate = null_date);
  create(s) == [ [s], 0, emptyIncome, null_date, {}, empty];
  validMatchWithBuy(p, t) ==
   if t.kind = sell then validMatches(p, t, false)
   else if t.kind = tbill_mat
        then validMatches(p, t, true) ∧ tbillInterestOk(p, t)
        else if t.kind = exchange
             then validMatch(p, t) ∧ findMatch(p, t).amt ≥ t.amt
             else t.kind = cap_dist ∧ validMatch(p, t);
  totalCost(p) == if p.lastTransDate = null_date ∨ p.amt = 0 then 0
                  else sum_net(p.openLots);
  t.kind = buy ⇒
   update_buy(p, t) =
    set_amtOlotsDate(p, p.amt + t.amt, insert(t, p.openLots), t.date);
  t.kind = cash_div ⇒
   update_dividends(p, t, yr) =
    set_lastTransDate(
     set_income(p, incDividends(p.income, t.net, year(t.date), yr)),
                      t.date);
  isInterestKind(t.kind) ⇒
   update_interest(p, t, yr) =
    set_lastTransDate(
     set_income(p, incInterestKind(p.income, t.net, t.kind, yr, year(t.date))),
                      t.date);
 implies converts create, validMatchWithBuy, totalCost, isInitialized
```

D.27 The genlib Interface

```
imports <stdio>;
typedef long nat {constraint ∀ n: nat (n ≥ 0)};
typedef char cstring[] {constraint ∀ s: cstring (nullTerminated(s))};
uses genlib (cstring for String);
constant char separator_char = sep_char;

int get_line (FILE *from_file, out char output[], nat maxLength) {
  let line be getLineNChars((*from_file)^, maxLength - 1),
      inline be replaceNewlineByNull(line);
  requires canRead((*from_file)^) ∧ (maxIndex(output) > maxLength);
  modifies *from_file, output;
  ensures if peekChar((*from_file)^) = 'EOF'
          then result = char2int('EOF') ∧ unchanged(*from_file, output)
          else (*from_file)' = removeLine((*from_file)^, line)
               ∧ result = len(inline) + 1 ∧ nullTerminated(output')
               ∧ getString(output') = inline;
}
bool str_to_double (char str[], nat length, out double *res) FILE *stderr; {
  let instr be prefix(str^, length),
      fileObj be *stderr^;
  requires len(str^) ≥ length;
  modifies *res, fileObj;
  ensures (result = length > 0 ∧ okFloatString(instr))
       ∧ if result
          then (*res)' = string2double(instr) ∧ unchanged(fileObj)
          else ∃ errm: cstring (appendedMsg(fileObj', fileObj^, errm));
  claims okFloatString(".5") ∧ okFloatString("-1,339.89")
         ∧ okFloatString("1,339.89");
}
void getParts (double d, out nat *wholePart, out double *decPart)
              FILE *stderr; {
  let fileObj be *stderr^;
  modifies *wholePart, *decPart, fileObj;
  ensures if d ≥ 0
          then d = int2double((*wholePart)′) + (*decPart)′ ∧ unchanged(fileObj)
          else ∃ errm: cstring (appendedMsg(fileObj′, fileObj^, errm));
  claims (*decPart)′ < 1.0;
}
bool within_epsilon (double num1, double num2, double epsilon) {
  ensures result = (abs(num1 - num2) < abs(epsilon));
}
bool near0 (double num) {
  ensures result = (abs(num) < 0.01);
}
double min (double i, double j) {
  ensures result = min(i, j);
}
double max (double i, double j) {
  ensures result = max(i, j);
}
char *copy_string (cstring s) {
  ensures result[]′ = s^ ∧ fresh(result[]);
}
```

D.28 The date Interface

