On the Language Design and Semantic Foundation of LCL, a Larch/C Interface Specification Language

Patrice Chalin

A Thesis
in
The Department
of
Computer Science

Presented in Partial Fulfillment of the Requirements
For the Degree of Doctor of Philosophy
Concordia University
Montréal, Québec, Canada

October 1995

ISBN 0-612-10829-5

Abstract

On the Language Design and Semantic Foundation of LCL, a Larch/C Interface Specification Language

Patrice Chalin, Ph.D.
Concordia University, 1996

The specialization of a specification language to a particular programming language is an important characteristic of module interface specification languages (MISL's). The only well-developed MISL's are the Larch interface languages and among these LCL, a Larch/C interface specification language, would seem to be the most mature.

Our efforts to elaborate a semantic model for LCL lead to the identification of inadequacies and insufficiencies in the language and its informal definition. After defining and motivating the concept of object dependency, we demonstrate that LCL lacks the necessary language constructs for specifying object dependency relationships. We illustrate shortcomings caused by implicit constraints that are related to function parameters and object trashing. We show that the implicit constraint associated with the trashing of objects results in a violation of the principle of referential transparency.

The identified inadequacies and insufficiencies are overcome in LCL′, the variant of LCL described in this thesis. The main contribution of this thesis is a semantic model within which a core subset of LCL′ (consisting of constant declarations, variable declarations and function specifications) is formally defined. We present the semantics in a style known as natural semantics. The meaning of the non-interface part of an LCL′ specification is captured by an embedding into LSL. The primary notation used to write the semantics is Z. We have chosen to use LL, the logic underlying LSL, as the logical basis for LCL′. At the heart of the semantic model is our model of the store. The storage model is exceptional in that it supports object dependencies in their full static generality. Previously published definitions of the meaning of a function specification are shown to be inaccurate; we present a corrected definition.

Finally, we note that our semantic model (particularly the model of the store) is general enough that it can serve as a base for the formal definition of other imperative

programming languages and MISL's—especially the Larch interface languages LCPP and LM3.

Acknowledgments

Foremost, I would like to gratefully acknowledge the amicable guidance and support that Peter Grogono, my thesis supervisor and mentor, has provided during my years at Concordia. I am also thankful for the helpful comments that he has made on the many drafts of the thesis work as it was taking shape.

I am grateful to T. Radhakrishnan for accepting to be my co-supervisor. His insistence that I justify my claims and think about my work in a broader context has resulted in a clearer exposition.

I thank Yan Meng Tan for answering my initial queries concerning LCL and his thesis. David Evans and Stephen Garland have also been helpful in answering my questions about LCLint, the LSL checker and the Larch Prover. I very thankful to Gary Leavens and David Evans for their comments on a earlier version of the shortcomings section and for the many discussions that insued. Their remarks have lead to improvements in the thesis.

Thanks is due to our analyst pool for doing an exceptional job at maintaining and upgrading the computer systems in our department; particularly, Stan Swiercz, Michael Assels and Paul Gill.

Many people who are close to me have helped me to grow and to put the "rest" of my life in perspective while I was working towards my degree. To my Temagami family I say: I'm finally done! I thank Evangelia for our many discussions, from artwork to the zodiac passing through dreams and parenting. Finally, I thank Sylvie, Maude and Yan Eric for the sacrifices that they have made, for their patience and for never ceasing to educate me on how to become a better *humane* being.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

"...a robust program [or specification] written in an insecure language is like a house built upon sand." [MTH90, p. vii]

Formal methods are mathematically based techniques that can be used to model, design, and analyze computer based systems [CGR93]; they are seen as the applied mathematics of computer systems engineering [Cra89]. Although formal methods are immature in some important aspects, it is believed that an appropriate use of formal methods can contribute to the cost-effective and timely production of systems of the highest quality [BH94]. Recent studies support this claim: increasingly, formal methods are being successfully applied to the development of industrial-scale projects [CGR93].

Underlying every formal method is a mathematically based notation—usually a specification language. For a specification language to be suitable for use in an industrial setting it must be

- expressive
- precisely defined
- free from errors and inconsistencies
- supported by appropriate tools.

Languages are not simply defined and then used, they are subject to evolution. Language evolution—as in the evolution of many complex engineered products--is a long and laborious process in which initial inception is followed by cycles of use, assessment and change. As in any engineering discipline, assessment is best achieved by the use of mathematically based techniques.

In this thesis we have used mathematical methods to increase our understanding of LCL, a Larch interface specification language for ISO C [ISO]. We have done so by attempting to create a semantic model for LCL. Our initial analysis of the language brought to light inadequacies in the definition and insufficiencies in the expressiveness of LCL. To date, LCL does not have a formal semantics and hence it precariously holds the status of being a *formal* specification language. We propose changes to the language that overcome the identified inadequacies and insufficiencies, and we provide a semantic model in which a core subset of the language is formally described.

1.1 Larch

The Larch approach to specification promotes the modular development of programs and encourages the use of data abstraction. In Larch there are two specification tiers or levels. The *shared tier* contains specifications (called *traits*) written in the *Larch Shared Language* (LSL) [GH93]. A trait defines a multisorted first-order theory. The *interface tier* contains interface specifications written in a Larch interface language. There are several Larch interface languages. The most widely used are LCL [GH93], LCPP (an interface language for C++) [LC95] and LM3 (an interface language for Modula-3) [GH93]. Each[1] interface language is specialized for use with a particular programming language. Using constructs, concepts and terminology from the programming language, an interface specification describes what resources are being provided by a module. Although the shared tier is independent of the interface tier the opposite is not true. The meanings of the (concrete and abstract) data types used in a module are captured in the form of LSL traits. That is, traits are used to provide formal theories for the data types that are used in interface specifications.

1.2 Module Interface Specification Languages

Specification languages can be used during the entire software development process to document requirements, designs and the interface specifications for modules and

[1]With the exception of the two generic interface languages GIL [Che89] and GCIL [Ler91].

program components. One must be careful in choosing an appropriate specification language for the task at hand [Hoa87, BH94]. The specialization of a specification language to a particular programming language is an important characteristic of *module interface specification languages* (MISL's). After some preliminary definitions, we clarify the role of interface in MISL's and we argue that there are very few MISL's.

1.2.1 Specifications

There are many kinds of specification. We are concerned with specifications that describe the functional (behavioral) characteristics of computer-based systems. Specifications can describe many different kinds of system, ranging from independent or embedded computer-based systems to individual statements within a software component.

A *specification* is a description of [PM91, PST91, Jon90]:

- an interface, and
- externally observable behaviors of a system as visible at its interface.

A *formal specification* is a specification written in a formal language. Note that under the given definition, a program is a specification.

A system I is said to *satisfy* a specification S if:

- the interface of I satisfies the interface of S, and
- the behaviors of I are among the behaviors permitted by S.

The 'satisfies' relation for specifications is defined in terms of the more primitive 'satisfies' relation for interfaces. The latter is often defined as the identity relation—i.e. I satisfies the interface of S if and only if they have the same interface. When I satisfies S we may say that I is an *implementation* of S, or that I is a *refinement* of S and write $S \sqsubseteq I$.

1.2.2 On the Importance of Interface

The interface component of a specification is often taken for granted and even forgotten. Yet, it is a very crucial part of a specification [Lam89]. In [Mer74], "interface" is defined as:

"the place at which two independent systems meet and act on or communicate with each other."

Thus, the interface part of a specification is a description of the *boundary* between the system and its environment. All interactions between a system and its environment occur *at* the interface between them. The interface part of a specification:

- prescribes the kinds of communication that are possible between a system and its environment, and
- for each kind of communication, identifies what (i.e. the system or the environment) can initiate a communication.

The interface allows us to determine what kind of system is being described [Lam89]. The importance of interface becomes evident when we recognize that different kinds of system can exhibit the same behavior. In such cases the behavioral component of a specification becomes insufficient to determine which kind of system is being described. Consider, for example, all systems that do nothing. These all have identical behaviors, but a computer system that does nothing is certainly quite different from a program statement that does nothing. The interface of a specification must allow us to determine which system the specification is meant to describe.

1.2.3 MISL's and Other Specification Languages

Most formal specification languages are general purpose languages. Among the most popular are VDM-SL [Jon90] and Z [Spi92]. These languages are best suited for design specification. Even the wide-spectrum languages COLD-K [FJKRdL89] and RAISE [NHWG89] are meant for expressing designs and design refinements. If any of these languages were to be used as an MISL, an interface refinement relation would have to be defined. Even so, it is unlikely that programming language specific features could be captured in these design specification languages. Another approach is used with the B Method [Abr91]: in this case the programming language used is specially tailored to fit into the overall development scheme.

Much less attention has been given to MISL's by the research community than to design or wide-spectrum specification languages. To our knowledge, the only MISL's are the Larch interface languages and an adaptation of the language used with the

Trace Assertion Method (TAM) [PW89]. Of the Larch interface languages, LCL would seem to be the most developed and used. Development of the TAM-based language is at a preliminary stage and a new release is in preparation [IMPK93].

1.3 MISL's and Industry

Although this thesis is quite theoretical, the research has been motivated by a practical concern for industry. We believe that MISL's are an excellent way of introducing formal methods into industrial settings. It is particularly important to industry that start-up costs be minimized and that benefits be apparent even with small investments. We believe that MISL's can offer this. Some of the advantages of the use of MISL's are enumerated next[2].

MISL's can be immediately and 'unintrusively' integrated into current industrial development processes [WZ91]. A company that has invested considerable resources in the creation and installation of their development processes (e.g. training of personnel and construction of tools) is more likely to welcome formal methods that can be used in conjunction with inhouse development standards.

One of the greatest challenges faced by industry is the maintenance of legacy code. Not only can MISL's be applied to new developments, they can also be integrated into the maintenance cycle of existing software systems. This is of great value since it means that formal methods can be retroactively brought into projects that were developed without formal methods.

MISL's can be *gradually* integrated into a project:

- they can be applied to isolated portions of a system (such as those aspects for which reliability is most critical); they need not be applied throughout the system.

- MISL's can be applied to varying degrees of rigor: from merely documenting function signatures to providing complete behavioral descriptions for functions. At all levels one can reap benefits.

Tool support for most other classes of specification language is limited to type checking; in some cases proof assistants can be used. More automated checks can be

[2]These advantages are not necessarily exclusive to MISL's—they may be shared by other classes of specification language.

performed for MISL's. For example, the LCLint tool can be used on LCL specifications to detect abstraction boundary violations, illicit access to global variables, and undocumented modification of client-visible objects [EGHT94]. As another example, Vandevoorde has developed a prototype program optimizer that makes use of the information derived from module interface specifications to perform optimizations that cannot be accomplished by the analysis of code alone [Van94].

1.4 Why a Formal Semantics?

The main goal of the research reported in this thesis has been to provide a formal semantics for LCL. What are the advantages of having a formal semantics for a specification language?

A formal semantics provides the foundation that will allow for the rigorous (and, if necessary, formal) analysis of specifications. In general a specification cannot be verified [GH93, p. 41,p. 121] (i.e. shown to satisfy another 'more abstract' specification), it can only be validated. The principal (and in most cases the only[3]) means of validating a specification is by proving that it has certain desirable properties.

A formal semantics provides the necessary framework for establishing a precise definition of correctness (of implementations) and the means by which correctness can be established [Spi88, §1.2].

A formal language definition provides the necessary framework within which the language definition itself can be analyzed. Thus, properties of the language can be proven (e.g. well-formedness, compositionality and certain forms of consistency), and language design alternatives or proposed extensions can be more rigorously evaluated and compared [MT91].

Modularity is our weapon against complexity. It allows us to decompose a system, whose specification is S, into a collection of modules, described by the specifications $S_1, \ldots, S_n$. Development of each of the S_i can proceed independently—resulting, say, in the implementations I_i. But what assurance do we have that when we put the parts together they will work as a whole as expected—i.e. that the combined I_i will satisfy S? A method that allows us to assert the correctness of the whole from the correctness of the parts is said to be *compositional* [Jon93]. In industrial

[3]The other means of validating a specification is by testing, but this is possible only if the specification is executable. Generally, specifications are not executable.

settings, where we face programming-in-the-large, compositionality is an essential characteristic of a modular development method. The formal definition of a language allows us to prove (or refute) the claim that a language is compositional.

The *process* of elaborating a formal semantics is just as important as the end result (namely, the formal semantics itself) since the process provides us with a deeper understanding of the language and it often allows us to become aware of subtleties in its definition of which it is good to be aware [Win93, p. xv]. The evolution of a language is best achieved by a close interplay between language design, formal definition, and use. It has also been argued that a semantic definition should be used prescriptively rather than descriptively: that is, the language design should be *guided by* the underlying semantic model [AW82]. Defining a language around a coherent semantic model (in contrast to developing a model for a 'given' language) should result in a more coherent language because, the understanding of a language comes from an understanding of its underlying model. Since we think and reason about a language by means of its semantic model, a simpler model will result in a language that is easier to use in practice. When writing specifications, simplicity is our best defense against errors.

A formal semantics can be used to validate a program calculus or an algebra of programs—e.g. Morgan's Refinement Calculus [Mor90] and the "Laws of Programming" of Hoare *et al* [HHJ+87]—that permits program development and program optimization by transformation [Hoa94] . It is more practical to make use of such rules or laws to reason about programs than it is to appeal directly to the semantic model.

Finally, the soundness of checkers (such as LCLint [EGHT94]) or tools that manipulate specifications (e.g. a tool translating LCL specifications into LP scripts) can be ascertained only if the specifications have a formally defined semantics [Spi88, §1.2].

1.5 Contributions

The principal contribution of this thesis is a semantic model within which the stable aspects of the LCL language can be formally documented and alternatives for the contentious aspects of the language can be formally expressed and evaluated. Other major contributions of this thesis are:

- an identification of shortcomings in the LCL language and its definition,
- solutions to the identified shortcomings,
- a formal semantics for a core—consisting of constant and variable declarations and function specifications over basic and array types—of the stable aspects of LCL.

1.6 Related Work

The conventional approaches to programming language semantics include operational, axiomatic, algebraic and denotational semantics. An operational semantics describes the behavior of the constructs of a programming language in terms of the effect their execution would have on an abstract machine [Hoa94, Chapter 4]. The meaning of a construct is taken to be the set of all possible behaviors that it can exhibit on the abstract machine or, the abstract machine code to which the construct corresponds. An operational semantics can be presented in several different styles. *Natural semantics* [Kah87] and *structural operational semantics* are two styles in which the semantics is presented in the form of inference rules [SK95, Chapter 8]. An axiomatic semantics defines the meaning of a program by means of a proof system that allows us to deduce assertions of the form

$$\{P\}\ S\ \{Q\}$$

where P and Q are predicates and S is a command in the programming language. In words, such an assertion means: if S is executed when the program state satisfies P, then Q will be true when S terminates (provided it terminates). In the algebraic approach, equations and inequations are used. This makes it ideal for use in program development by transformation as well as program optimization [Hoa94]. A denotational semantics compositionally maps each syntactic object into an object from an appropriate semantic domain.

The various approaches are not in opposition to each other, they are complementary. The distinction between some of these approaches can be fuzzy, e.g. natural semantics can be seen a stylistic variant of denotational semantics [SK95, p. 262]. In his paper on "Unified Theories of Programming", Hoare has shown that the denotational, algebraic and operational approaches are equally expressive [Hoa94].

Relatively few programming languages (that are in wide-spread use) are formally defined. Of those that are, most are incompletely defined; often "problematic" language features (such as aliasing) are excluded from the definition. For example, the early published axiomatic semantics of Pascal [HW73] does not cover functions with side-effects, aliasing or **goto** statements. Similarly, the semantics of the Turing language [HMRC88] is defined under the assumptions that functions are side-effect free and aliasing does not occur, yet programmers are not prevented from making use of these features[4]. Hence programmers must either program in a restrictive[5] subset of the language or write programs for which there is no semantics.

Standard ML is a notable example of a language for which a complete and formal definition, in the form of a natural semantics, has been given [MTH90]. Although some errors have been found in the definition, solutions have also been proposed [Kah93]. New approaches to programming language semantics have been applied to several languages, but the completeness and accuracy of these definitions have not been assessed. For example, evolving algebras have been used to describe the semantics of Ada, C [GH92], C++, Modula-2, Occam and Prolog [Hug95]. Action semantics [SK95, Chapter 13] of Pascal and Standard ML have also been published.

It is also the case that relatively few specification languages are formally defined. Of those that are, Z has been given a denotational semantics [Spi88] and COLD-K is formally defined in a style known as translational or transformational semantics [FJKRdL89].

In Section 1.2.3 we discussed module interface specification and design specification languages. We stated that the only well-developed module interface specification languages are the Larch languages. None of the Larch interface languages have a formal semantics. Rigorous (and partial) semantic definitions exist for LCL, LCPP, and LM3. LCL is informally described in the Larch book [GH93, Chapter 5]. The most complete semantics published for LCL (actually, for any Larch interface language) is in Tan's PhD thesis [Tan94, Chapter 7]. LCPP is a Larch interface language for

[4]The Turing compiler does not check for aliasing and potential side-effects in functions even though the authors of Turing conceded that it would [HMRC88, p. 20].

[5]E.g. any set of rules used to detect potential side-effects in functions would prevent programmers from writing legitimate functions since it is impossible, in general, to assess from inspection of a program alone whether certain side-effects are benevolent or not.

C++ designed by Leavens and Cheon. The syntax and informal semantics for LCPP are given in the LCPP Reference Manual [LC95]. LM3 is a Larch interface language for Modula-3 whose principal designer was Kevin Jones. The syntax and an informal semantics of LM3 were first published in [Jon91]. LM3 is also described in the Larch book [GH93, Chapter 6]. In a more complete version of the semantics of LM3 [Jon92], Jones defines the meaning of an LM3 specification by means of rules for translating the specification into an LSL trait. A technical assessment of the semantics of LCL, LCPP and LM3 is given in the related work sections of subsequent chapters.

Chapter 2

LCL

2.1 Relationship between LSL and LCL

We find it helpful to symbolize the relationship between LCL and LSL by the equation

$$\text{LCL} = \text{CISL}(\text{LSL}) \qquad (1)$$

We can view LCL as consisting of a C interface specification language (CISL) that makes use of LSL as a sublanguage. LCL expressions can in fact be viewed as LSL terms. The equation suggests that we could "instantiate" the CISL with another language [GH93, §8]. In fact, our very preliminary research efforts were dedicated to the creation of CISL(Z). We felt it necessary, though, to establish a precise semantics for LCL as a first step. Consequently, CISL(Z) is now beyond the scope of the thesis.

Each LCL type is *based on* or *associated with* an LSL sort [GH93, p.21, p.58]; if the type T is based on the sort S we also say that S is the *sort of* T. For example, `int` and `Arr[int]` are the respective sorts of the types `int` and `int[3]`. If T is an abstract type then the sort of T is also named T. For any sort S, an object containing values of sort S will be of sort `Obj[S]`[1]. `Obj[S]` is called the *object sort* of S. If the sort of the type T is S, then the *object sort* of T is the object sort of S[2].

[1]`Obj[S]` is simply a composite LSL sort name; it does not represent a parameterized sort.

[2]Tan calls S the *value sort* of T [Tan94]. Our definition of "object sort" is more general than that used by Tan.

2.2 Organization of a Module's Interface and Implementation: C *vs.* LCL

Under the usual C programming conventions, a C module `M` is documented as a pair of files: a header file `M.h` and a main source file `M.c`[3]. The header file `M.h` is meant to document the module interface and `M.c` holds the implementation. Unfortunately, it is often necessary (due to the way the C language is defined) to include part of the implementation in the header file. Since `M.h` is used as the principal source of documentation for `M`, this scheme violates the principle of separation of concerns and increases the likelihood that clients will make use of implementation details that are meant to be private to `M`.

By using LCL, developers can provide a clean separation between a module's interface and its implementation. LCL also allows complete documentation of the module's functional properties. The interface of a module `M` is expressed as an LCL specification conventionally contained in the file named `M.lcl`. Using LCLint, one can generate from `M.lcl` the include file `M.lh` containing C declarations for the components exported by `M`. The implementation of `M` is contained in `M.h` and `M.c`. The programmer need not repeat the declarations of the components exported by `M` since they are present in `M.lh`; instead `M.lh` is included by `M.h`. Clients refer to the specification of `M` for documentation, and client code continues to include `M.h`.

In addition to the checks performed by the traditional Unix `lint` program verifier, LCLint also performs stricter type checking. In particular LCLint will report illegal access, by a client, to the representation of an abstract type. By making use of LCL and LCLint, one can achieve a level of type security that is comparable to that of Ada.

2.3 Brief Review of LCL

By means of examples, this section provides a brief overview of the main characteristics of LCL. This is not a tutorial on LCL; for this purpose, readers may consult the Larch book [GH93].

[3]For simplicity we assume that the main source of the module is contained in one file. This need not be the case.

2.3.1 Example: QueueOne

The interface specification for a module named QueueOne is given in Figure 1. The

```
constant int MAX = 100;
spec int saved;

int q1(int x) int saved; {
   requires x < MAX;
   modifies saved;
   ensures  result = saved^
            ∧ saved' = x;
}
```

Figure 1: LCL Specification QueueOne

module declares an integer constant MAX, an integer global variable **saved**[4] and a function q1.

A constant declaration defines the type and, optionally, the value of a constant. Global variables are declared as in C (but without the keyword **extern**). A function declaration is provided by means of a *function specification* which consists of a *header* and a *body*. A header is (essentially) a C function prototype optionally followed by a list of global variable declarations. The function prototype identifies the name and types of the function parameters as well as the name and return type of the function. The global variable list identifies which global variables the function implementation is permitted to access. Thus, q1 accepts a single integer parameter and yields an integer value. The implementation of q1 can access only **saved**[5].

A function specification body documents the permitted behavior of a function. The behavior of a function is described relative to two states: the state before the function is entered, called the *pre-state*, and the state after the function returns, called the *post-state*. Most function specifications contain requires, modifies and ensures clauses. From the point of view of a client, a function should be invoked only when the program state satisfies the predicate in the requires clause. Given that a client respects this obligation, the function will terminate in a state that satisfies the predicate in the ensures clause. Furthermore, the only client-visible objects that

[4]The **spec** attribute is discussed later in this section.

[5]Although, in this case, **saved** happens to be the only global variable declared in the specification.

may have changed value are those objects referenced in the modifies clause. If a function is invoked when the pre-state does not satisfy the requires clause predicate, then the behavior of the function is unconstrained. All of the clauses in a function specification body are optional. Omitting either the requires or the ensures clauses is equivalent to including the clause with the predicate `true`. An omitted modifies clause is equivalent to the clause `modifies nothing`—which specifies that no client-visible object may be modified by the function.

The requires predicate of `q1` states that the function parameter `x` should be less than `MAX`. The expressions e^ and e' denote the values contained in the object (referred to by the subexpression) e in the pre-state and post-state respectively[6]. The pseudovariable `result` denotes the value returned by the function. Therefore, the result of `q1` will be the value of `saved` in the pre-state and the post-state value of `saved` will be changed to `x`.

Any declaration can be preceded by the **spec** keyword. A module only exports the non-**spec** components that are documented in its interface, therefore, client code cannot make use of **spec** components. **spec** components are used as an aid in specifying the overall functionality of a module. The module `QueueOne` exports two components: `MAX`, and `q1`.

2.3.2 Types

C types are called *exposed* types in LCL. They are so named because the representation of the type (i.e. the type itself) is "exposed". This is in contrast to *abstract* types, which are also supported by LCL. The representation of an abstract type is hidden from its clients[7].

There are two kinds of abstract type: immutable and mutable. Immutable abstract types are treated in a way that is similar to exposed types. Mutable abstract types provide for a more object-oriented style of programming [GH93, p. 59]. Given the declarations in Figure 2, the differences between immutable and mutable types are highlighted in Table 1. (T, in Figure 2, represents an arbitrary exposed type and S, in Table 1, is the sort of T.) The first column of the table consists of expressions

[6] _^, _' are synonyms for _\pre and _\post respectively.

[7] In general, it is not possible to hide the representation of an abstract type in C, but use of LCL and its conventions allows programmers to detect, and hence prevent, illicit access to the representation of an abstract type.

involving the identifiers declared in Figure 2. Just as e^ and e' denote the value of the object e in the pre- and post-states, $e^{\bullet}$ denotes the value of the object e in an arbitrary (generic) state. _• is a synonym for _\any. The second column shows the LSL sort associated with the expression from the first column and in the same row.

```
immutable type I;
mutable type M;

constant T ct;
constant I ci;
constant M cm;

T vt;
I vi;
M vm;
```

Figure 2: Sample LCL Constant and Variable Declarations

LCL Expression	LSL Sort
ct	S
ci	I
cm	Obj[M]
cm^, cm′, cm•	M
vt	Obj[S]
vi	Obj[I]
vm	Obj[Obj[M]]
vt^, vt′, vt•	S
vi^, vi′, vi•	I
vm^, vm′, vm•	Obj[M]

Table 1: Sorts of Sample LCL Expressions

Notice that cm, a constant of the mutable abstract type M, denotes an *object* containing values of sort M. The "value" of cm is an object and this "value" is invariant: cm will always denote the same object (although the value contained in the object may change since this value depends on the program state). Also note that vm is an object that can contain objects containing values of sort M. Thus, the "values" of a mutable abstract type M are objects containing values of sort M.

2.3.3 Example: Queue

The Queue LCL specification is given in Figure 3. Queue is declared to be a mutable abstract type, and Elt as an abbreviation (called a typedef name) for int. The LSL theory for Queue's is defined in the Queue trait given in the Larch book [GH93, p. 171].

```
mutable type Queue;
typedef int Elt;

uses Queue(Elt for E, Queue for C);

Queue Create_Queue(void) {
  ensures  result' = empty ∧ fresh(result);
}

void Dispose_Queue(Queue q) {
  modifies q;
  ensures  trashed(q);
}

void Enqueue(Queue q, Elt e) {
  modifies q;
  ensures  q' = append(e,q^);
}

Elt Head(Queue q) {
  requires q^ ≠ empty;
  ensures  result = head(q^);
}
```

Figure 3: Queue LCL Specification

Modules defining an abstract type typically have creator and destructor functions. A creator function, as the name implies, creates an instance of the abstract type. A destructor function is applied to an object when it is known that the object will no longer be needed. Usually, destructors deallocate the storage associated with the object.

Create_Queue is the sole creator function for Queue module. The value contained in the created instance is the empty queue. An occurrence of the expression fresh(e) in an ensures clause guarantees that the object e is not aliased to any object that was visible to a client in the pre-state [GH93, pp. 76–77]. Hence, the instance created by

Create_Queue is not aliased to any client-visible object. The expression trashed(e) is used in an ensures clause to state that the object e can no longer be reliably accessed by a client—usually because (in the implementation) the storage associated with e is reclaimed. If a client attempts to access a trashed object, then the program behavior is undefined [GH93, p. 76]. The destructor Dispose_Queue disposes of its argument q. Enqueue adds the element e to the head of the queue q and Head yields the element that is at the head of the nonempty queue q without altering the value of q.

2.4 Shortcomings of LCL 2.4

Preliminary work on the semantics of LCL brought to light errors, omissions and inconsistencies in the language and its intended interpretation. The purpose of this section is to document those shortcomings that can be understood without detailed knowledge of the semantics of LCL. Solutions to the shortcomings are discussed in Section 3.2.

2.4.1 Dependencies Between Objects

In this section we introduce the concept of object dependency and describe how dependencies can arise. We argue that programmers rely on certain "desirable" kinds of dependency and that they tend to overlook other "less desirable" forms. Our examples will serve to illustrate that LCL lacks operations that would allow specifiers to document and reason about dependency relationships in interface specifications.

2.4.1.1 Definitions

In C, an object is a region of data storage consisting of a contiguous sequence of storage units [ISO, p. 2]. In LCL, the term is used in a more abstract sense (in particular because of the need to model objects that are instances of abstract types): an *object* is a container for values of a particular type [GH93, p. 59].

Informally we say that an object x_1 *depends on* an object x_2 if changing the value contained in x_2 may affect the value contained in x_1. It is possible for x_1 to depend on x_2 without x_2 depending on x_1[8]. If x_1 depends on x_2 or x_2 depends on x_1, then we

[8]This kind of asymmetry may exist between instances of an abstract type.

say that a dependency exists between x_1 and x_2. If x_1 is not dependent on x_2, then we say that x_1 *is independent of* x_2. The objects in a given collection are *independent*, if each object from the collection is independent of every other object in the collection. Given an expression e that refers to an object—e is called an *lvalue* in C—we shall often lighten our prose by speaking of "the object e" instead of the more verbose but precise "the object referred to by e". Thus, for example, we may state that e_1 and e_2 are independent by which we mean that the objects that are denoted by the expressions are independent. As a consequence, we note that if e_1 and e_2 are independent then the expressions cannot be aliases.

Turning to the low-level model of C for an example, we understand that two objects with overlapping regions of storage are dependent on each other. Thus, objects of array, structure and union types depend on the objects that correspond to their members and *vice versa*. For example, given the following declarations

```
struct { int i; } s;
int a[10];
```

`s.i` and `s` depend on each other since these expressions refer to the same region of memory. Also, by definition, `s` and `s.i` are dependent on each other since changing the value of one will affect the value of the other; the dependency relationship can be characterized by the following expression:

```
s.i^ = s.i' ⇔ s^ = s'
```

An object of a structure type contains values that are LSL tuples [GH93, p. 61]. In this case, the value contained in `s` will be a one-component tuple whose value is always equal to the value of `s.i`: that is, `(s•).i = (s.i)•`. Similarly, `a` depends on its members—e.g. `a[9]`. On the other hand, `a[0]`, `a[1]`, ..., `a[9]`, and `s.i` are independent. The dependency relationship that holds between an aggregate or union object and its members is one of the kinds of dependency that programmers rely on and actually take for granted.

When dealing with abstract types we can no longer appeal to the low-level concept of overlapping storage for an intuitive model of dependency. Whether a dependency exists between two instances of an abstract type will depend on the implementation of the abstract type [EHO94].

2.4.1.2 Motivating Example: Error in the Larch Book

The purpose of this example is twofold: we wish to illustrate that there are legitimate uses of dependencies (beyond those mentioned in Section 2.4.1.1) and that there are certain kinds of dependency that are often overlooked by specifiers and implementors.

```
typedef struct {... char name[maxEmployeeName]; ...} employee;

bool employee_setName(employee *e, char na[]) {
   requires nullTerminated(na^);
   modifies e->name;
   ensures result = lenStr(na^) < maxEmployeeName
            ∧ (if result
                then sameStr(e->name', na^)
                     ∧ nullTerminated(e->name')
                else e->name' = e->name^);
}
```

Figure 4: An Excerpt from `employee.lcl`

Our example (see Figure 4) is an excerpt from the Larch book `employee` specification [GH93, p. 65]. This specification is part of a small database program used to store and perform simple queries on employee records. Employee records are represented by the exposed type `employee` which is defined as a C structure. Of the functions provided for manipulating employee records we show only the function `employee_setName`. It can be used to assign a string to the `name` field of an employee record. Before calling `employee_setName`, a client must ensure that the parameter `na` is a null terminated string. After the call, the name field of the given employee record will be set to the string contained in `na` if the string length is less than `maxEmployeeName`. Otherwise, the name field of the record is left unchanged. The function result is true if and only if the length of the string contained in `na` is less than `maxEmployeeName`.

Suppose that all of the employee records in a given database begin with either of the titles “Mr.” or “Ms.” and that the database maintainer wishes to remove the titles. He or she decides to write a program that will accomplish this task by accessing each employee record, say, as the variable `e`, and then performing the call

```
employee_setName(&e,e.name + 3)
```

Unfortunately the program crashes[9] and inspection of the implementation of `employee_setName` reveals the cause:

```
bool employee_setName(employee *e, char na []) {
    int i;

    for (i = 0; na[i] != '0'; i++)
      if (i == maxEmployeeName) return FALSE;
    strcpy(e→name, na);
    return TRUE;
}
```

The particular way in which `employee_setName` is being invoked causes the standard library function `strcpy` to be called with overlapping arguments (since `e->name` and na are part of the same array). The behavior of `strcpy` is undefined when it is called under such circumstances [ISO, §7.11.2.3]. The specification of `employee_setName` does not prohibit calls for which its arguments are dependent. It is possible that the specification inaccurately reflects the intent of its authors or that the source of error is the implementation: in either case the implementation is incorrect with respect to its specification. With appropriate (but small) changes, the implementation can be corrected by making use of the standard library function `memmove` instead of `strcpy` (since `memmove` may be called with overlapping arguments). The reader may wonder whether `memmove` can be specified in LCL; we address this question in Section 2.4.1.4.

We can trace the publication of the database program to the original technical report on LCL 1.0 [GH91]. The program was subsequently revised and published as part of the Larch book [GH93, §5.3]. To determine the effectiveness of LCLint at detecting certain classes of errors in LCL specifications and their implementations, David Evans applied LCLint to (among others) the database program. Evans writes:

> "The specifications [of the database program] had been checked by the LCL checker [a predecessor of the LCLint tool] ..., and the source code had been compiled and tested extensively. Since the code and specifications were written by experts, and checked copiously by hand prior to

[9]A sample program compiled with gcc version 2.6.3 and run under SunOS release 4.1.3 generates a segmentation fault.

publication, it was expected that not many bugs would be found." [Eva94, p. 41]

The case study "did uncover two abstraction violations, and one legitimate modification error" [Eva94, p. 50]. We have demonstrated an additional error in the database program which has also escaped the scrutiny of the original designers and subsequent reviewers.

This example illustrates that there are legitimate uses of dependencies (such as the dependency permitted between `*e` and `na` in `employee_setName`) beyond those mentioned in Section 2.4.1.1. It also illustrates that errors resulting from unexpected dependencies between arguments can easily be overlooked. We believe that this is true because developers have not been encouraged to think about dependencies that may exist among parameters or between parameters and global variables. A specification language that permits dependencies must have constructs that allow the description of dependency relationships as well as a semantic model that supports reasoning about dependencies: LCL is deficient in both these respects.