```
imports genlib;
immutable type date;
uses date (cstring for String);

bool date_parse (cstring indate, cstring inputStr, out date *d) FILE *stderr; {
  let dateStr be getString(indate^),
      fileObj be *stderr^;
  modifies *d, fileObj;
  ensures result = okDateFormat(dateStr)
          ∧ if result then (*d)′ = string2date(dateStr) ∧ unchanged(fileObj)
             else ∃ errm: cstring (appendedMsg(fileObj′, fileObj^,
                                       inputStr^ || errm));
}
date create_null_date (void) {
  ensures result = null_date;
}
nat date_year (date d) {
  checks isNormalDate(d);
  ensures result = year(d);
  claims result ≤ 99;
}
bool is_null_date (date d) {
  ensures result = (d = null_date);
}
```

```
bool date_is_LT (date d) {
  ensures result = isLT(d);
}
bool date_same (date d1, date d2) {
  ensures result = (d1 = d2);
}
bool date_is_later (date d1, date d2) {
  ensures result = (d1 > d2);
}
bool is_long_term (date buyD, date sellD, nat hp) {
  checks isNormalDate(buyD) ∧ isNormalDate(sellD);
  ensures result = (buyD ≤ sellD ∧ hp ≤ 365
                  ∧ ((year(sellD) - year(buyD)) > 1 ∨ (sellD - buyD) > hp));
}
char *date2string (date d) {
  let res be getString(result[]');
  ensures fresh(result[]) ∧ nullTerminated(result[]')
          ∧ (isNormalDate(d) ⇒ res = date2string(d))
          ∧ (isLT(d) ⇒ res = "LT")
          ∧ (isNullDate(d) ⇒ res = "null");
}

/*** claims ***/

claims century_assumption (date d) {
  ensures isNormalDate(d) ⇒ ((string2date("0/0/50") ≤ d)
                             ∧ (d ≤ string2date("12/31/49")));
}

/* This claim shows the unconventional interpretation of dates.  It is
useful for regression testing because if we change the interpretation
of year encodings, the claim is likely to fail.  It can also be useful
for test case generation. */

claims given_day_excluded (void) {
  ensures daysToEnd(string2date("12/31/93").normal) = 0
          ∧ dayOfYear(string2date("01/01/93").normal) = 1
          ∧ dayOfYear(string2date("3/1/0").normal) = 61
          ∧ dayOfYear(string2date("3/1/93").normal) = 60;
}

/* The claim emphasizes the boundary case: days_to_end does not
include given day, checks the boundary case in the defn of daysToEnd.
It is also useful for test case generation.  NB: 0 represents 2000 (a
leap year), not 1900, a non-leap year. */

claims date_formats (void) {
  ensures okDateFormat("0/0/93") ∧ okDateFormat("1/0/93")
          ∧ not(okDateFormat("13/2/92") ∨ okDateFormat("1/32/92")
                ∨ okDateFormat("1/2") ∨ okDateFormat("1/1/1993"));
}
```

D.29 The security Interface

```
imports genlib;
```

```
immutable type security;
uses security (cstring, nat);

security security_create (cstring sym) {
  ensures result = [getString(sym^)];
}
char *security_sym (security sec) {
  ensures nullTerminated(result[]') ∧ getString(result[]') = sec.sym
          ∧ fresh(result[]);
}
bool security_same (security sec1, security sec2) {
  ensures result = (tolower(sec1.sym) = tolower(sec2.sym));
}
bool security_lessp (security sec1, security sec2) {
  ensures result = sec1 < sec2;
}
bool security_is_tbill (security sec) {
  ensures result = hasNamePrefix(sec, "TBill");
}
bool security_is_tnote (security sec) {
  ensures result = hasNamePrefix(sec, "TNote");
}
bool security_is_cash (security s) {
  ensures result = isCashSecurity(s);
}
```

D.30 The lot_list Interface

```
imports genlib;
immutable type lot;
mutable type lot_list;
uses lot_list (cstring, nat);

lot lot_parse (cstring s) {
  requires len(s^) > 0 ∧ isNumeric(s^);
  ensures result = string2lot(s^);
}
bool lot_equal (lot l1, lot l2) {
  ensures result = (l1 = l2);
}
lot_list lot_list_create (void) {
  ensures fresh(result) ∧ result' = nil;
}
bool lot_list_add (lot_list s, lot x) {
  modifies s;
  ensures result = x ∈ s^ ∧ if result then unchanged(s) else s' = cons(x, s^);
}
bool lot_list_remove (lot_list s, lot x) {
  modifies s;
  ensures (result = x ∈ s^) ∧ ¬(x ∈ s')
          ∧ ∀ y: lot ((y ∈ s^ ∧ y ≠ x) ⇒ y ∈ s');
}
lot_list lot_list_copy (lot_list s) {
  ensures result' = s^ ∧ fresh(result);
}
bool lot_list_is_empty (lot_list s) {
  ensures result = (s^ = nil);
}
nat lot_list_length (lot_list s) {
  ensures result = length(s^);
}
lot lot_list_first (lot_list s) {
  requires s^ ≠ nil;
  ensures result = car(s^);
}
bool lot_list_lessp (lot_list s1, lot_list s2) {
  ensures result = (s1^ < s2^);
}
void lot_list_free (lot_list s) {
  modifies s;
  ensures trashed(s);
}

/*** claims ***/

claims lotsInvariant (lot_list x) {
  ensures ∀ e: lot (count(e, x~) ≤ 1);
}

/* This claim highlights an important property of lot_list: it is also
a set, i.e., there are no duplicate members.  It is useful (a) for
debugging lot_list_add or other future functions that insert new
members into a lot_list; (b) as a program verification lemma, esp. in
lot_list_remove: allow us to stop after the first match is found. */
```

D.31 The trans Interface

```
imports security, date, lot_list;
typedef enum {buy, sell, cash_div, cap_dist, tbill_mat, exchange, interest,
              muni_interest, govt_interest, new_security, other} kind;
immutable type trans;
constant nat maxInputLineLen;
uses trans (cstring, kind for kind);

bool trans_parse_entry (cstring instr, out trans *entry) FILE *stderr; {
  let input be prefix(getString(instr^), maxInputLineLen),
      parsed be string2trans(input),
      fileObj be *stderr^;
  modifies *entry, fileObj;
  ensures result = (okTransFormat(input)
                    ∧ transIsConsistent(parsed, parsed.kind))
          ∧ if result then (*entry)′ = parsed ∧ unchanged(fileObj)
             else ∃ errm: cstring (appendedMsg(fileObj′, fileObj^, errm));
}
trans trans_adjust_net (trans t, double newNet) {
  checks t.kind = buy ∧ newNet ≥ 0;
  ensures result = set_price(set_net(t, newNet), newNet/t.amt);
}
trans trans_adjust_amt_and_net (trans t, double newAmt, double newNet) {
  checks t.kind = buy ∧ within1(newNet/newAmt, t.price) ∧ newNet ≥ 0
         ∧ newAmt > 0;
  ensures result = set_amt(set_net(t, newNet), newAmt);
}
bool trans_match (trans t, security s, lot e) {
  ensures result = (t.security = s ∧ length(t.lots) = 1 ∧ car(t.lots) = e);
}
bool trans_less_or_eqp (trans t1, trans t2) {
  ensures result = (t1 ≤ t2);
}
double trans_get_cash (trans entry) {
  ensures if isCashSecurity(entry.security) ∧ entry.kind = buy
          then result = entry.net
          else result = 0;
}
char *trans_input (trans t) {
  ensures nullTerminated(result[]′) ∧ getString(result[]′) = t.input
          ∧ fresh(result[]);
}
char *trans_comment (trans entry) {
  ensures nullTerminated(result[]′) ∧ getString(result[]′) = entry.comment
          ∧ fresh(result[]);
}
lot_list trans_lots (trans entry) {
  ensures result′ = entry.lots ∧ fresh(result);
}
security trans_security (trans entry) {
  ensures result = entry.security;
}
kind trans_kind (trans entry) {
  ensures result = entry.kind;
}
double trans_amt (trans entry) {
  ensures result = entry.amt;
}
double trans_net (trans entry) {
  ensures result = entry.net;
}
date trans_date (trans entry) {
  ensures result = entry.date;
}
bool trans_is_cash (trans entry) {
  ensures result = isCashSecurity(entry.security);
}