2.4.1.3 Example: *lookup*

The specification given in Figure 5 defines a global `struct` variable `as` consisting of an array of elements, `elts`, and the size of the prefix of `elts` that is in use. It also defines the function `lookup` which can be used to search for the occurrence of a given value `v` in `as`[10]. If `v` is present in `as`, then `*i` is set to the index of an element of `as` containing `v` and `as` is left unchanged; otherwise, `v` is added to `as` and `*i` is set to the index of the newly added value. The function result is true precisely when the value `v` occurs in `as` (before `lookup` is invoked). The predicate that follows the `else` in the ensures clause of `lookup` is not shown since it is not relevant to our discussion.

After a careful review, the reader may feel that the specification of `lookup` is accurate. It is actually inconsistent—there is no implementation that can satisfy it—since there are situations for which the postcondition cannot be satisfied. For example, suppose that `v` occurs in `as` and that `*i` aliases `as.size` or any of the elements of `as.elts` that are in use. Then the ensures clause states that the value of `*i` may change while requiring that the value of `as` remain unchanged; this constraint, in general, will be unsatisfiable in the presence of the described aliasing.

[10]We will at times use the term "`as`" to refer to "the prefix of `as.elts` that is in use".

```
constant int N;
struct AS {int size; int elts[N];} as;

bool lookup(int v, int *i) struct AS as; {
  requires as.size^ < N;
  modifies *i, as;
  ensures    result = v ∈ prefix(as.elts^,as.size^)
          ∧ if result
             then  0 ≤ (*i)′  ∧  (*i)′ < as.size^
               ∧  as.elts^ [(*i)′] = v
               ∧  as′ = as^
             else /* v is inserted into as.elts */ ...;
}
```

Figure 5: Specification of *lookup*

We can attempt to remedy the situation by strengthening the precondition of lookup so that *i is prohibited from aliasing any of the subcomponents of as (see Figure 6). The resulting specification is less clear and more complex (this augments

```
bool lookup(int v, int *i) struct AS as; {
  requires as.size^ < N   ∧   *i ≠ as.size
          ∧  (∀ j:int ((0 ≤ j  ∧  j ≤ as.size^)
                        ⇒ *i ≠ as.elts[j]));
  ...
}
```

Figure 6: Strengthened Precondition for *lookup*

the risk of introducing errors into the specification) and less maintainable since the specification is now more sensitive to changes in the AS structure.

More importantly, the specification is still inconsistent since it is possible for *i and as to satisfy the requires clause without being independent. In formulating the strengthened precondition we have relied on the following *false* assumption: if two distinct objects are instances of base types (char, int, etc.), then they must be independent. In C, as in some other imperative programming languages, this assumption can be invalidated by the use of union types. Type casting can also invalidate the assumption.

This example illustrates the need for new LCL language primitives which accurately and succinctly express the independence of objects.

2.4.1.4 Example: ISO C String Library Functions

It would be reasonable to expect LCL to be expressive enough to allow one to document the behavior of most ISO C standard library functions. Consider the task of writing specifications for the standard string copying functions memcpy and memmove [ISO, §7.11.2].

```
void *memcpy(void *s1, const void *s2, size_t n);
void *memmove(void *s1, const void *s2, size_t n);
```

Both functions can be used to copy n characters from the object pointed to by s2 into the object pointed to by s1. There is an extra requirement for memcpy: the objects *s1 and *s2 must not overlap [ISO, §7.11.2]. It is impossible to write an LCL specification for memcpy since we cannot express the requirement that its arguments are independent of each other.

2.4.1.5 Dependencies and Abstract Types

The fresh operator is the only LCL operator, other than equality over objects, that allows specifiers to document dependency relationships between objects. An occurrence of the expression fresh(*e*) in the ensures clause of a function specification asserts that the object referred to by *e* is not aliased to any object that was visible to the client before function entry [GH93, p. 77]. By means of the next example, we highlight the need for LCL primitives that would allow for a more precise description of the dependency relationships that may exist between objects.

Most abstract type constructors yield instances of the abstract type that are independent of other client-visible objects. It is not uncommon, though, to find "quick" or "destructive" versions of some constructors that fail to guarantee the independence of the resulting abstract type instance; independence is sacrificed for sake of efficiency.

For example, a list module might provide two versions of the concatenation operation—see Figure 7. Notice that the specification of fastConcat does not ensure fresh(result). It would be more useful, for example, if we could assert that the only dependency

```
mutable type List;

uses List(int,List);

List mkList(void) {
  ensures  result′ = empty ∧ fresh(result);
}
List concat(List x1, List x2) {
  ensures  result′ = x1^ || x2^ ∧ fresh(result);
}
List fastConcat(List x1, List x2) {
  ensures  result′ = x1^ || x2^;
}
```

Figure 7: `List` specification.

created by `fastConcat` is between `result` and `x2`. This extra information would allow us to make better use of `fastConcat`, for example, in the optimization of a series of successive concatenations (as is illustrated in Figure 8).

```
List concatBunch(List x[10]) {
  int i;
  List result;

  result = mkList();
  for(i = 9; 0 ≤ i; i--)
    result = fastConcat(x[i],result);
  return result;
  /* ensures fresh(result)
             ∧ result′ = x[0]^ || ... || x[9]^; */
}
```

Figure 8: Function **concatBunch**

2.4.1.6 Summary

In this section we have defined the concept of dependency between objects. An object x_1 is said to *depend on* an object x_2 if changing the value contained in x_2 may affect the value contained in x_1. Dependencies may arise between:

- an object of an array, structure, or union type and the objects that correspond to its members,
- the members of an object of a union type,
- subarrays of a mutually common array,
- "arbitrary" objects (related by means of a type cast),
- instances of abstract types,
- an instance of an abstract type and the objects that constitute its representation.

Dependences can arise in programs written in any imperative programming language that provides array, structure (record), union or abstract types. It can be said that dependencies are a distinguishing characteristic of imperative programming languages and that they are also to blame for much of the complexity in the semantics of these languages. This would seem to be the price to pay for giving programmers low-level control over storage reuse.

Programmers take certain forms of dependency for granted; such as the dependency between aggregate or union objects and their members. Under certain circumstances, there are other forms of dependency that are undesirable—e.g. the occurrence of aliasing among function parameters. Since it is not possible in general to detect dependencies by static analysis, it becomes the responsibility of the developer to reason about and, when necessary, prove the absence of dependencies. This can be achieved only if an appropriate proof system—supported by a proper collection of language constructs—is available.

By means of examples we have shown that LCL could benefit from the addition of language constructs that would allow specifiers to accurately and succinctly express dependency relationships. Without these constructs there are useful programs that cannot be specified.

2.4.2 Implicit Constraints on Parameters

In LCL, it would seem that the specifications of functions with parameters have implicit constraints, derived from the parameter declarations, that affect the meaning of the specifications. Unfortunately, most of these implicit constraints are either not documented or inadequately defined. The purpose of this section is to expose these implicit parameter constraints and to discuss the consequences of their inclusion in LCL.

2.4.2.1 Constraint for All Parameters

There is an implicit constraint that applies to all parameters in a function specification; it requires that a function be called with arguments that are *defined.* Although the LCL literature is not clear, it would seem that the "defined" means "initialized". For example, given the specification

```
T gv;
void f(T pv) { ... }
```

(where T is any LCL type) a client would be required to initialize gv before calling f with gv as an argument. More concretely, let T be the mutable abstract type empset (of sets of employee records) and f the function empset_clear from the Larch book empset specification [GH93, p. 73]:

```
void empset_clear(empset s) {
   modifies s;
   ensures s' = { };
   }
```

By the absence of a requires clause, no explicit requirements are placed on clients of empset_clear. Implicitly, though, it is assumed that on function entry, s is bound to a defined empset (as can be concluded from the informal description of empset_clear): "empset_clear, is provided for reinitializing an existing empset" [GH93, p. 76].

This implicit constraint on parameters is not documented in the Larch book [GH93] or in Tan's semantics [Tan94] but there is abundant evidence of reliance on the constraint. We have only been able to find an explicit (but incomplete) statement of the implicit parameter constraint in Evans's thesis:

> "An omitted requires clause means there are no constraints on the caller, other than the implied constraint that all parameters that are not specified out must be defined before the call." [Eva94, p. 15]

(The out parameter qualifier is discussed in Section 2.4.2.3.) Although Evans only describes the implicit constraint in the context of an omitted requires clause, the constraint applies even when the requires clause is present. (Otherwise, the specification of a function—of one or more parameters—with an omitted requires clause would differ from the equivalent specification which has **true** as an explicit requires predicate.)

2.4.2.2 Parameters of Pointer Types

There is an additional constraint for parameters of pointer types. The implicit property requires that a pointer parameter reference an allocated object and that this object be defined. This constraint is not documented in the Larch book nor in Tan's semantics[11]. Evans writes:

> "Normally, if a parameter to a function is a pointer, it is assumed that the value it points to is defined and may be used in the body of the function." [Eva94, p. 36]

We discuss some of the shortcomings associated with this implicit constraint.

2.4.2.2.1 Constraint is Overly Restrictive The implicit constraint for pointer parameters is overly restrictive since it prevents us from using certain useful implementation techniques. Consider the specification fragment[12]

```
typedef struct node { ... } *List;
constant List emptyList = 0;
List mkList(int info, List tail) { ... }
```

in which the empty list is represented by a null pointer. The function **mkList** is meant to allow clients to construct a new list from a given integer and list. The implicit constraint for pointer parameters effectively prohibits us from representing the empty list by means of a null pointer, since, for example, we cannot call **mkList** with **emptyList** as an argument for **tail**. This is because all pointer parameters

[11]Tan documents the effect of the out parameter qualifier as applied to parameters of pointer types, but he fails to describe the implicit constraints derived from pointer parameters that are *not* qualified with out.

[12]This specification fragment is not accepted by LCLint 1.4c (the latest release as of the time of writing) because it does not recognize 0 as a null pointer. This will be fixed in a future release.

must refer to allocated objects and a null pointer "is guaranteed to compare unequal to a pointer to any object or function" [ISO, §6.2.2.3]—i.e., a null pointer can never refer to an allocated object.

2.4.2.2.2 Constraint is Ambiguous and Problematic

From a given pointer parameter `p` we can access all of the objects `p+i` for `i` in the index set

$$I = \{\, i \mid \texttt{minIndex(p)} \leq i \leq \texttt{maxIndex(p)} \,\}$$

[GH93, p. 60]. With this in mind, there would seem to be two reasonable interpretations for the implicit constraint. Firstly, we can interpret the implicit constraint as applying to all of the objects that can be accessed via `p`: i.e. all objects `p+i` (for `i` $\in I$) would have to be allocated and defined. Such an interpretation renders the constraint too restrictive. For example, this would require that every member of a string (represented by a pointer into an array of `char`) be initialized before the string can be passed as a parameter, even if the string does not occupy the entire array. There is no reason to require that the string be initialized beyond the null character that terminates the string.

Another possible interpretation for the implicit constraint would require that all objects `p+i` (`i` $\in I$) be allocated but that only the object at `p` need be defined. Assuming $1 \in I$, how would a specifier express the additional requirement that `p+1` be defined? There are no LCL language constructs available to the specifier that would allow the expression of this property.

2.4.2.3 The out Parameter Qualifier

It is common in C for a function to return values to its caller by means of objects that are referenced by the function's pointer parameters; the **out** parameter qualifier serves to indicate which parameters are being used for this purpose [Tan94, §4.3]. The specification of `add` given in Figure 9 illustrates the use of **out**. The **out** qualifier has the effect of partly "relaxing" the extra constraint that is usually applied to pointer parameters. An **out** qualified pointer parameter is still implicitly required to refer to an allocated object, but that object need not be defined [Tan94, §4.3].

As a final remark, we highlight a contradiction in [Tan94]: although Tan states that the **out** qualifier is applicable *only* to parameters of pointer types [Tan94, §4.3],

```
void add(int m, int n, out int *sum) {
  modifies *sum;
  ensures  (*sum)' = m + n;
}
```

Figure 9: Use of out in a function specification.

he also applies it to array parameters [Tan94, §D.27]. Of course, this more liberal use of out is reasonable (and is accepted by LCLint), but it has not been documented. Array parameters are discussed in Section 2.4.2.4.

2.4.2.4 Parameters of Array Types

Although we have found no explicit description of it, there is an implicit constraint on array parameters that is similar to the one for pointer parameters. This would seem reasonable, due to the close relationship between pointers and arrays in C. In fact, someone familiar with C might think that it would be unnecessary to reformulate the implicit constraint for pointer parameters in terms of array parameters because the type of an array parameter is "adjusted to" a pointer type [ISO, §6.7.1]. But in LCL, parameters of array types have a different semantics from those of pointer types [GH93, p. 60], [Tan94, §7.3.1].

The specification of date_parse [Tan94, §D.28] given in Figure 10 provides evidence of the implicit assumption that array parameters refer to objects that have been *allocated* and whose contents are defined. cstring's are null-terminated arrays

```
bool date_parse (cstring indate,..., out date *d)... {
  modifies ...;
  ensures  result = okDateFormat(getString(indate^))
       ∧ if result
         then (*d)' = string2date(getString(indate^))
         ...;
}
```

Figure 10: Tan's date_parse Function

of char. If indate is a well-formatted date, then this date is parsed and returned in `*d`. The function `date_parse` makes use of the content of `indate`, hence `indate` must refer to allocated storage and its contents must be defined.

The implicit constraint over array parameters suffers from the same ambiguities and drawbacks as the constraint for pointer parameters discussed in Section 2.4.2.2.2; i.e., it is not clear whether the implicit constraint requires that all or only some of the array elements be defined—either interpretation leads to difficulties.

2.4.2.5 Parameters of Other Types

Consider a function specification with the header

```
    void f(int **i)
```

The implicit constraints require that `i` be defined and that `*i` be allocated and defined. Suppose that we further wished to constrain the parameter by requiring that `**i` be allocated and defined. We cannot document this extra property for lack of language primitives in LCL. Similar remarks can be made about parameters of other types (e.g. pointer to pointer, array of pointer, struct containing a pointer member).

2.4.2.6 Parameters *vs.* Global Variables

In designing a module one must decide on the mechanisms by which information will be communicated between the module and its clients. In particular, one must choose between information exchange by means of function parameters or global variables. A designer's freedom of choice is impeded (in favor of the use of function parameters) by the lack of expressiveness of LCL.

For example, given

```
    int *gv;

    void f(int *pv) { ... }
    void g(void) int *gv; { ... }
```

one could not express, in the specification of `g`, a constraint on `gv` that would be equivalent to the implicit parameter constraint on `pv` in `f`. This is because there are no language constructs in LCL that express the property that a given object is allocated, or that it is both allocated and defined. It is also because, unlike for function parameters, implicit constraints are not imposed on variables (like **gv**) that are part of the global variable list of a function specification.

2.4.2.7 Summary

We have introduced the various kinds of implicit parameter constraint that affect the meaning of LCL function specifications, and we have argued that the constraints are

- not well documented (they are ambiguous, and in some cases, simply not documented at all),
- without formal or informal semantics, and
- in some cases, overly restrictive (since they prohibit us from writing specifications for useful programs).

We have also illustrated that LCL lacks language constructs that would allow specifiers to assert whether or not an object is an allocated object and whether or not it contains a defined value.

2.4.3 Trashing of Objects

The **trashed** operator can be used in the ensures clause of a function specification to indicate that a given object cannot be reliably accessed after the function returns. The **trashed** operator is typically used in the specifications of functions that deallocate memory or that dispose of instances of mutable abstract types. For example, after a call to the function trashIntObj

```
void trashIntObj(int *i) {
  modifies *i;
  ensures  trashed(*i);
}
```

a client must not attempt to access the contents of `*i` "because referencing a trashed object can even cause the client program to crash" [GH93, p. 76]. Notice the presence of `*i` in the modifies clause: an object can be trashed only if it is listed in the modifies clause—although specifications in the LCL literature consistently mention trashed objects in the modifies clause, there is no explicit statement of this requirement.

Hence, the modifies clause plays a dual role: it serves to identify those objects that may be trashed as well as those objects that may be preserved but whose values may be modified.

On the other hand, after the invocation of `changeVal`

```
void changeVal(int *i) {
  modifies *i;
  ensures  true;
}
```

a client may still make use of `*i` (though no constraint is placed on the value contained in `*i`) [GH93, p. 76]. Thus, an object that is not explicitly trashed is implicitly preserved—i.e. *not* trashed. We will illustrate next that this aspect of the semantics of LCL can lead to contradictory interpretations for function specifications that should logically have the same meaning.

2.4.3.1 Referential Opacity

Consider the following specification of `trashOrChange`, which may nondeterministically choose between trashing and not trashing `*i`:

```
void trashOrChange(int *i) {
  modifies *i;
  ensures  trashed(*i) ∨ ¬trashed(*i);
}
```

The predicate in the ensures clause is an instance of the law of excluded middle and hence, it is logically equivalent to `true`. One would expect to be able to simplify the ensures clause while preserving the meaning of the specification.

```
void trashOrChange(int *i) {
  modifies *i;
  ensures  true;
}
```

The resulting specification of `trashOrChange` cannot trash `*i` because of the implicit constraint that `*i` be preserved.