/*** claims ***/

claims buyConsistency (trans t) {
  requires t.kind = buy;
  ensures t.net ≥ 0 ∧ t.amt > 0 ∧ t.price ≥ 0 ∧ length(t.lots) = 1
          ∧ within1(t.amt*t.price, t.net);
}

claims sellConsistency (trans t) {
  /* When sell′s amt is 0, it fakes capital gain in lawsuit settlements. */
  requires t.kind = sell;
  ensures t.net > 0 ∧ t.amt ≥ 0 ∧ isNormalDate(t.date) ∧ length(t.lots) ≥ 1
          ∧ (t.amt > 0 ⇒ within1(t.amt*t.price, t.net));
}
```

```
/* The above 2 claims document the constraints among the fields of a
buy and sell trans. */

claims singleLot (trans t) {
  requires t.kind = buy ∨ t.kind = cap_dist ∨ t.kind = exchange;
  ensures length(t.lots) = 1;
}

claims multipleLots (trans t) {
  requires t.kind = sell ∨ t.kind = tbill_mat;
  ensures length(t.lots) ≥ 1 ∧ uniqueLots(t.lots);
}

/* The above 2 claims document constraints on the fields of a trans.
The first is useful as a verification lemma: we need not worry about
other members on t.lots in matching sell lots to buy lots in
position_update. */

claims amountConstraint (trans t) {
  requires t.kind = buy ∨ t.kind = tbill_mat ∨ t.kind = exchange
           ∨ t.kind = cap_dist;
  ensures t.amt > 0;
}

claims trans_lots lotsInvariant;

/* The above output claim is useful for ensuring that the output of
trans_lots meets the invariant of the lot_list module. */
```

D.32 The trans_set Interface

```
imports trans;
mutable type trans_set;
mutable type trans_set_iter;
uses trans_set (cstring, obj trans_set);

trans_set trans_set_create (void) {
  ensures result' = [{}, 0] ∧ fresh(result);
}
bool trans_set_insert (trans_set s, trans t) {
  checks s^.activeIters = 0;
  modifies s;
  ensures (result = matchKey(t.security, car(t.lots), s^.val)
                    ∧ length(t.lots) = 1)
           ∧ if result then unchanged(s) else s' = [insert(t, s^.val), 0];
}
bool trans_set_delete_match (trans_set s, security se, lot e) {
  checks s^.activeIters = 0;
  modifies s;
  ensures result = matchKey(se, e, s^.val) ∧ s'.activeIters = 0
          ∧ s'.val ⊆ s^.val
          ∧ ∀ t:trans (t ∈ s^.val ⇒
                          if t.security = se ∧ car(t.lots) = e
                          then ¬ (t ∈ s'.val)
                          else (t ∈ s′.val));
}
trans_set trans_set_copy (trans_set s) {
  ensures fresh(result) ∧ result′ = [s^.val, 0];
}
void trans_set_free (trans_set s) {
  modifies s;
  ensures trashed(s);
}
trans_set_iter trans_set_iter_start (trans_set ts) {
  modifies ts;
  ensures fresh(result) ∧ ts′ = startIter(ts^) ∧ result′ = [ts^.val, ts];
}
trans trans_set_iter_yield (trans_set_iter tsi) {
  checks tsi^.toYield ≠ {};
  modifies tsi;
  ensures yielded(result, tsi^, tsi′)
          ∧ ∀ t: trans (t ∈ tsi′.toYield ⇒ result ≤ t);
}
bool trans_set_iter_more (trans_set_iter tsi) {
  ensures result = (tsi^.toYield ≠ {});
}
void trans_set_iter_final (trans_set_iter tsi) {
  let sObj be tsi^.setObj;
  modifies tsi, sObj;
  ensures trashed(tsi) ∧ sObj′ = endIter(sObj^);
}

/*** claims ***/

claims trans_setUID (trans_set s) {
  ensures ∀ t1: trans, t2: trans
           ((t1 ∈ s~.val ∧ t2 ∈ s~.val ∧ t1.security = t2.security
             ∧ t1.lots = t2.lots) ⇒ t1 = t2);
}

/* The claim says: no two trans's in a trans_set have the same
security and lots fields.  It is useful as a program verification
lemma in trans_set_delete: allows us to stop after deleting the first
matched trans. */
```

D.33 The position Interface