We have illustrated a violation of the principle of referential transparency which states, in essence, that the only important property of an expression is its value and that we can, consequently, substitute equals for equals. Referential transparency is a fundamental principle of mathematical formalisms. If P is logically equivalent to Q then the meaning of a function specification should remain unchanged if an occurrence of P is replaced by Q in the ensures clause. We have illustrated a counter example above (using `true` and `trashed(*i)` $\vee \neg$ `trashed(*i)`), thus demonstrating an instance of "referential opacity". Not only do formal specification languages permit precise documentation, but they also provide the grounds for the formal analysis and transformation of specifications. Formal arguments are most often conducted within a proof system (rather that by direct application of a model theory). For example, in the Refinement Calculus [Mor90], one can make use of "refinement laws" (which can be used as proof rules) to establish the correctness of an implementation with respect to its specification. As a consequence of the identified referential opacity, we note that laws, such as the strengthen postcondition law, do not hold for LCL. The strengthen postcondition law states that if $Q \Rightarrow R$, then any implementation satisfying (the specification body)

```
requires P;
modifies m1, ..., mk;
ensures  Q;
```

will also satisfy

```
requires P;
modifies m1, ..., mk;
ensures  R;
```

Two applications of the strengthen postcondition law allow us to infer that the two specifications of `trashOrChange` should be equivalent, but they are not. Hence, one of the problems with the implicit constraint related to nontrashed objects is that it invalidates the strengthen postcondition law. Intuitively this law is reasonable (in fact it holds in other specification languages—such as the Refinement Calculus [Mor90] and VDM [Jon90]—in which operations are documented using pre- and postconditions) and invalidity of the law should be taken as an indication of an error in the design of LCL.

2.4.3.2 Trashing the Whole or the Parts of an Object

What can be asserted about an aggregate or union object if one of its members is trashed? For example, given the following declarations

```
int a[10];
struct {int i; char c;} s;
union  {int i; char c;} u;

void trashIntObj(int *i) {
  modifies *i;
  ensures  trashed(*i);
}
```

what can be said about a, s or u after invoking `trashIntObj` with `&a[1]`, `&s.i`, or `&u.i` (respectively) as an argument? (The answer to this question is not as trivial as it first appears. Consider, for example, memory management libraries that allow clients to deallocate (trash) parts of an allocated array.) Conversely, does trashing an instance of an aggregate or union object trash its members too? The answers to these question are not found in the current literature on LCL.

2.4.4 Relevance of Shortcomings

Why are these shortcomings important? The concept of object dependency is apparently lacking from the current Larch culture. There seems to be an implicit assumption that object dependencies exist only in a restricted form. The key problem here is not the assumption itself but that there is no precise statement of this assumption. Hence, it is not possible to determine the kinds of object dependency that are being modeled. This is a serious issue since, fundamentally, it is the semantic model that determines what specifiers can describe and reason about.

Function specifications form the major part of most LCL specifications. Thus, a clear definition of the implicit constraints that are applicable to function specifications is crucial. To this end we have highlighted the nebulosity surrounding the definition of implicit parameter constraints and we have shown the inadequacy of some of these constraints.

The implicit constraint related to nontrashed objects, though clear, creates an instance of referential opacity. This is undesirable since, for example, it invalidates certain laws which will eventually be used to reason about LCL specifications.

Chapter 3

LCL$'$

Two of the points that are considered to be "the essence of Larch" [GH93, §8] are:

- The most important use for specification is as a tool for helping to understand and document interfaces. Therefore, clarity is more important than any other property.

- Specification languages should be carefully designed. Having an elegant semantics is not enough. Careful attention to syntax and static semantic checking is crucial.

Clarity and understandability of specifications can only be achieved if there is clarity in the definition of the specification language. Although an elegant semantics is not sufficient, an accurate and clearly defined semantics is *necessary.* Hence, our primary goal has been to maximize the simplicity and clarity of the semantic model of LCL while maintaining or enhancing the expressiveness of the language.

In this chapter we discuss the major design decisions (Section 3.1) behind LCL$'$, the variant of LCL that is the subject of this thesis. (In the remainder of the thesis an unqualified use of the name "LCL" will refer to LCL$'$ unless noted otherwise.) These design decisions, which have been guided by our primary goal of simplicity, clarity and enhanced expressiveness, affect both the semantic model (and in particular the model of the store given in Chapter 6) and the language definition. In Section 3.2 we propose changes to the LCL language that allow us to overcome the shortcomings documented in Section 2.4. Our approach to the semantic definition of LCL is introduced in Section 3.3 and an overview of the semantics is given in Section 3.4.

3.1 Major Design Decisions

3.1.1 Undefined Variables and Undefined Values

How we choose to model undefined variables, if at all, has a profound impact on the semantic model of LCL. We cannot escape the fact that objects are implemented in some medium—e.g. volatile storage—that is used to encode the values contained in the objects. For a given object of type T, we may ask whether all bit patterns in storage represent values of type T; there are cases where the answer to this question is no. We say that an object is *well-defined* with respect to a type T if it contains an encoding that corresponds to a value of type T; that is, if the object contains a *valid representation* of a value of type T. When we say, without qualification, that an object is well-defined, we mean that the object is well-defined with respect to its declared type.

Uninitialized objects are problematic because there is no guarantee, in the general case, that they are well-defined. The use, in a computation, of an object that is not well-defined can at best have no effect, and at worse lead to program faults whose origin (use of the undefined object) is difficult to locate. Various approaches have been adopted in programming languages to cope with uninitialized variables. Some languages have been designed so as to ensure that every variable always has a well-defined value. Standard ML is one example[1]. In Turing [HMRC88], use of uninitialized variables is illegal and is reported by the compiler, when it can, or detected by run-time checks. The semantics of Turing are given under the assumption that uninitialized variables are not used in computations. Ernst [EHO94] describes a variant of Modula-2 that makes use of implementation specific initialization routines that are invoked automatically when instances of abstract types are created.

The prevailing belief would seem to be that the use, in a computation, of an uninitialized variable is an error. There are exceptions, for example, Hehner's theory of programming supports reasoning about uninitialized variables [Heh93]. In support of the prevailing belief, many compilers (and other static program analysis tools such as LCLint and its ancestor lint) can, in some cases, indicate points in a program which will lead to the use of a variable before it has been assigned a value.

[1]Since SML is principally regarded as a functional programming language its use as an example may seem inappropriate. This is not so, since SML is "equipped ... with full imperative power" [MTH90, p. vii].

Most semantic models (for programming or interface specification languages) do not support uninitialized variables and, if they do, they make a fundamental assumption: once an object has been initialized (i.e. becomes well-defined) it remains well-defined for the remainder of its existence. Of course, this need not be the case.

As is illustrated by the following example, the issue of uninitialized objects can potentially carry over into the underlying logic:

```
immutable type T;
T gv;
void f(T pv) { ... }
```

Since T is an immutable type, the parameter pv is of sort T. If f is invoked with gv as an argument when gv is not well-defined, then pv will have an undefined value. There exist logics that support undefined values, among them are MPL_ω [MdL94] (the multisorted partial infinitary two-valued[2] logic underlying COLD-K [FJKRdL89]) and LPF [JM94] (the three-valued Logic of Partial Functions underlying VDM [Jon90]). There are no theoretical limitations that would prevent LCL from having a logic that would differ from that of LSL. As is discussed further in Section 3.1.2, it is our opinion that, if possible, this should be avoided. In this thesis we explore the feasibility of using the same logic for both LSL and LCL.

3.1.2 Logical Basis of LCL′

LSL is a *specification language*. LSL is used to document traits whose interpretations are viewed as theories in a first-order logic. We will refer to the logic underlying LSL as LL. Although LL has not been formally defined[3], it is understood to be a classical multisorted logic with equality in which all operations are interpreted as *total* functions. LL is very similar to $ML^=$ defined by Middelburg and Renardel de Lavalette [MdL94]. What should be the logic underlying LCL? Although there is no obligation to do so, we have chosen LL. Our principal motivation for choosing LL as the logic underlying LCL is based on our primary design goal: simplicity. Using a logic that would be different from that of LSL would render LCL more complex and

[2]Hence, in MPL_ω, all sorts except the Boolean sort have an undefined value.

[3]Guttag *et al.* provide a semantics for LSL [GHM90], but they do not define LL. Similarly, in the Larch book a tutorial introduction to the logic underlying the Larch languages is given [GH93, §2], but LL is not formally defined.

more confusing to users. Any design decision that will simplify the semantics of LCL should be exploited. If LL proves to be inadequate, then the logic can be changed; maybe in this case we can consider changing the logical foundations of *LSL* to match that of LCL.

3.2 Shortcomings Resolved

Following the organization of Section 2.4, we document solutions to the shortcomings that have been identified in that section.

3.2.1 Dependencies Between Objects

The history of programming languages has been marked by a tendency to make languages more abstract. Increasingly, languages are based on programming concepts (i.e. *semantic objects*) that allow designers to think at a level of abstraction that is closer to the problem domain and further from the computer architectures on which the programs are being executed. In the programming language community, object dependencies tend to be frowned upon. High-level languages tend to severely restrict the kinds of dependency that can be created and low-level languages are characterized by the opposite. In the extreme, object dependencies are prohibited from high-level languages—as in logic or functional programming languages in which computation is based on values rather than objects (by definition, object dependencies cannot exist between values, only between objects). It is important to note that object dependencies *cannot be eliminated* from imperative programming languages that support abstract and indexable[4] types.

By suggesting the systematic adherence to certain programming conventions (e.g. with respect to mechanisms for the implementation and use of abstract types), LCL attempts to raise the level of abstraction at which C programmers think. In providing a semantics for LCL, there would seem to be a tension: although use of LCL promotes C programming at a higher level of abstraction, it is also necessary that the semantic model of LCL subsume that of C since LCL is an interface specification language *for* C. The LCL semantic model must capture the behavior of as large a

[4]E.g. array or dynamic types.

class of C programs as is possible. Hence arises the question: to what degree should dependencies be supported in LCL?

Usually, a model that supports descriptions from two levels of abstraction must be defined in terms of concepts that are from the lowest level. Hence, the semantic model for LCL must accurately capture the kinds of object dependency that can be created in C programs. Our approach to modeling dependencies is formally described in Chapter 6. Of course, it is also necessary that the LCL language have an expressively complete set of constructs for describing dependency relationships. These constructs are introduced next.

In its full generality, the object dependency relation is a dynamic property. For example, dependencies between instances of abstract types implemented by shared realizations may change at run-time [LHO94]. Modeling the object dependency relation as a dynamic property would complicate the semantics and would have important repercussions at the language level. It is not clear, at this point in our research, what language constructs would be best suited to supporting a dynamic dependency relation. The extent to which the dynamic quality of the dependency relation would be actually needed in documenting interface specifications is also unclear. Consequently, in this version of the semantic model the object dependency relation is represented by a static relation, that is, a relation whose value is independent of the program state.

We propose the introduction, in LCL, of two predicates:

- `depOn`(e, e') holds when the object referred to by e depends on[5] the object referred to by e'.
- `indep`$(e_1, e_2, \ldots, e_n)$ holds when the expressions e_1, e_2, …, e_n denote objects that are independent.

The **depOn** predicate allows specifiers to describe any (static) dependency relation that can exist between objects. Although **indep** can be defined in terms of **depOn**, **indep** is more likely to be used in practice since we generally wish to specify that the objects in a given collection are independent (as opposed to characterizing a particular dependency relationship). For example, **indep** can be used to write concise and accurate specifications for the functions `lookup` and `memcpy`. Concretely, in the case of `lookup`, we capture the requirement that `as` and `*i` be independent by adding `indep(as,*i)` to the requires clause:

[5]The definition of dependence is given in Section 2.4.1.1.

```
bool lookup(int v, int *i) struct AS as; {
  requires as.size^ < N  ∧  indep(as,*i);
  ...
}
```

The last example of Section 2.4.1 required that we be able to strengthen the specification of `fastConcat` by ensuring that the only dependencies created by `fastConcat` are between `result` and `x2`. More precisely, we wish to ensure that `result` is independent of any client-visible object that is active in the pre- and post-states and that is also independent of `x2`. One way of rewriting the specification to include this property is as follows[6]

```
List fastConcat(List x1, List x2) {
  ensures ∀ void *x (
           ((*x)\activePre  ∧  (*x)\activePost
           ∧  indep(*x,x2)) ⇒
              indep(result,*x))
         ∧  result′ = x1^  ||  x2^;
}
```

(The \activePre and \activePost operators are discussed in the next section.) The ensures clause is somewhat intimidating. Frequent occurrence, in specifications, of properties like these may warrant the introduction of special notation that would allow us to say, e.g. "`fresh(result)` *except for* `x2`."

3.2.2 Implicit Constraints on Parameters

3.2.2.1 Constraints for All Parameters

Because of our choice of LL as the logic underlying LCL and because LL does not support undefined elements for each sort, we are obliged to keep the implicit constraint that applies to all parameters. Thus, every function parameter is implicitly constrained to have a well-defined value. For example, when applied to the parameter pv of the function `f` in the following specification fragment,

[6]The notation that we are using for the declaration of the quantifier variable is not the notation of LCL 2.4. LCL′ does not currently support quantified expressions; when it does we plan on using C style declarators as is done here.

```
immutable type T;
void f(T pv) { ... }
```

the implicit constraint forbids calling `f` with an lvalue that is not well-defined.

3.2.2.2 New LCL Operators

In Sections 2.4.2.2, 2.4.2.5 and 2.4.2.6, we noted that it is not possible in LCL to express the property that an object is allocated or that it is both allocated and well-defined. For this purpose we propose the introduction of the following boolean operators

```
__ \activePre,  __ \wellDefPre,
__ \activePost, __ \wellDefPost,
__ \activeAny,  __ \wellDefAny : T → Bool
```

The expression *e*`\activePre` holds when the object *e* is active (i.e. allocated) in the pre-state. *e*`\wellDefPre` holds when the object *e* is (active and) well-defined in the pre-state. The other operations provide similar predicates over the post and generic states. Note that the meaning of the trashed operator can be given in terms of \activePost

```
trashed(gv) ⇔ ¬ (gv\activePost)
```

Due to the problems discussed in Section 2.4.2, implicit constraints for pointer and array parameters are not part of LCL′. The new operators can be used to express the necessary constraints. For example, the following specification of `f` requires that the object pointed to by `i` be allocated and that the global variable gv be well-defined.

```
void f(int *i) int gv; {
  requires (*i)\activePre ∧ gv\wellDefPre
  modifies *i;
  ensures  (*i)\wellDefPost ∧ (*i)′ = gv^;
}
```

The function ensures that the post-state value of `*i` is well-defined and that it is equal to the pre-state value of gv.

3.2.3 Trashing of Objects

The semantics of function specifications, in LCL 2.4, is defined in such a way that under certain circumstances some objects are implicitly preserved. We now explain this aspect of the semantics of LCL 2.4 in more detail than in Section 2.4.3 and we reexamine the resulting violation of the principle of referential transparency.

The *modified set* of a function specification consists of those objects that are referenced by expressions occurring in the modifies clause. The *trashed set* of a function specification consists of those objects that are referenced by expressions occurring as arguments to the `trashed` operator in the ensures clause [Tan94, §7.4.1]. For example, the modified and trashed sets for the following specification of `trashSome` are $\{*a, b, *c\}$, and $\{*a, b\}$ respectively.

```
mutable type M;

void trashSome(int *a, M b, int *c) {
  modifies *a,b,*c;
  ensures  (*c)' = (*c)^ + 1 ∧ trashed(b)
           ∧ (if (*a)^ != (*c)^ then
              then ¬trashed(*a) ∧ (*a)' = (*c)^
              else trashed(*a));
}
```

As was indicated in Section 2.4.3, an object that is a member of the modified set may be either trashed or modified. An object in the modified set is implicitly preserved only if it is not a member of the trashed set. In the `trashSome` example, `*c` is implicitly preserved. Thus, the presence or absence of certain argument expressions (of the `trashed` operator) affects the meaning of the function specification. Since the meaning of a function specification depends on more than the truth or falsity of the ensures clause predicate, this clearly leads to a violation of the principle of referential transparency.

To recover referential transparency we need only eliminate that aspect of the semantics that relies on the presence or absence of argument expressions to the `trashed` operator. With this new approach to the semantics, specifiers must explicitly indicate when objects are to be preserved. For example, the specification of `trashSome` would have to be rewritten as

```
void trashSome(int *a, M b, int *c) {
  modifies *a,b,*c;
  ensures  (*c)′ = (*c)^ + 1 ∧ trashed(b)
           ∧ ¬trashed(*c)
           ∧ (if (*a)^ != (*c)^ then
               then ¬trashed(*a) ∧ (*a)′ = (*c)^
               else trashed(*a));
}
```

(Notice the addition to the ensures clause of a predicate asserting that `*c` is not trashed.) In practice, very few functions trash the objects in their modified sets. For example, of the fifty-two functions given in LCL specifications in the Larch book, only two of the thirty-two expressions (that occur in the modifies clauses) are arguments to the `trashed` operator [GH93]. Thus, requiring an explicit statement of the fact that objects are preserved would (unnecessarily) lengthen specifications; function specifications that are less concise are more difficult to write, understand and maintain.