```
imports trans_set;
typedef struct {double capGain, dividends, totalInterest, ltCG_CY, stCG_CY,
                       dividendsCY, taxInterestCY, muniInterestCY,
                       govtInterestCY;} income;
mutable type position;
constant nat maxTaxLen;
spec nat cur_year, holding_period;
spec bool seenError;
uses position (cstring, income for income);

bool position_initMod (nat year, nat hp) nat cur_year, holding_period;
```

```
                                         bool seenError; {
  modifies cur_year, holding_period;
  ensures result ∧ ¬ seenError′ ∧ cur_year′ = year ∧ holding_period′ = hp;
}
position position_create (cstring name) {
  ensures fresh(result) ∧ result′ = create(getString(name^));
}
void position_reset (position p, cstring name) {
  modifies p;
  ensures p′ = create(getString(name^));
}
void position_free (position p) {
  modifies p;
  ensures trashed(p);
}
bool position_is_uninitialized (position p) {
  ensures result = ¬ isInitialized(p^);
}
void position_initialize (position p, trans t) FILE *stderr; bool seenError; {
  let fileObj be *stderr^;
  modifies p, seenError, fileObj;
  ensures p′ = (if t.kind = buy
                then update_buy(create(t.security.sym), t)
                else create(t.security.sym))
       ∧ if t.kind = buy ∨ t.kind = new_security
          then unchanged(fileObj, seenError)
          else seenError′
               ∧ ∃ errm: cstring (appendedMsg(fileObj′, fileObj^, errm));
}
security position_security (position p) {
  ensures result = p^.security;
}
double position_amt (position p) {
  ensures result = p^.amt;
}
void position_update (position p, trans t) nat cur_year, holding_period;
                                           bool seenError; FILE *stderr; {
  let fileObj be *stderr^,
      report be seenError′
                ∧ ∃ errm: cstring (appendedMsg(fileObj′, fileObj^, errm)),
      ok be unchanged(seenError, fileObj);
  checks p^.security = t.security;
  modifies p, seenError, fileObj;
  ensures
   if p^.lastTransDate > t.date
   then report
   else if t.kind = buy ∧ ¬ validMatch(p^, t) ∧ length(t.lots) = 1
        then p′ = update_buy(p^, t) ∧ ok
        else if t.kind = cash_div
             then p′ = update_dividends(p^, t, cur_year^) ∧ ok
        else if isInterestKind(t.kind)
             then p′ = update_interest(p^, t, cur_year^) ∧ ok
        else if validMatchWithBuy(p^, t)
             then if t.kind = cap_dist
```

```
                  then p′ = update_cap_dist(p^, t, cur_year^) ∧ ok
                  else if t.kind = tbill_mat
                       then p′ = update_tbill_mat(p^, t, cur_year^) ∧ ok
                  else if t.kind = exchange
                       then p′ = update_exchange(p^, t) ∧ ok
                  else if t.kind = sell
                       then p′ = update_sell(p^, t, cur_year^, holding_period^,
                                             maxTaxLen) ∧ ok
                       else report
             else report;
  claims ¬ (seenError′) ⇒
          ((t.kind = cap_dist ⇒ (p′.dividends ≥ p^.dividends
                                  ∧ p′.totalInterest = p^.totalInterest
                                  ∧ p′.capGain = p^.capGain))
           ∧ (t.kind = sell ⇒
               ((p′.ltCG_CY - p^.ltCG_CY) + (p′.stCG_CY - p^.stCG_CY))
                = (p′.capGain - p^.capGain)));
}
void position_write (position p, FILE *pos_file) {
  modifies *pos_file;
  ensures (*pos_file)′ = (*pos_file)^ ‖ position2string(p^);
}
void position_write_tax (position p, FILE *pos_file) {
  modifies *pos_file;
  ensures (*pos_file)′ = (*pos_file)^ ‖ position2taxString(p^);
}
trans_set position_open_lots (position p) {
  ensures fresh(result) ∧ result′ = [p^.openLots, 0];
}
double position_write_olots (position p, FILE *olot_file) {
  modifies *olot_file;
  ensures (*olot_file)′ = (*olot_file)^ ‖ position2olotsString(p^)
          ∧ result = totalCost(p^);
}
income position_income (position p) {
  ensures result = p^.income;
}
income income_create (void) {
  ensures result = emptyIncome;
}
void income_sum (income *i1, income i2) {
  modifies *i1;
  ensures (*i1)′ = sum_incomes((*i1)^, i2);
}
void income_write (income i, FILE *pos_file) {
  modifies *pos_file;
  ensures (*pos_file)′ = (*pos_file)^ ‖ income2string(i);
}
void income_write_tax (income i, FILE *pos_file) {
  modifies *pos_file;
  ensures (*pos_file)′ = (*pos_file)^ ‖ income2taxString(i);
}