Fortunately there is a better solution. We suggest the introduction of a trashes clause which is syntactically like the modifies clause except for the leading `trashes` keyword. That is, the trashes clause is optional and when present, it may be followed by the `nothing` keyword, or by a list of lvalues (expressions denoting objects). A function may trash an object if and only if that object is referenced by an expression that occurs in the trashes clause[7]. Thus, the modifies clause recovers its intended role: it identifies which objects may have their values *modified*. The roles of the modifies and trashes clauses are *independent*; an expression may occur in both, in either or neither of the clauses. Under this scheme, the specification of `trashSome` would be identical to its original specification but with the addition of the clause `trashes *a,b`.

```
void trashSome(int *a, M b, int *c) {
  modifies *a,b,*c;
  trashes  *a,b;
  ensures  (*c)′ = (*c)^ + 1 ∧ trashed(b)
```

[7]Actually, object dependencies must be taken into account for both the modifies and trashes clauses. Details are given in Section 8.6.

```
                ∧ (if (*a)^ != (*c)^ then
                     then ¬trashed(*a) ∧ (*a)' = (*c)^
                     else trashed(*a));
    }
```

Most function specifications will be written without a trashes clause, implying that no (client-visible) object may be trashed. For those few functions that do trash objects, these objects will be explicitly identified by listing them in the trashes clause.

3.3 A Semantics for LCL′

A language is characterized by the abstract entities that it can be used to describe. These entities are usually called *semantic objects* in contrast to the elements of the (concrete and abstract) syntax of the language that are often referred to as *syntactic objects*[8]. Semantic rules and functions are used to relate syntactic objects to the semantic objects that define their meanings. We present the semantics in a style known as *natural semantics* [Kah87] (having been inspired by the formal definitions of Standard ML [MTH90, MT91] and Extended ML [KST94]). In this style, inference rules are used as the principal means of relating syntactic and semantic objects.

An LCL specification is denoted by a semantic object called an *LCL environment*. An LCL environment has two parts. One part, called an *LCL signature*, describes the specification's *interface* and the other part (which we will call the non-interface part) captures the *meaning* of the interface components. As was noted in Chapter 2, LCL can be regarded as a language consisting of a C interface specification language that makes use of LSL as a sublanguage within which data types can be defined and expressions (over these data types) can be written. Symbolically, LCL = CISL(LSL). Thus, any semantic description of LCL must include a semantics for LSL. Consequently, it would seem that the simplest approach to defining the meaning of the non-interface part of an LCL specification is by means of a "semantic embedding" into LSL. Thus, an LCL specification is denoted (in essence) by an LCL signature and an LSL trait.

There are several advantages to the approach which we have adopted for the formal definition of LCL.

[8]Yet another use of the term "object".

- The semantics is considerably simpler than would be, e.g., a corresponding denotational semantics. Since LCL uses LSL as a sublanguage, a denotational semantics of LCL would have to include a complete semantics for LSL. In our approach we are using LSL as the notation within which the meanings of LCL interface components are expressed. Hence, we can use the features of LSL to enhance the conciseness and clarity of the semantic definition. In a sense, the resulting definition is expressed at a "higher" level of abstraction than could otherwise be achieved.

- Since the meaning of (the non-interface part of) an LCL specification is captured as an LSL trait, the semantic definition should be more accessible to the Larch community in general and to LCL users in particular since they are already familiar with LSL.

- Using a natural semantics (which is a form of translational semantics) allows us to easily "implement" the semantic definition. The result is a tool that generates an LSL trait from an LCL specification. In conjunction with the use of such a tool, we can capitalize on the existence of other Larch tools. For example, scripts for the Larch Prover [GH93, §7] can be obtained for an LCL specification from its corresponding LSL trait by use of the LSL checker [GH93, §7.2].

3.4 Overview of the Semantic Definition

The formal definition of LCL is written primarily in Z [Spi92], a formal specification language based on a typed set theory. Alternatively, we could have invented our own mathematically based notation (as was done for the definition of SML [MTH90]) or, we could have used LSL. We have chosen Z because it is

- a mature notation that is in widespread use,
- an expressive language—also, a high level of conciseness and clarity can be achieved with its use,
- formally defined [Spi88],
- supported by tools (for type-checking and reasoning).

Although in theory, LSL and Z have the same expressive power, the Z formulation of the semantic definition is much clearer than an equivalent formulation written in LSL would be. The Z notation is introduced throughout the thesis as required.

Since LSL is the target formalism into which the non-interface part of LCL specifications are being translated, LSL syntactic objects form a major part of the semantic objects used in the definition of LCL. A partial formalization of LSL is given in Chapter 4; our purpose is not to present a complete definition of LSL, but to provide definitions for a simple and yet expressive subset of LSL which will allow us capture the meaning of any LCL specification. The syntax of LCL is given in Chapter 5. At the heart of the semantic model for LCL is the model of program states, or more precisely, the model of the store, given in Chapter 6. Semantic objects, other than LSL constructs, are given in Chapter 7. Finally, the semantic rules and the semantic functions relating syntactic to semantic objects are given in Chapter 8.

Chapter 4

A Partial Formalization of LSL

In our semantics, the meaning of the non-interface part of an LCL specification is captured by an LSL trait. Thus, constructs of the LSL language are used as semantics objects. In this chapter we present a formalization of LSL. Since our goal is not to provide a semantics for LSL we do not cover all aspects of the language. We formalize only those LSL constructs that are needed to express the meaning of LCL specifications. The material in this section is organized *as if* we were defining a (denotational) semantics for LSL. Hence, Section 4.1 defines an annotated abstract syntax. Semantic objects and semantic functions are given in Sections 4.4 and 4.6, respectively.

Before continuing, we review the major components of an LSL specification by illustrating a simple theory defining groups. A group can be regarded as a monoid with an inverse operation; Figure 11 contains a trait named Group that defines a group in this way. Using the includes statement, traits can be defined by making

```
Group: trait
  includes Monoid
  introduces
    inverse: T → T
  asserts ∀ x:T
    inverse(x) ∘ x = unit;
```

Figure 11: Group Trait

use of other traits. The Monoid trait [GH93, §A.14] that is included in Group defines an associative binary operator '__∘__' over the sort T and a unit element named unit. Following the `introduces` keyword, an operator can be introduced into a trait by providing its name and signature. An operator represents a mathematical function in the theory defined by the trait. The signature of an operator identifies the sorts of its domain and range. Thus, `inverse` is introduced as a unary operation mapping group elements into group elements. Assertions, i.e. predicates that hold in the theory defined by the trait, are given in the assertions part of the trait (following the `asserts` key word): usually a universal quantifier is given, followed by a list of declared variables and a sequence of Boolean terms separated by semicolons.

4.1 Annotated Abstract Syntax

We begin by defining syntactic classes that describe a subset of LSL. The given abstract syntax is termed "annotated" because we assume that the following ambiguities have been resolved:

- simple names have been classified as either logical variables or constant operators.
- overloading has been resolved: that is,
  - each occurrence of a logical variable is qualified with a sort, and
  - each occurrence of an operator is qualified with an operator signature.

4.1.1 Names, Symbols, and Operator Signatures

In an LSL trait a name can refer to a sort, logical variable, operator or trait. These define four name spaces: hence, the same name may be used to refer to any one of these entities. Which entity is being referred to will be determined from the context.

The set of all possible LSL names is represented by Nm. We are not concerned with how names are themselves represented, hence Nm is introduced as a *given set*, also called a *basic type* in Z.

$$[Nm]$$

Sort, logical variable, operator and trait names are (possibly proper) subsets of Nm. $\mathbb{P}\,X$ denotes the power set of X.

$$\mid \; \mathit{SortNm}, \mathit{LVarNm}, \mathit{OpNm}, \mathit{TraitNm} : \mathbb{P}\, \mathit{Nm}$$

`Bool`, `x`, `empty`, `Array` are examples of names. We will at times represent elements of *Nm* literally in the formal Z text by formatting the names in `typewriter` type style. E.g.

$$\{\texttt{Bool}, \texttt{x}, \texttt{empty}\} \subseteq \mathit{Nm}$$
$$\texttt{Bool} \in \mathit{SortNm}$$

In our annotated abstract syntax trees, occurrences of logical variables are qualified with their sorts and operators are qualified with their signatures. The notation *LHS* == *RHS* introduces *LHS* as an abbreviation for the expression *RHS*.

$$\mathit{LVarNS} == \mathit{LVarNm} \times \mathit{SortNm}$$

```
┌─ OpSig ──────────────────────────────
│ opDom : seq SortNm
│ opRan : SortNm
└──────────────────────────────────────
```

$$\mathit{OpNmSig} == \mathit{OpNm} \times \mathit{OpSig}$$

An operator signature identifies the operator's domain (as a sequence of sort names) and its range (as a single sort name). Thus, each operator maps tuples of values into values. We have defined operator signatures by means of the schema type *OpSig*. A schema type, as we have used it here, is analogous to a structure or record type in a programming language. Field selection uses the usual post-fix notation: $E.\mathit{opDom}$ represents the value of the *opDom* component of E.seq X is the set of all finite sequences of X's. Sequences are modeled as partial functions. The domain of a sequence s is the set of integers from 1 up to $\#s$, the length of s: formally, $\mathrm{dom}\, s = \{i : \mathbb{Z} \mid 1 \leq i \leq \#s\}$. A finite sequence of elements can be written by listing the elements enclosed in angle brackets: e.g. $\langle 1, 2, 3 \rangle$.

An LSL operator that takes no arguments is called a *constant operator*. The function *mkConOpSig* yields the signature of a constant operator from the sort name of the operator's range.

$$\begin{array}{|l}
\mathit{mkConOpSig} : \mathit{SortNm} \longrightarrow \mathit{OpSig} \\
\hline
\mathit{mkConOpSig}\ \mathit{sort} = \\
\quad (\mu\ \mathit{OpSig} \mid \mathit{opDom} = \langle\rangle \land \mathit{opRan} = \mathit{sort})
\end{array}$$

In Z, the application of a function f to an argument x is written as $f\,x$. Equivalently, we can write the customary, $f(x)$, which is regarded as the application of f to the parenthesized expression (x). The expression $(\mu\ \mathit{OpSig} \mid P)$ denotes the unique *OpSig* whose components satisfy the predicate P. The set of all sort names that occur in the operator signature *opSig* is given by *opSigSorts opSig*.

$$\begin{array}{|l}
\mathit{opSigSorts} : \mathit{OpSig} \longrightarrow \mathbb{F}\ \mathit{SortNm} \\
\hline
\mathit{opSigSorts} = (\lambda\ \mathit{OpSig} \bullet \{\mathit{opRan}\} \cup \operatorname{ran} \mathit{opDom})
\end{array}$$

The range of *opSigSorts* is the set of all finite subsets of sort names.

The vocabulary of symbols of a trait are the sort names, logical variables names (qualified with their sorts) and operator names (qualified with their signatures) that appear in the trait.

$$\begin{array}{rl}
\mathit{LSym} ::= & \mathit{SortSym}\langle\langle \mathit{SortNm} \rangle\rangle \\
\mid & \mathit{LVarNSSym}\langle\langle \mathit{LVarNS} \rangle\rangle \\
\mid & \mathit{OpNSSym}\langle\langle \mathit{OpNmSig} \rangle\rangle
\end{array}$$

This is our first example of a free type definition. It introduces *LSym* as a basic type with *SortSym*, *LVarNSSym* and *OpNSSym* as constructors. The constructors are modeled as injective functions with disjoint ranges. Each constructor maps elements from the indicated domain (the expression in the $\langle\langle \ldots \rangle\rangle$) into *LSym*. Thus $\mathit{lsym} = \mathit{SortSym}\ \mathtt{S}$ denotes a sort symbol for which the following properties hold:

$$\mathit{lsym} \in \operatorname{ran} \mathit{SortSym}$$
$$\mathtt{S} = \mathit{SortSym}^{\sim}\ \mathit{lsym}$$

For any relation (and hence any function) R, the relational inverse of R is denoted by $R^{\sim}$.

4.1.2 Terms

A term[1] can be either

- an occurrence of a logical variable,
- an application of an operator to some arguments, or
- a quantified term.

$$\begin{aligned} \mathit{Term} ::= {} & \mathit{LVarTm}\langle\!\langle \mathit{LVarNS} \rangle\!\rangle \\ \mid {} & \mathit{AppTm}\langle\!\langle \mathit{OpNmSig} \times \mathrm{seq}\ \mathit{Term} \rangle\!\rangle \\ \mid {} & \mathit{QntTm}\langle\!\langle \mathit{QntTmSchG}[\mathit{Term}] \rangle\!\rangle \end{aligned}$$

LSL has the usual universal and existential quantifiers. *QntNm* is a free type with two constant constructors; i.e., *QntNm* is a type that has only two values named *ForAllNm* and *ExistsNm*.

$$\mathit{QntNm} ::= \mathit{ForAllNm} \mid \mathit{ExistsNm}$$

A quantified term consists of a quantifier, a logical variable (qualified with its sort), and an LSL predicate.

```
┌─ QntTmSchG[X] ──────────────────────
│ qntNm : QntNm
│ qntVar : LVarNS
│ lslPred : X
└─────────────────────────────────────
```

Because of the mutual dependency between *Term* and *QntTmSchG* we are obliged to define *QntTmSchG* as a *generic schema*. For any set S, $\mathit{QntTmSchG}[S]$ represents the schema type that results when the formal generic parameter X is replaced by S in the schema definition of *QntTmSchG*. The value of the actual parameter for the generic schema can sometimes be deduced from the context, hence it need not be written explicitly. An LSL predicate is a Boolean valued term (and hence it is merely represented as a term in the abstract syntax).

$$\begin{aligned} & \mathit{LSLPred} == \mathit{Term} \\ & \mathit{QntTmSch} \mathrel{\widehat{=}} \mathit{QntTmSchG}[\mathit{LSLPred}] \end{aligned}$$

[1]An unqualified use of the word "term" will always, unless indicated otherwise, refer to an *LSL term*.

We define *QntTmSch* to be the schema type *QntTmSchG*[*LSLPred*].

The name of a quantifier variable is not relevant to the meaning of the quantified term of which it is a part. We say that two terms are (*syntactically*) *equivalent* if they are identical or, if one term can be obtained from the other by the renaming of its bound variables. A term can be either a logical variable, the application of an operator to arguments or a quantified term: if two terms are to be equivalent, then they must be of the same kind of term. We define *equivTm* as a binary relation over *Term*'s; i.e. a subset of the Cartesian product *Term* × *Term*.

$$
\begin{array}{|l}
equivTm : Term \leftrightarrow Term \\
\hline
(tm_1, tm_2) \in equivTm \Rightarrow \\
\quad \{tm_1, tm_2\} \subseteq \operatorname{ran} LVarTm \lor \\
\quad \{tm_1, tm_2\} \subseteq \operatorname{ran} AppTm \lor \\
\quad \{tm_1, tm_2\} \subseteq \operatorname{ran} QntTm
\end{array}
$$

Two terms that are logical variables are equivalent if they have the same name and sort.

$$
\begin{array}{|l}
(LVarTm\ lvarNS_1, LVarTm\ lvarNS_2) \in equivTm \Leftrightarrow \\
\quad lvarNS_1 = lvarNS_2
\end{array}
$$

Two terms, that are the application of operators to arguments, are equivalent if the operators are the same, they have the same number of arguments and the corresponding arguments are equivalent terms.

$$
\begin{array}{|l}
\text{let } tm_1 == AppTm(opNmSig_1, tms_1); \\
\quad tm_2 == AppTm(opNmSig_2, tms_2) \bullet \\
(tm_1, tm_2) \in equivTm \Leftrightarrow \\
\quad opNmSig_1 = opNmSig_2 \land \\
\quad \#tms_1 = \#tms_2 \land \\
\quad (\forall i : \operatorname{dom} tms_1 \bullet \\
\qquad (tms_1\ i, tms_2\ i) \in equivTm)
\end{array}
$$

Two quantified terms are equivalent if renaming the quantifier variable of the second term to the quantifier variable of the first term results in equivalent subterms. (Renamings are defined in Section 4.2.)

$$
\begin{array}{|l}
\textbf{let } tm_1 == QntTm\,\theta QntTmSch_1; \\
\quad tm_2 == QntTm\,\theta QntTmSch_2 \bullet \\
(tm_1, tm_2) \in equivTm \Leftrightarrow \\
\quad qntNm_1 = qntNm_2 \land \\
\quad second\ qntVar_1 = second\ qntVar_2 \land \\
\quad (\textbf{let } ren == (\mu\, Ren \mid \\
\qquad sortPRen = \varnothing \land \\
\qquad lvarPRen = \{qntVar_2 \mapsto first\ qntVar_1\} \land \\
\qquad opPRen = \varnothing) \bullet \\
\quad (lslPred_1, renTm\ ren\ lslPred_2) \in equivTm)
\end{array}
$$

We will sometimes use the $x \mapsto y$ abbreviation for (x, y), as we have done above. The functions *first* and *second* can be used to extract the first, respectively second, component of an ordered pair: $first(x, y) = x$ and $second(x, y) = y$.

In Z, we can achieve the effect of a record constructor for schemas by using the θ notation. For example, the expression $\theta OpSig$ denotes the value of *OpSig* (called a *schema binding*) formed from the values of the variables *opDom* and *opRan* from the surrounding scope. For example, in the following expression,

$$
\begin{array}{l}
\textbf{let } opDom == \langle \mathsf{S}_1, \mathsf{S}_2 \rangle; \\
\quad opRan == \mathsf{S} \bullet E
\end{array}
$$

an occurrence of $\theta OpSig$ within the expression E will denote an *OpSig* whose *opDom* is $\langle \mathsf{S}_1, \mathsf{S}_2 \rangle$ and whose *opRan* is S.

By *decorating* a schema name we obtain a new schema type. Any combination of primes ($'$), ?, ! and numeric subscripts can be used for decoration. For example, *OpSig'* represents the schema

$$
\begin{array}{|l}
\hline
opDom' : \mathrm{seq}\ SortNm \\
opRan' : SortNm \\
\hline
\end{array}
$$

which has the same components as *OpSig* except that the decoration ($'$) is applied to all of the component names to obtain new, decorated, component names.

Since terms are fully qualified with their sorts, it is always possible to determine the sort of a term. The sort of a term *tm* is given by *tmSort tm*.

$$
\begin{array}{|l}
tmSort : Term \longrightarrow SortNm \\
\hline
tmSort(LVarTm(lvarNm, sort)) = sort \\
tmSort(AppTm(opNm \mapsto \theta OpSig, tms)) = opRan \\
tmSort(QntTm\ \theta QntTmSch) = \texttt{Bool}
\end{array}
$$

The function *tmSyms* yields the set of all symbols that occur in a term. This function is used to define *tmSorts*, *tmLVarNSs*, *tmOpNSs* that yield the set of all sort, logical variable and operator names (respectively) that occur in term.

$$
\begin{array}{|l}
tmSyms : Term \longrightarrow \mathbb{F}\ LSym \\
tmSorts : Term \longrightarrow \mathbb{F}\ SortNm \\
tmLVarNSs : Term \longrightarrow \mathbb{F}\ LVarNS \\
tmOpNSs : Term \longrightarrow \mathbb{F}\ OpNmSig \\
\hline
tmSyms(LVarTm(lvarNm, sort)) = \\
\qquad \{LVarNSSym(lvarNm, sort), SortSym\ sort\} \\
tmSyms(AppTm(opNm \mapsto opSig, tms)) = \\
\qquad \{OpNSSym(opNm, opSig)\} \cup \\
\qquad SortSym (\!| opSigSorts\ opSig |\!) \cup \\
\qquad \bigcup (tmSyms (\!| \mathrm{ran}\ tms |\!)) \\
tmSyms(QntTm\ \theta QntTmSch) = \\
\qquad \{LVarNSSym\ qntVar, SortSym(second\ qntVar)\} \cup \\
\qquad \{SortSym\ \texttt{Bool}\} \cup \\
\qquad tmSyms\ lslPred \\
\\
tmSorts\ tm = SortSym^{\sim} (\!| tmSyms\ tm |\!) \\
tmLVarNSs\ tm = LVarNSSym^{\sim} (\!| tmSyms\ tm |\!) \\
tmOpNSs\ tm = OpNSSym^{\sim} (\!| tmSyms\ tm |\!)
\end{array}
$$

If $R \in X \leftrightarrow Y$ and $xs \in \mathbb{P}\, X$, then $R(\!|xs|\!)$ (called the relational image of xs under R) is the set of all Y's that are related by R to the X's in xs. $\bigcup xss$ denotes the distributed union of the sets in xss.

The function *tmFreeLVars* yields the set of variables that occur *free* in a given term.

$$\begin{array}{|l}
tmFreeLVars : Term \longrightarrow \mathbb{F}\ LVarNS \\
\hline
tmFreeLVars(LVarTm\ lvarNS) = \{lvarNS\} \\
tmFreeLVars(AppTm(opNmSig, tms)) = \\
\qquad \bigcup(tmFreeLVars(\!|\mathrm{ran}\ tms|\!)) \\
tmFreeLVars(QntTm\ \theta QntTmSch) = \\
\qquad (tmFreeLVars\ lslPred) \setminus \{qntVar\}
\end{array}$$

The set difference operator is written as '\' in Z.

4.1.3 Traits

A trait has a name and a body. The body of a trait is simply defined as a sequence of trait components.

$$TraitBody == \mathrm{seq}\ TraitCpt$$

$$\begin{array}{|l}
\textit{Trait} \\
\hline
traitNm : TraitNm \\
tb : TraitBody \\
\hline
\end{array}$$

A trait component is either

- a variable declaration (a variable name and a sort)
- an operator declaration (an operator name and signature)
- an assertion, or
- an include statement (including a single trait)

$$\begin{array}{rl}
TraitCpt ::= & LVarCpt\langle\!\langle LVarNS\rangle\!\rangle \\
\mid & OpCpt\langle\!\langle OpNmSig\rangle\!\rangle \\
\mid & AsnCpt\langle\!\langle LSLPred\rangle\!\rangle \\
\mid & InclCpt\langle\!\langle TraitRef\rangle\!\rangle
\end{array}$$

We identify the trait to be included by means of a trait reference containing the name of the trait and a renaming. Renamings are presented in Section 4.2.

$$\begin{array}{|l}
\hline \mathit{TraitRef} \\
\mathit{traitNm} : \mathit{TraitNm} \\
\mathit{ren} : \mathit{Ren} \\
\hline
\end{array}$$

We sometimes wish to include a trait without performing any renaming. This is achieved by a trait reference with an empty renaming.

$$\begin{array}{|l}
\mathit{traitNm2Ref} : \mathit{TraitNm} \longrightarrow \mathit{TraitRef} \\
\hline
\mathit{traitNm2Ref}\ \mathit{traitNm}' = \\
\quad (\mu\ \mathit{TraitRef} \mid \mathit{traitNm} = \mathit{traitNm}' \wedge \mathit{ren} = \mathit{emptyRen})
\end{array}$$

The function *mkTB* yields the set of all trait bodies that contain the trait references, logical variables, operators and assertions that are provided as arguments to the function.

$$\begin{array}{|l}
\mathit{mkTB} : \mathbb{F}\ \mathit{TraitRef} \times \mathbb{F}\ \mathit{LVarNS} \times \\
\qquad \mathbb{F}\ \mathit{OpNmSig} \times \mathbb{F}\ \mathit{LSLPred} \longrightarrow \mathbb{P}\ \mathit{TraitBody} \\
\hline
\mathit{tb} \in \mathit{mkTB}(\mathit{refs}, \mathit{lvarNSs}, \mathit{opNmSigs}, \mathit{asns}) \Leftrightarrow \\
\quad \mathit{refs} = \mathit{InclCpt}^{\sim}(\!|\mathrm{ran}\ \mathit{tb}|\!) \wedge \\
\quad \mathit{lvarNSs} = \mathit{LVarCpt}^{\sim}(\!|\mathrm{ran}\ \mathit{tb}|\!) \wedge \\
\quad \mathit{opNmSigs} = \mathit{OpCpt}^{\sim}(\!|\mathrm{ran}\ \mathit{tb}|\!) \wedge \\
\quad \mathit{asns} = \mathit{AsnCpt}^{\sim}(\!|\mathrm{ran}\ \mathit{tb}|\!)
\end{array}$$

4.1.4 Extension of LSL Syntax and a Special Notation

Given an arbitrary context, most LSL terms written in concrete syntax are ambiguous. E.g. the term

```
0 + p
```

is potentially ambiguous since the operators 0 and + could be overloaded. Furthermore, we do not know whether p refers to a constant operator or a logical variable (either of which could be overloaded). To allow us to write terms unambiguously, we extend the concrete syntax of LSL so that constant operators can be followed by parentheses, (), to distinguish them from similarly named logical variables. Using this extension, along with explicit qualification of terms with their sorts, we obtain a scheme that allows us to unambiguously denote terms. E.g.

```
(0:int + p():int):int
```

When denoting an LSL term as a value of the Z type *Term*, it will be convenient to use LSL concrete syntax along with the following special notation. In an LSL term (written in concrete syntax), fragments that are underlined represent Z expressions which denote values of type *Term* or *Nm* which are meant to replace (in the concrete syntax tree) the underlined Z expressions. For example, given

$$\begin{aligned}
&sort = \mathtt{Z}\\
&tm = (\textbf{let } xtm == LVarTm(\mathtt{x} \mapsto \mathtt{X});\\
&\qquad ptm == AppTm(\mathtt{p} \mapsto mkConOpSig\,\mathtt{Y}, \langle\rangle);\\
&\qquad opSig == (\mu\ OpSig \mid\\
&\qquad\qquad\qquad opDom = \langle \mathtt{X}, \mathtt{Y}\rangle \land\\
&\qquad\qquad\qquad opRan = \mathtt{Z}) \bullet\\
&\qquad AppTm(\mathtt{f} \mapsto opSig, \langle xtm, ptm\rangle))
\end{aligned}$$

then

$\mathtt{z}\!:\!\underline{sort} < \underline{tm}$

represents

```
z:Z < f(x:X,p():Y):Z
```

In a Z expression, an LSL term (written in concrete syntax) within metabrackets '$[\![\ldots]\!]$' is meant to represent the corresponding Z expression of type *Term*. For example, the following Z predicate is true

$[\![\mathtt{f(x:X,p():Y):Z}]\!] = tm$

4.2 Renamings

The symbols in the vocabulary of a trait may be renamed. Renamings are sometimes used, for example, when including a trait into another trait. We represent a renaming by three partial renamings partitioned according to the symbols that they rename: there is a partial renaming for sort, operator and logical variable names[2].

PRen

$sortPRen : SortNm ⇻ SortNm$
$lvarPRen : LVarNS ⇻ LVarNm$
$opPRen : OpNmSig ⇻ OpNm$

---

disjoint $\langle sortPRen, \mathrm{id}\, SortNm \rangle$
disjoint $\langle lvarPRen, \mathrm{id}\, LVarNm \circ first \rangle$
disjoint $\langle opPRen, \mathrm{id}\, OpNm \circ first \rangle$

Since operators can be overloaded, the renaming for operators requires that an operator and its signature be specified as the target of a renaming. Similar remarks apply to the renaming for logical variables. For reasons explained below, we require identity renamings be excluded from the functions *sortPRen*, *lvarPRen* and *opPRen*.

We define functions that allow us to *apply* renamings to terms, trait components, and trait bodies. Note that the components of *PRen* are partial functions. The definitions of renaming application can be simplified if we have total functions with identical domain and ranges. For this purpose we define *Ren*.

[2] $X ⇻ Y$ denotes the set of partial functions from X's into Y's that have finite domains. id X is the Z notation for the identity function over X's. disjoint $\langle xs_1, \ldots, xs_n \rangle$ is true precisely when the sets xs_i in the given sequence are disjoint. '$\circ$' is the (backward) relational composition operator—$(f \circ g)\, x = f(g\, x)$.

Ren

```
$PRen$
$sortRen : SortNm \longrightarrow SortNm$
$lvarNSRen : LVarNS \longrightarrow LVarNS$
$opNSRen : OpNmSig \longrightarrow OpNmSig$
――――――――
$sortRen\ sort =$
    if $sort \in \mathrm{dom}\ sortPRen$
    then $sortPRen\ sort$
    else $sort$

$lvarNSRen(nm, sort) =$
    (let $nm' ==$
        if $(nm, sort) \in \mathrm{dom}\ lvarPRen$
        then $lvarPRen(nm, sort)$
        else $nm$ •
    $(nm', sortRen\ sort))$

$opNSRen(nm, opSig) =$
    (let $nm' ==$
        if $(nm, opSig) \in \mathrm{dom}\ opPRen$
        then $opPRen(nm, opSig)$
        else $nm$ •
    $(nm', opSigRen\ sortRen\ opSig))$
```

where *opSigRen* renames the sorts in *opSig* according to the sort renaming *sortRen*

```
$opSigRen : (SortNm \longrightarrow SortNm) \longrightarrow OpSig \longrightarrow OpSig$
――――――――
$opSigRen\ sortRen =$
    $(\lambda\ OpSig' \bullet$
        $(\mu\ OpSig \mid$
            $opDom = sortRen \circ opDom' \wedge$
            $opRan = sortRen\ opRan'))$
```

It is also possible to define the total functions in *Ren* more succinctly as[3]

[3]The *pair* function is defined in Appendix C. '$\oplus$' denotes the functional overriding operator.

$$\begin{array}{|l}
\textit{Ren\_AltDef} \\
\hline
\textit{Ren} \\
\hline
\textit{sortRen} = \mathrm{id}\,\textit{SortNm} \oplus \textit{sortPRen} \\
\textit{lvarNSRen} = \\
\qquad \textit{pair}\,(\textit{first} \oplus \textit{lvarPRen})\,(\textit{sortRen} \circ \textit{second}) \\
\textit{opNSRen} = \\
\qquad \textit{pair}\,(\textit{first} \oplus \textit{opPRen})\,(\textit{opSigRen}\ \textit{sortRen} \circ \textit{second}) \\
\hline
\end{array}$$

The constraint imposed on *PRen*'s allows us to formulate the following property: each *Ren* is uniquely determined by its *PRen* and each *PRen* corresponds to a unique *Ren*[4]:

$$(\lambda\, Ren \bullet \theta PRen) \in Ren \rightarrowtail\!\!\!\!\twoheadrightarrow PRen$$

Therefore, when defining a renaming it is sufficient to provide the values of the partial functions in *PRen*. For example, when ren_0, given by

$$\begin{array}{l}
ren_0 = (\mu\, Ren \mid \\
\qquad sortPRen = \{\mathtt{Y} \mapsto \mathtt{YY}\} \land \\
\qquad lvarPRen = \{(\mathtt{z}, \mathtt{Z}) \mapsto \mathtt{zz}\} \land \\
\qquad opPRen = \{(\mathtt{f}, (\mathtt{X}, \mathtt{Y} \longrightarrow \mathtt{Z})) \mapsto \mathtt{ff}\})
\end{array}$$

is applied:

- All occurrences of the sort name Y will be changed to YY. All other sort names will be left unchanged.
- All occurrences of the logical variable z declared to be of sort Z will be renamed to zz. All other logical variable names will be left unchanged.
- Finally, all occurrences of the operator `f` with signature X, Y ⟶ Z will be renamed to `ff`. All other operator names will be left unchanged.

4.2.1 Applying a Renaming to a Term

The application of the renaming θRen to the term *tm* is given by $renTm\,\theta Ren\ tm$.

[4] $X \rightarrowtail\!\!\!\!\twoheadrightarrow Y$ is the set of bijections from X onto Y.

$$renTm : Ren \longrightarrow Term \longrightarrow Term$$

We define *renTm* by cases on *tm*.

- If *tm* is an occurrence of the logical variable *lvarNS*, then the logical variable is renamed.

$$\begin{array}{l} renTm\ \theta Ren\ (LVarTm\ lvarNS) = \\ \quad LVarTm(lvarNSRen\ lvarNS) \end{array}$$

- If *tm* is the application of the operator *opNmSig* to the arguments *tms*, then the operator is renamed and θRen is recursively applied to each of the arguments.

$$\begin{array}{l} renTm\ \theta Ren\ (AppTm(opNmSig, tms)) = \\ \quad (\mathbf{let}\ tms' == renTm\ \theta Ren \circ tms \bullet \\ \qquad AppTm(opNSRen\ opNmSig, tms')) \end{array}$$

- Care must be taken when a renaming is applied to a quantified term to ensure that bound variables are not renamed. The renaming $\theta Ren'$ is the same as θRen with the exception that application of $\theta Ren'$ leaves the quantifier variable *qntVar* unchanged. The result of applying θRen to *tm* is the quantified term that results from the application of $\theta Ren'$ to *lslPred*[5]. In the definition we use μ-expressions in their generalized form. For example, the value of $(\mu\ Ren' \mid P \bullet E)$ is the value that E takes on when the schema components of Ren' are bound to the unique values that satisfy the predicate P.