/*** claims ***/
```

```
claims noShortPositions (position p) bool seenError; {
  ensures ¬(seenError~) ⇒ p~.amt ≥ 0;
}

claims okCostBasis (position p) bool seenError; {
  ensures ¬(seenError~) ⇒ (∀ t: trans (t ∈ p~.openLots ⇒ t.price ≥ 0));
}

/* The above 2 claims document the key properties of the program,
codified in the position interface: No short selling of securities is
allowed, and the cost basis of a security cannot be negative. */

claims amountConsistency (position p) bool seenError; {
  let pv be p~;
  ensures ¬(seenError~) ⇒ pv.amt = sum_amt(pv.openLots);
}

/* The above claim documents one key constraint among the different
fields of a valid position.  */

claims openLotsTransConsistency (position p) bool seenError; {
  ensures ¬(seenError~) ⇒
          ∀ t: trans ((t ∈ p~.openLots) ⇒
               ((t.amt > 0) ∧ (t.net ≥ 0) ∧ (t.price ≥ 0) ∧ (t.kind = buy)
                ∧ within1(t.amt*t.price, t.net)
                ∧ length(t.lots) = 1 ∧ (t.security = p~.security)
                ∧ (t.date ≤ p~.lastTransDate)));
}

claims uniqueOpenLots (position p) bool seenError; {
  let olots be p~.openLots;
  ensures ¬(seenError~) ⇒
          ∀ t1: trans, t2: trans
            ((t1 ∈ olots ∧ t2 ∈ olots) ⇒
               (((t1.security = t2.security) ∧ t1.lots = t2.lots) ⇒
                t1 = t2));
}

/* The above claims document the key constraints on the open lots of a
position.  They can be useful in regression testing if future changes
bundle different securities together in the open lot. */

claims distributionEffect (position p, trans t) bool seenError; {
  requires t.kind = cap_dist;
  body { position_update(p, t); }
  ensures ((findMatch(p^, t).net ≠ 0) ∧ ¬(seenError′)) ⇒
           (findMatch(p′, t).net < findMatch(p^, t).net);
}

/* The above claim highlights the essence of the cost basis
computation without giving much details: If the cost basis is
non-zero, capital distribution decreases it. */

claims distributionEffect2 (position p, trans t) bool seenError; {
  requires t.kind = cap_dist;
  body { position_update(p, t);}
  ensures ((t.net > findMatch(p^, t).net) ∧ ¬(seenError′))⇒
```

```
            p'.dividends = p^.dividends + (t.net - findMatch(p^, t).net);
}

/* The above claim illustrates the impact of a distribution
transaction: the excess amount in a distribution is considered as
dividends (not interest or capital gain). */

claims openLotsUnchanged (position p, trans t) bool seenError; {
  requires (t.kind = cash_div ∨ isInterestKind(t.kind)
            ∨ t.kind = new_security);
  body { position_update(p, t); }
  ensures ¬ (seenError') ⇒ p'.openLots = p^.openLots;
}

claims openLotsBuy (position p, trans t) bool seenError; {
  requires t.kind = buy;
  body { position_update(p, t); }
  ensures ¬ (seenError') ⇒ size(p'.openLots) = size(p^.openLots) + 1;
}

claims openLotsSell (position p, trans t) bool seenError; {
  requires t.kind = sell ∨ t.kind = exchange;
  body { position_update(p, t); }
  ensures ¬ (seenError') ⇒ size(p'.openLots) = size(p^.openLots) - 1;
}

claims openLotsTbill (position p, trans t) bool seenError; {
  requires t.kind = tbill_mat;
  body { position_update(p, t); }
  ensures ¬ (seenError') ⇒ size(p'.openLots) ≤ size(p^.openLots);
}