[5] $xs \lhd\!\!\!- R$ is the relation obtain by omitting from R the pairs $(x, y) \in R$ for which $x \in xs$. '$\lhd\!\!\!-$' is called the domain anti-restriction operator. Similarly, $xs \lhd R$ is the relation obtained from R by keeping those pairs $(x, y) \in R$ for which $x \in xs$. '$\lhd$' is called the domain restriction operator.

$$
\begin{array}{|l}
renTm\,\theta Ren\,(QntTm\,\theta QntTmSch) = \\
\quad (\mu\ Ren' \mid \\
\qquad sortPRen' = sortPRen \land \\
\qquad lvarPRen' = \{qntVar\} \lhd lvarPRen \land \\
\qquad opPRen' = opPRen \bullet \\
\quad (\mu\ QntTmSch' \mid \\
\qquad qntNm' = qntNm \land \\
\qquad qntVar' = qntVar \land \\
\qquad lslPred' = renTm\,\theta Ren'\ lslPred \bullet \\
\quad QntTm\,\theta QntTmSch'))
\end{array}
$$

As a concrete example, the application of ren_0 (defined above) to the term

```
z:Z < f(x:X,p():Y):Z
```

will result in the term

```
zz:Z < ff(x:X,p():YY):Z
```

4.2.2 Applying a Renaming to a Trait Component

We define the application of the renaming θRen to the trait component *traitCpt* by cases. To do so we need a composition function for renamings

$$
\begin{array}{|l}
compRen : Ren \times Ren \longrightarrow Ren \\
\hline
compRen(\theta Ren', \theta Ren'') = \\
\quad (\mu\ Ren \mid \\
\qquad sortRen = sortRen'' \circ sortRen' \land \\
\qquad lvarNSRen = lvarNSRen'' \circ lvarNSRen' \land \\
\qquad opNSRen = opNSRen'' \circ opNSRen')
\end{array}
$$

The definition of *compRen* is such that the application of the composition (of two renamings) will be the same as the application of each renaming in succession (this property is formalized in Section 4.2.4).

$$
\begin{array}{|l}
renTraitCpt : Ren \longrightarrow TraitCpt \longrightarrow TraitCpt
\end{array}
$$

- If *traitCpt* is a variable or operator declaration, then the appropriate partial renaming is applied.

$$
\begin{array}{|l}
renTraitCpt\ \theta Ren\ (LVarCpt\ lvarNS) = \\
\qquad LVarCpt(lvarNSRen\ lvarNS) \\
\\
renTraitCpt\ \theta Ren\ (OpCpt\ opNmSig) = \\
\qquad OpCpt(opNSRen\ opNmSig)
\end{array}
$$

- The renaming of an assertion component is achieved by the renaming of its constituent assertion (i.e. term).

$$
\begin{array}{|l}
renTraitCpt\ \theta Ren\ (AsnCpt\ tm) = \\
\qquad AsnCpt(renTm\ \theta Ren\ tm)
\end{array}
$$

- If *traitCpt* is an include component, then *θRen* and the renaming in the trait reference are composed. The resulting renaming is used to form a new trait reference.

$$
\begin{array}{|l}
renTraitCpt\ \theta Ren\ (InclCpt\ \theta TraitRef') = \\
\qquad (\mu\ TraitRef\ | \\
\qquad\qquad traitNm = traitNm' \land \\
\qquad\qquad ren = compRen(ren', \theta Ren) \bullet \\
\qquad InclCpt\ \theta TraitRef)
\end{array}
$$

4.2.3 Applying a Renaming to a Trait Body

The application of a renaming to a trait body is obtained by applying the renaming to each of the components in the trait body.

$$
\begin{array}{|l}
renTB : Ren \longrightarrow TraitBody \longrightarrow TraitBody \\
\hline
renTB\ ren\ tb = renTraitCpt\ ren \circ tb
\end{array}
$$

4.2.4 Properties

The application of the empty renaming

$$\begin{array}{|l}
\mathit{emptyRen} : \mathit{Ren} \\
\hline
\mathit{emptyRen} = (\mu\, \mathit{Ren} \mid \\
\qquad \mathit{sortPRen} = \varnothing \land \\
\qquad \mathit{lvarPRen} = \varnothing \land \\
\qquad \mathit{opPRen} = \varnothing)
\end{array}$$

leaves things unchanged, that is

$$\begin{array}{l}
\mathit{renTm}\ \mathit{emptyRen} = \mathrm{id}\ \mathit{Term} \\
\mathit{renTraitCpt}\ \mathit{emptyRen} = \mathrm{id}\ \mathit{TraitCpt} \\
\mathit{renTB}\ \mathit{emptyRen} = \mathrm{id}\ \mathit{TraitBody}
\end{array}$$

Applying two renamings in succession is the same as applying their composition.

$$\begin{array}{l}
\mathit{renTm}\ \mathit{ren}'\ (\mathit{renTm}\ \mathit{ren}\ \mathit{tm}) = \\
\qquad \mathit{renTm}\ (\mathit{compRen}(\mathit{ren}, \mathit{ren}'))\ \mathit{tm}
\end{array}$$

$$\begin{array}{l}
\mathit{renTraitCpt}\ \mathit{ren}'\ (\mathit{renTraitCpt}\ \mathit{ren}\ \mathit{traitCpt}) = \\
\qquad \mathit{renTraitCpt}\ (\mathit{compRen}(\mathit{ren}, \mathit{ren}'))\ \mathit{traitCpt}
\end{array}$$

$$\begin{array}{l}
\mathit{renTB}\ \mathit{ren}'\ (\mathit{renTB}\ \mathit{ren}\ \mathit{tb}) = \\
\qquad \mathit{renTB}\ (\mathit{compRen}(\mathit{ren}, \mathit{ren}'))\ \mathit{tb}
\end{array}$$

Thus, renamings form a monoid under composition (*compRen*) with the identity *emptyRen*.

4.3 Substitutions

A substitution can be applied to a given term to replace all free occurrences of one or more logical variables by arbitrary terms. We represent a substitution by a partial function from logical variable names (qualified with their sorts) into terms.

$$\mathit{Subst} == \mathit{LVarNS} \nrightarrow \mathit{Term}$$

A naive implementation of substitution application, which we will call unsafe substitution, is defined next.

```
| unsafeAppSubst : Subst ⟶ Term ⟶ Term
|------------
| unsafeAppSubst subst (LVarTm lvarNS) =
|     if lvarNS ∈ dom subst
|     then subst lvarNS
|     else LVarTm lvarNS
|
| unsafeAppSubst subst (AppTm(opNmSig, tms)) =
|     (let tms' == unsafeAppSubst subst ∘ tms •
|         AppTm(opNmSig, tms'))
|
| unsafeAppSubst subst (QntTm θQntTmSch') =
|     (let subst' == {qntVar'} ⩤ subst •
|     QntTm (μ QntTmSch |
|         qntNm = qntNm' ∧
|         qntVar = qntVar' ∧
|         lslPred = unsafeAppSubst subst' lslPred'))
```

This definition of substitution application is termed unsafe because the free variables that occur in the terms (that are replacing the logical variables) may become bound. For example,

```
let subst == { (x, Bool) ↦ ⟦b:Bool⟧ };
    tm == ⟦∀b : Bool x:Bool⟧ •
    unsafeAppSubst subst tm = ⟦∀b : Bool b:Bool⟧
```

Since the name of a quantifier variable is not relevant to the meaning of a term, variable capture can be avoided with an appropriate renaming of bound variables. Hence, we define *safe substitution* [MW85, §3.3]. The result of safely applying the substitution *subst* to the term *tm* is any one of the terms which is obtained as follows: if necessary, to avoid variable capture, we rename the quantifier variables that appear in *tm*, then we can "safely" make use of *unsafeAppSubst*.

$$
\begin{array}{|l}
appSubst : Subst \longrightarrow Term \longrightarrow \mathbb{P}\ Term \\
\hline
tm' \in appSubst\ subst\ tm \Leftrightarrow (\exists\, tm_0 : Term \bullet \\
\quad (tm, tm_0) \in equivTm \land \\
\quad (subst, tm_0) \notin capture \land \\
\quad tm' = unsafeAppSubst\ subst\ tm_0)
\end{array}
$$

The details regarding the circumstances under which a variable is captured can be found in, e.g. [MW85, §3.3].

4.4 Semantic Objects

4.4.1 Signatures

The sort, logical variable and operator names that are used in a trait define the trait's *vocabulary of names*. An *LSL signature* (or *signature* when it is clear from the context that we are referring to an LSL signature), can be used to represent the vocabulary of a trait. Since an LSL operator name may be associated with more than one operator signature and a logical variable name may be associated with more than one sort, a signature must actually identify a trait's *vocabulary of symbols.* In Section 4.1.1 we defined a symbol to be either a sort name, a logical variable name qualified with its sort, or an operator name qualified with its signature.

$$
\begin{array}{|l}
\textit{LSig} \\
\hline
sorts : \mathbb{F}\ SortNm \\
lvarNSs : \mathbb{F}\ LVarNS \\
opNSs : \mathbb{F}\ OpNmSig \\
lvars : \mathbb{F}\ LVarNm \\
ops : \mathbb{F}\ OpNm \\
\hline
ops = first(\!|opNSs|\!) \\
\bigcup(opSigSorts(\!|second(\!|opNSs|\!)|\!)) \subseteq sorts \\
lvars = \mathrm{dom}\ lvarNSs \\
second(\!|lvarNSs|\!) \subseteq sorts \\
\hline
\end{array}
$$

sorts, *lvarNSs*, and *opNSs* represents the vocabulary of sort, logical variable and operator symbols respectively[6]. At times we need to refer to names rather than symbols. For this purpose, we define *lvars* and *ops* to represent the set of logical variable and operators names that are part of the vocabulary of names.

For a signature to be consistent, every sort name that occurs in a logical variable or operator symbol must be present in the sort name vocabulary. The LSL Checker [GH93, §7.2] disallows a constant operator and a logical variable from having the same name and sort (within any given trait). We find this restriction unnecessary and hence we do not support it. When such a situation arises, an occurrence of the name is taken to be an instance of the variable (since variables have tighter scopes). Since relations are represented as sets of ordered pairs in Z, we note that

$$\begin{array}{l} opNSs \in OpNm \leftrightarrow OpSig \\ lvarNSs \in LVarNm \leftrightarrow SortNm \end{array}$$

The empty signature contains no names:

$$\begin{array}{|l} emptyLSig : LSig \\ \hline emptyLSig = (\mu\ LSig \mid \\ \quad sorts = \varnothing \land \\ \quad opNSs = \varnothing \land \\ \quad lvarNSs = \varnothing) \end{array}$$

Two signatures can be combined. The result is a signature that contains the names that were present in both of the original signatures.

$$\begin{array}{|l} addLSig : LSig \times LSig \rightarrow LSig \\ \hline addLSig = (\lambda\ LSig';\ LSig'' \bullet \\ \quad (\mu\ LSig \mid sorts = sorts' \cup sorts'' \land \\ \qquad opNSs = opNSs' \cup opNSs'' \land \\ \qquad lvarNSs = lvarNSs' \cup lvarNSs'')) \end{array}$$

The expression *lsigHideOps lsig opNmSigs* is the signature obtained by hiding the operators in *opNmSigs* from the vocabulary of *lsig*.

[6]For convenience, *sorts*, *lvarNSs*, and *opNSs* are declared to be of types $\mathbb{F}\ SortNm$, $\mathbb{F}\ LVarNS$ and $\mathbb{F}\ OpNmSig$ respectively, rather than $\mathbb{F}\ LSym$.

$$
\begin{array}{|l}
lsigHideOps : LSig \longrightarrow \mathbb{F}\ OpNmSig \longrightarrow LSig \\
\hline
lsigHideOps = (\lambda\, LSig' \bullet \\
\quad (\lambda\, opsToHide : \mathbb{F}\ opNSs' \bullet \\
\qquad (\mu\, LSig \mid \\
\qquad\quad sorts = sorts' \land \\
\qquad\quad lvarNSs = lvarNSs' \land \\
\qquad\quad opNSs = opNSs' \setminus opsToHide)))
\end{array}
$$

4.4.2 Special Trait Bodies

The non-interface part of an LCL specification is translated into an LSL trait body. During the translation process we need to have convenient access to some information from the trait body that is being built. Hence, we define *TBS* to hold a trait body along with components derived from the trait body.

$$
\begin{array}{|l}
\textit{TBS} \\
\hline
tb : TraitBody \\
lsig, \\
localLSig : LSig \\
includes : \mathrm{seq}\ TraitRef \\
asns : \mathrm{seq}\ LSLPred \\
\hline
lsig = tb2LSig\ tb \\
localLSig = tb2LSig(tb \upharpoonright (\mathrm{ran}\ LVarCpt \cup \mathrm{ran}\ OpCpt)) \\
includes = InclCpt^{\sim} \circ (tb \upharpoonright \mathrm{ran}\ InclCpt) \\
asns = AsnCpt^{\sim} \circ (tb \upharpoonright \mathrm{ran}\ AsnCpt) \\
\hline
\end{array}
$$

The derived components include[7]:

- a "local" LSL signature (*localLSig*) containing only the vocabulary of symbols that are explicitly declared in *tb*,
- a "global" LSL signature (*lsig*) containing the vocabulary of all symbols declared explicitly or implicitly (via included traits) in *tb*,

[7]The function *tb2LSig* is defined in Section 4.6.2.

- the sequence of trait references that occur in *tb*, and
- the sequence of LSL predicates that occur as assertions in *tb*.

A *TBS* is uniquely defined by its *tb* component.

$$(\lambda\ TBS \bullet tb) \in TBS \rightarrowtail\!\!\!\!\twoheadrightarrow TraitBody$$

4.5 Trait Store and Include Dependencies

We assume the existence of a "store" that maps trait names to trait bodies.

$$\left|\ traitStore : TraitNm \longrightarrow TraitBody\right.$$

Using this store we can define a mapping from trait references into trait bodies. Given the reference $\theta\, TraitRef$, this is achieved by extracting the body of the named trait *traitNm*, and applying the renaming *ren* to this body.

$$\begin{array}{|l}
ref2TB : TraitRef \longrightarrow TraitBody \\
\hline
ref2TB = (\lambda\ TraitRef \bullet renTB\ ren\ (traitStore\ traitNm))
\end{array}$$

A trait body *tb* is said to include the trait named *traitNm'* when *tb* includes a reference to *traitNm'*. Similarly, a trait named *traitNm* is said to include a trait named *traitNm'* if (the body of) *traitNm* includes a reference to *traitNm'*.

$$\begin{array}{|l}
tbIncl : TraitBody \leftrightarrow TraitNm \\
includes : TraitNm \leftrightarrow TraitNm \\
\hline
(tb, traitNm') \in tbIncl \Leftrightarrow \\
\quad (\exists\ traitRef : TraitRef \mid \\
\quad\quad traitRef.traitNm = traitNm' \bullet \\
\quad\quad InclCpt\ traitRef \in \operatorname{ran} tb) \\
(traitNm, traitNm') \in includes \Leftrightarrow \\
\quad (traitStore\ traitNm, traitNm') \in tbIncl
\end{array}$$

A trait body *tb* or a trait named *traitNm* is *free of include cycles* if none of the traits that it (transitively) includes (transitively) include themselves.

$$
\begin{array}{|l}
noInclCycles : \mathbb{P}\ TraitNm \\
noInclCycInTB : \mathbb{P}\ TraitBody \\
\hline
traitNm \in noInclCycles \Leftrightarrow \\
\quad (\forall\ traitNm' : includes^{*}(\!|\{traitNm\}|\!) \bullet \\
\qquad (traitNm', traitNm') \notin includes^{+}) \\
tb \in noInclCycInTB \Leftrightarrow \\
\quad (\forall\ traitNm : tbIncl(\!|\{tb\}|\!) \bullet traitNm \in noInclCycles)
\end{array}
$$

4.6 Semantic Functions

4.6.1 Well-structuredness and Well-formedness

A term is *well-structured* if within the term each occurrence of

- an operator is applied to an appropriate number of arguments and that each argument is well-structured and has a sort that matches the corresponding sort in the operator signature,
- a term that is part of a (universally or existentially quantified) binding has the sort Bool and is well-structured.

$$
\begin{array}{|l}
wsTm : \mathbb{P}\ Term \\
\hline
LVarTm\ lvarNS \in wsTm \\
AppTm(opNm \mapsto \theta OpSig, tms) \in wsTm \Leftrightarrow \\
\quad tmSort \circ tms = opDom\ \land \\
\quad \operatorname{ran} tms \subseteq wsTm \\
QntTm\ \theta QntTmSch \in wsTm \Leftrightarrow \\
\quad tmSort\ lslPred = \mathtt{Bool}\ \land \\
\quad lslPred \in wsTm
\end{array}
$$

A term *tm* is *well-formed* with respect to a given signature *lsig* if *tm* is well-structured and if all free occurrences of symbols are present in *lsig*.

$$\begin{array}{|l}
wfTm : LSig \longrightarrow \mathbb{P}\ Term \\
\hline
LVarTm\ lvarNS \in wfTm\ \theta LSig \Leftrightarrow lvarNS \in lvarNSs \\
\\
AppTm(opNm \mapsto \theta OpSig, tms) \in wfTm\ \theta LSig \Leftrightarrow \\
\quad opNm \mapsto \theta OpSig \in opNSs \land \\
\quad tmSort \circ tms = opDom \land \\
\quad \mathrm{ran}\ tms \subseteq wfTm\ \theta LSig \\
\\
QntTm\ \theta QntTmSch \in wfTm\ \theta LSig \Leftrightarrow \\
\quad tmSort\ lslPred = \mathtt{Bool} \land \\
\quad (\exists\ lsig' : LSig \mid \\
\quad\quad lsig' = tb2LSig\ \langle LVarCpt\ qntVar \rangle \bullet \\
\quad\quad\quad lslPred \in wfTm(addLSig(\theta LSig, lsig')))
\end{array}$$

4.6.2 From Traits to Signatures

Provided there are no include cycles in the trait body tb, then the $LSig$ that corresponds to tb is simply the sum of the $LSig$'s obtained from each of the components in the trait body.