/* In the position trait, we specify how each transaction kind affects
each field of a position.  The above 4 claims provide a complementary
view: we highlight how each field of a position is affected by
different transaction kinds.  It illustrates the technique of
describing the specification in different and complementary ways: they
can help clients understand the specs and help the specifier catch
specification errors.  */

claims taxUnchanged (position p, trans t) nat cur_year; bool seenError; {
  requires year(t.date) ≠ cur_year^ ∨ t.kind = exchange;
  body { position_update(p, t); }
  ensures ¬ (seenError') ⇒
           (p'.ltCG_CY = p^.ltCG_CY ∧ p'.stCG_CY = p^.stCG_CY
            ∧ p'.dividendsCY = p^.dividendsCY
            ∧ p'.taxInterestCY = p^.taxInterestCY
            ∧ p'.muniInterestCY = p^.muniInterestCY
            ∧ p'.govtInterestCY = p^.govtInterestCY);
}

/* Transactions that do not occur in the current year does not change
our current year tax position.  An exchange does not change our tax
position either. */
```

Bibliography

[1] American National Standards Institute. *American National Standard for Information Systems – Programming Language C*, 1989. X3.159-1989.

[2] W. Bartussek and D.L. Parnas. Using assertions about traces to write abstract specification for software modules. In *Proc. 2nd Conf. European Cooperation in Informatics, LNCS 65*, pages 211–136. Springer-Verlag, 1978. Also in Software Specification Techniques, N. Gehani and A.D. McGettrick (ed.) Addison-Wesley, 1986.

[3] Jolly Chen. The Larch/Generic interface language. S. B. Thesis, Department of Electrical Engineering and Computer Science, MIT, 1989.

[4] Yoonsik Cheon and Gary Leavens. The Larch/Smalltalk: A specification language for Smalltalk. TR 91-15, Iowa State University, June 1991.

[5] David Evans. Using specifications to check source code. Master's thesis, MIT EECS Dept., May 1994.

[6] Bob Fields and Morten Elvang-Goransson. A VDM case study in *mural*. *IEEE Transactions on Software Engineering*, 18(4):279–295, April 1992.

[7] Stephen J. Garland and John V. Guttag. A guide to LP, the Larch Prover. Report 82, DEC Systems Research Center, Palo Alto, CA, December 1991.

[8] Stephen J. Garland, John V. Guttag, and James J. Horning. Debugging Larch Shared Language specifications. *IEEE Transactions on Software Engineering*, 16(9):1044–1075, September 1990. Reprinted as DEC Systems Research Center Report 60. Superseded by Chapter 7 in [15].

[9] Narain Gehani and Andrew McGettrick, editors. *Software Specification Techniques*. Addison-Wesley, 1986.

[10] Christopher Paul Gerrard, Derek Coleman, and Robin M. Gallimore. Formal specification and design time testing. *IEEE Transactions on Software Engineering*, 16(1):1–12, January 1990.

[11] Michael Gordon. *HOL: A Proof Generating Systems for Higher-Order Logic*, chapter in VLSI Specification, Verification and Synthesis, pages 73–129. Kluwer, 1988.

[12] David Guaspari, Carla Marceau, and Wolfgang Polak. Formal verification of Ada. *IEEE Transactions on Software Engineering*, 16(9):1058–1075, September 1990.

[13] John V. Guttag and James J. Horning. Formal specification as a design tool. In *7th ACM Symposium on Principles of Programming Languages*, pages 251–261, Las Vegas, January 1980. Reprinted in [9].

[14] John V. Guttag and James J. Horning. LCL: A Larch interface language for C. Report 74, DEC Systems Research Center, Palo Alto, CA, July 1991.

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

[16] David Hinman. On the design of Larch interface languages. Master's thesis, Department of Electrical Engineering and Computer Science, MIT, January 1987.

[17] Shigeru Igarashi, Ralph L. London, and David C. Luckham. Automatic program verification I: logical basis and its implementation. *Acta Informatica*, 4(2):145–182, 1975.

[18] Daniel Jackson. Composing data and process descriptions in the design of software systems. TR 419, MIT Lab. for Computer Science, May 1988.

[19] Daniel Jackson. Aspect: A formal specification language for detecting bugs. TR 543, MIT Lab. for Computer Science, June 1992.

[20] S.C. Johnson. Lint, a C program checker. Unix Documentation.

[21] C.B. Jones, K.D. Jones, P.A. Lindsay, and R. Moore. *mural: A Formal Development Support System.* Springer-Verlag, 1991.

[22] Cliff B. Jones. *Systematic Software Development Using VDM 2nd ed.* Prentice-Hall International, 1990.

[23] Richard A. Kemmerer. Testing formal specifications to detect design errors. *IEEE Transactions on Software Engineering*, 11(1):32–43, January 1985.

[24] Brian W. Kernighan and Dennis M. Ritchie. *The C Programming Language, 2nd ed.* Prentice Hall, 1988.

[25] Gary T. Leavens and Yoonsik Cheon. Preliminary design of Larch/C++. In Ursula Martin and Jeannette M. Wing, editors, *First International Workshop on Larch.* Springer-Verlag, July 1992.

[26] B. Liskov, R. Atkinson, T. Bloom, E.M. Moss, J.C. Schaffert, R. Scheifler, and A. Snyder. *CLU Reference Manual.* Springer-Verlag, 1981.

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

[28] David C. Luckham, Sriram Sankar, and Shuzo Takahashi. Two-dimensional pinpointing: Debugging with formal specifications. *IEEE Software*, 8(1):74–84, January 1991.

[29] David C. Luckham and Friedrich W. von Henke. An overview of ANNA, a specification language for Ada. *IEEE Software*, 2(3):9–22, March 1985.

[30] Keith W. Miller, Larry J. Morell, and Fred Stevens. Adding data abstraction to Fortran software. *IEEE Software*, 5(6):50–58, November 1988.

[31] Greg Nelson, editor. *Systems Programming with Modula-3.* Prentice Hall, 1991.

[32] Piotr Rudnicki. What should be proved and tested symbolically in formal specifications? In *Proc. 4th Int. Workshop on Software Specifications and Design*, pages 42–49, April 1987.

[33] J.M. Spivey. An introduction to Z and formal specifications. *Software Engineering Journal*, January 1989.

[34] Bjarne Stroustrup. *The C++ Programming Language.* Addison-Wesley, 1991.

[35] Yang Meng Tan. Semantic analysis of formal specifications. MIT EECS Dept PhD thesis proposal, November 1992.

[36] Mark T. Vandevoorde. Exploiting formal specifications to produce faster code. MIT EECS Dept. PhD thesis proposal, March 1991.

[37] Mark T. Vandevoorde. Exploiting specifications to improve program performance. TR 598, MIT Lab. for Computer Science, February 1994. PhD Thesis, EECS Dept.

[38] Jeannette M. Wing. A two-tiered approach to specifying programs. TR 299, MIT Lab. for Computer Science, May 1983. PhD Thesis, EECS Dept.

[39] Jeannette M. Wing. Writing Larch interface language specifications. *ACM Transactions on Programming Languages and Systems*, 9(1):1–24, January 1987.

[40] Jeannette M. Wing, Eugene Rollins, and Amy Moormann Zaremski. Thoughts on a Larch/ML and a new application for LP. In Ursula Martin and Jeannette M. Wing, editors, *First International Workshop on Larch.* Springer-Verlag, July 1992.

[41] Kaizhi Yue. What does it mean to say that a specification is complete? In *Proc. 4th Int. Workshop on Software Specifications and Design*, pages 42–49, April 1987.

[42] P. Zave. An operational approach to requirements specification for embedded systems. *IEEE Transactions on Software Engineering*, 8(3):250–269, May 1982.