$$\begin{array}{|l}
tb2LSig : TraitBody \nrightarrow LSig \\
\hline
\mathrm{dom}\ tb2LSig = noInclCycInTB \\
\\
tb2LSig\ \langle\rangle = emptyLSig \\
\langle traitCpt \rangle \frown tb \in \mathrm{dom}\ tb2LSig \Rightarrow \\
\quad tb2LSig(\langle traitCpt \rangle \frown tb) = \\
\quad\quad addLSig(trtCpt2LSig\ traitCpt, tb2LSig\ tb)
\end{array}$$

$tb2LSig$ can also be defined using $foldLL$

$$\begin{array}{l}
\forall\ tb : TraitBody \bullet \\
tb \in \mathrm{dom}\ tb2LSig \Rightarrow \\
\quad (\mathbf{let}\ lsigs == trtCpt2LSig \circ tb \bullet \\
\quad\quad tb2LSig\ tb = foldLL\ addLSig\ emptyLSig\ lsigs)
\end{array}$$

The function $trtCpt2LSig$ is defined by cases over $TraitCpt$.

$$
\begin{array}{|l}
trtCpt2LSig : TraitCpt \rightarrowtail\!\!\!\!\!\!\!\!+\;\; LSig \\
\hline
\mathrm{dom}\ trtCpt2LSig = \{traitCpt : TraitCpt \mid \\
\quad traitCpt \in \mathrm{ran}\ InclCpt \Rightarrow \\
\quad\quad (InclCpt^{\sim}\ traitCpt).traitNm \in noInclCycles\} \\
\\
trtCpt2LSig(LVarCpt(lvarNm, sort)) = \\
\quad (\mu\ LSig \mid \\
\quad\quad sorts = \{sort\} \land \\
\quad\quad lvarNSs = \{lvarNm \mapsto sort\} \land \\
\quad\quad opNSs = \varnothing) \\
\\
trtCpt2LSig(OpCpt(opNm, opSig)) = \\
\quad (\mu\ LSig \mid \\
\quad\quad sorts = opSigSorts\ opSig \land \\
\quad\quad lvarNSs = \varnothing \land \\
\quad\quad opNSs = \{opNm \mapsto opSig\}) \\
\\
trtCpt2LSig(AsnCpt\ tm) = emptyLSig \\
\\
traitRef.traitNm \in noInclCycles \Rightarrow \\
\quad trtCpt2LSig(InclCpt\ traitRef) = \\
\quad\quad tb2LSig(ref2TB\ traitRef)
\end{array}
$$

Chapter 5

Annotated Abstract Syntax

We define an annotated abstract syntax for LCL in which identifiers and operators are annotated with their types, sorts, or signatures (as appropriate) so as to remove any possibility of ambiguity. For sake of completeness, we provide the concrete syntax of LCL in Appendix B.

5.1 Identifiers

LCL identifiers are modeled as a subset of the set of all names that can appear in an LSL trait.

$$Id : \mathbb{P}\, Nm$$

Nm is defined in Section 4.1.1.

5.2 Types

In C there are qualified and unqualified types. A member of the syntactic class of phrases that constitute the qualified types is called a *type name.* (In contrast, a name given to a type by means of a `typedef` is called a *typedef name.*) A type name is formed from an *unqualified type name* and a (possibly empty) set of type qualifiers.

```
── TpNmG[XUTpNm] ──────────────────
tpQuals : 𝔽 TpQual
utn : XUTpNm
───────────────────────────────────
```

$$TpNm \triangleq TpNmG[UTpNm]$$

We will follow common practice and use the term "type name" to refer to a qualified type name.

There are two C type qualifiers: `const` and `volatile`. When used in a variable declaration, `const` informs the compiler that the value of the given variable will be constant; `volatile` variables may change value even when they are not assigned to—e.g. these variables could refer to memory mapped input/output ports. In LCL we also have the `out` parameter qualifier (discussed in Section 2.4.2.3) and the `obj` qualifier.

$$\begin{aligned} TpQual ::= &\ ConQual \\ \mid &\ VolQual \\ \mid &\ ObjQual \\ \mid &\ OutQual \end{aligned}$$

If the sort of an LCL type T is `S`, then the sort of the `obj`-qualified type `obj` T is `Obj[S]`.

5.2.1 Unqualified Types

Next, we define the unqualified types of C as well as collective names given to certain groups of C types. The following information is taken from the ISO C standard [ISO, §6.1.2.5].

- `void` corresponds to the "empty" type; there are no values of type `void`.
- `char` type is for characters.
- `signed char`, `short int`, `int`, and `long int` define the *signed integer types.*
- `unsigned char`, `unsigned short int`, `unsigned int`, and `unsigned long int` define the *unsigned integer types.*
- `char`, `signed char`, and `unsigned char` are the *character types.*
- `float`, `double`, and `long double` are the *floating types.*

- char, the signed and unsigned integer types, and the floating types are collectively named the *basic types.*

  $$BasicTp ::= IntTp \mid \ldots$$

- Each enumeration corresponds to a distinct *enumerated type.* (Enumerated types are not supported in the current semantic definition.)

- void, the basic types and the enumerated types are called the *fundamental types.*

The remaining unqualified types are *derived types*: array, structure, union, function and pointer types. An array type consists of a qualified *element type* and, optionally, an expression defining the array dimension. E.g.

```
const int aci[];
struct S *aps[10];
```

declares aci to be an array of const qualified integers with an unspecified number of elements—thus, the element type of aci is 'const int'. aps is an array of 10 pointers to S tagged structures—the element type of aps is 'struct S*' and its dimension is 10. Function types are discussed in Section 5.2.2. Structure, union and pointer types are not supported in this version of the semantic definition. Array and structure types are called the *aggregate types*—union types are not considered aggregate types since it is assumed that only one member of a union is active at any given time.

$$\begin{array}{lcl} OptExp & ::= & NoExp \\ & \mid & OneExp\langle\langle Exp\rangle\rangle \\[6pt] UTpNm & ::= & VoidUTN \\ & \mid & BasicUTN\langle\langle BasicTp\rangle\rangle \\ & \mid & ArrUTN\langle\langle TpNmG[UTpNm] \times OptExp\rangle\rangle \\ & \mid & FunUTN\langle\langle FunUTNSch[UTpNm]\rangle\rangle \\ & \mid & ImmUTN\langle\langle Id\rangle\rangle \\ & \vdots & \end{array}$$

Figure 12: Unqualified Type Names (*UTpNm*)

In addition to the types of C, LCL provides immutable and mutable abstract types. An abstract type is identified by its name. (Only immutable abstract types are supported in the current version of the semantic definition.) A Z free type definition for unqualified type names is given in Figure 12.

5.2.2 Function Types

A function type identifies the function return type ($retUTN$)—which must be an unqualified type [ISO, §6.5.3]—as well as the number of function parameters and the type of each parameter [ISO, §6.1.2.5].

$FunUTNSch0[XUTpNm]$

$retUTN : XUTpNm$

$prmTNs : \mathrm{seq}\ TpNmG[XUTpNm]$

In LCL, a function must be declared by means of a prototype (hence, old-style function declarations [ISO, §6.5.4.3] are not supported). We also impose the following additional restrictions:

- parameter names are mandatory (contrary to C, where they are optional),
- support is not (yet) provided for functions that accept a variable number of arguments.

It is more convenient if we generalize function types to include the names of function parameters and the global variable lists (that are a part of the headers of function specifications).

$FunUTNSch[XUTpNm]$

$FunUTNSch0[XUTpNm]$

$prmIds,$

$gvarIds : \mathrm{seq}\ Id$

$gvarUTNs : \mathrm{seq}\ XUTpNm$

$\#prmIds = \#prmTNs$

$\#gvarIds = \#gvarUTNs$

With this scheme, a function specification header will simply consist of an identifier declared with this generalized function type.

For example, the function type of `f`, which is a part of the following function specification header

```
void f(const int a[5], int *i) int gv;
```

is represented as

$$
\begin{array}{l}
(\mu\, \mathit{FunUTNSch} \mid \\
\quad \mathit{retUTN} = \texttt{void} \land \\
\quad \mathit{prmTNs} = \langle \texttt{const int}[5], \texttt{int*} \rangle \land \\
\quad \mathit{prmIds} = \langle \texttt{a}, \texttt{i} \rangle \land \\
\quad \mathit{gvarIds} = \langle \texttt{gv} \rangle \land \\
\quad \mathit{gvarUTNs} = \langle \texttt{int} \rangle)
\end{array}
$$

5.3 Expressions

Any LSL term is a valid LCL expression (within the appropriate context). Hence, we model LCL expressions in the same way as LSL terms. An LCL expression is either

- an occurrence of a logical variable (*LVarExp*),
- the application of an (LSL, LCL or C) operator to a sequence of arguments (*AppExp*),
- or a quantified expression[1].

$$
\begin{array}{lll}
\mathit{Exp} ::= & \mathit{LVarExp}\langle\!\langle \mathit{LVarNS} \rangle\!\rangle \\
& \mid \ \mathit{AppExp}\langle\!\langle \mathit{OpNmSig} \times \mathrm{seq}\, \mathit{Exp} \rangle\!\rangle \\
& \vdots
\end{array}
$$

An LCL predicate is simply a Boolean expression.

$$\mathit{Pred} == \mathit{Exp}$$

[1]Quantified expressions are not supported in the current version of the semantic definition.

5.4 Declarations

A declaration is composed of an identifier and a type. A declaration can be used to introduce a constant, variable, function, type or function parameter. Since there are qualified and unqualified type names, we also have qualified ($QDcl$) and unqualified ($UDcl$) declarations.

$$\begin{array}{|l}
\hline
UDclG[XUTpNm] \\
id : Id \\
utn : XUTpNm \\
\hline
\end{array}$$

$$\begin{array}{|l}
\hline
QDclG[XUTpNm] \\
id : Id \\
tn : TpNmG[XUTpNm] \\
\hline
\end{array}$$

$$UDcl \mathrel{\widehat{=}} UDclG[UTpNm]$$
$$QDcl \mathrel{\widehat{=}} QDclG[UTpNm]$$

5.5 Specifications and Specification Components

An LCL specification consists of a sequence of specification components

$$Spec == \mathrm{seq}\ Cpt$$

A specification component is either

- an import clause,
- a uses clause,
- a constant, type or variable declaration,
- a function specification or,
- a claim.

In this version of the semantics we restrict our attention to constant and variable declarations and function specifications since they are at the heart of every LCL specification.

Declarations are the only components to which the spec qualifier can be applied. Thus, a declaration component (*DclCpt*) is a spec or a non-spec declaration.

$$
\begin{array}{lll}
\mathit{DclQual} & ::= & \mathit{SpecQual} \mid \mathit{NotSpecQ} \\[1ex]
\mathit{Cpt} & ::= & \mathit{DclCpt}\langle\!\langle \mathit{DclQual} \times \mathit{Dcl} \rangle\!\rangle \\
 & \mid & \mathit{FSCpt}\langle\!\langle \mathit{FSHeader} \times \mathit{FSBody} \rangle\!\rangle \\
 & \vdots &
\end{array}
$$

Constant and variable declarations are given by means of unqualified and qualified declarations respectively. Type declarations (declarations of abstract types or typedef's) are not included in this version of the semantics.

$$
\begin{array}{lll}
\mathit{Dcl} & ::= & \mathit{ConDcl}\langle\!\langle \mathit{UDcl} \rangle\!\rangle \\
 & \mid & \mathit{VarDcl}\langle\!\langle \mathit{QDcl} \rangle\!\rangle \\
 & \vdots &
\end{array}
$$

A function specification (*FSCpt*) consists of a header and a body. The header consists of a C function prototype and a list of global variables. We represent the header as an unqualified declaration[2].

$$\mathit{FSHeader} \mathrel{\widehat{=}} \mathit{UDcl}$$

$$
\begin{array}{|l}
\hline
\textit{FSBody} \\
\mathit{reqExp} : \mathit{Pred} \\
\mathit{modExps}, \\
\mathit{trashExps} : \mathrm{seq}\ \mathit{Exp} \\
\mathit{ensExp} : \mathit{Pred} \\
\hline
\end{array}
$$

The behavior of the function is specified by means of a body (*FSBody*) which contains the following clauses:

[2]Why and how this is possible is discussed in Section 5.2.2.

- *requires* and *ensures* clauses which define requires and ensures predicates, respectively.
- *modifies* and *trashes* clauses which contain a sequence of expressions (or the keyword `nothing` which can be used to denote the empty sequence of expressions).

Chapter 6

A Storage Model

Use of LCL encourages developers to organize their programs as a collection of modules where each module may be documented by means of an LCL specification. Some of the components exported by modules are executable and when executed, they exhibit a certain behavior. Our model of behavior is based on states. At any given moment a program is assumed to be in a particular state. The state contains, among other things, the values contained in the objects that are in use by the program as well as program control information[1]. The behavior of a component is represented by the sequence of state transformations induced by the execution of the component. The behavior of noninteractive (nonconcurrent) components is recorded as a sequence of two states: the state before the component is executed, called the *pre-state*, and the state that results when the component returns (provided execution terminates), called the *post-state*.

For simple languages it is sufficient to model the state as a mapping from (variable) identifiers into denotable values.

$[Id, Val]$
$State0 == Id \longrightarrow Val$

This model is too elementary for most practical languages. When defining a semantic model for a programming language it is customary (and Tennent says convenient) to partition the computational state into two components: an environment and a store [Ten81, p. 16]. The environment consists of a mapping from identifiers into objects

[1]Although, usually, program control information is not explicitly represented.

and the store binds objects to the values they contain [Ten81, p. 61].

$$Env == Id \longrightarrow Obj$$
$$Store == Obj \longrightarrow Val$$
$$State1 == Env \times Store$$

Semantic functions tend to alter either the environment or the store and usually not both. For example, a variable declaration will alter the environment and an assignment of a value to an object will alter the store. Hence, this partitioning of the state usually results in a simpler semantic definition.

The semantics of the non-interface part of an LCL specification is given in the form of an LSL trait. There is no need to explicitly model the environment component of the program state since the environment is implicitly described in the LSL translation of the LCL specification. In this chapter, we present two formalizations of our model of the store. Our main vehicle for expressing the semantics of LCL is Z. Hence the first formalization, given in Section 6.2, is written in Z. LSL is the target notation into which the non-interface part of LCL specifications are encoded. To avoid having to use metalogical properties or to mix different notations, we encode the Z formalization of the store in LSL (Section 6.3). Section 6.4 contains the traits defining the meaning of the sorts on which LCL exposed types are based. We note that our model of the store is quite general and it could be used as a part of the semantic model of other imperative programming languages or module interface specification languages.

6.1 Requirements for the Model of the Store

The model of the store is the corner stone of the LCL semantic model. The storage model must accurately capture the nature of the store as it is (meant to be) used in LCL (and C). The model must also be expressive enough to allow the meaning of all LCL constructs to be described. Hence, both the C and LCL languages impose requirements on the model of the store. In the subsections that follow we highlight the most important of these requirements. All of the stated requirements are satisfied by our model of the store.

6.1.1 Objects of Aggregate and Union Types

In C, an object of an aggregate or union type is viewed as a collection of objects. The members of aggregate types are independent but those of union types need not be—dependencies are discussed in the next section. For example, given the following declarations

```
int a[i];
struct {int i; char c;} s;
union  {int i; char c;} u;

void change(int *i) {
  modifies *i;
  ensures (*i)' != (*i)^;
}
```

the expressions `a[i]`, `s.i` and `u.c` denote objects: we can thus, for example, use `&a[i]`, `&s.i` and `&u.i` as arguments to **change**. The model must support such a representation for aggregate and union types. This rules out, in particular, models in which an object of an aggregate type is represented as an "entire" (i.e. atomic) object[2].

6.1.2 Dependencies and `fresh`

Dependency relationships may exist, for example, between an aggregate or union object and its members and among the members of a union object. Dependencies may also exist between instances of abstract types. The model must support such object dependencies. For reasons given in Section 3.2.1, the current version of the semantic model only supports static object dependencies.

An occurrence of the expression `fresh(`*e*`)` in an ensures clause states that the object denoted by *e* is independent of any client visible object that is active in both the pre- and post-states. Hence, it is necessary to be able to identify the set of active objects for a given state and to assert that a given object is independent of each of the objects in this set.

[2]In LM3, for example, arrays and records are modeled as "entire" objects [Jon92].

6.1.3 Modifies Clause

Function specifications form the major part of most LCL specifications and the modifies clause has much to contribute to the meaning of each function specification. Informally, the meaning of the modifies clause is as follows[3]:

> The set of all objects that are explicitly or implicitly referred to by the list of expressions in the modifies clause is called a *frame*. Any client-visible object that is active in the pre-state, that is not in the frame, and that is still active in the post-state must have the same abstract value in both states, or be undefined in both states.

The formalization of this statement is expressed as a quantified predicate over the set of all objects; hence, it must be possible to represent this set in our model of the store. Furthermore, in formalizing this statement it is necessary to be able to determine, for any given state

- which objects are active,
- which objects are well-defined, hence
  - the sort attribute[4] of any active object, and, finally,
  - the value contained in any well-defined object.

The statement that an object has the same abstract value in two given states can be ascertained only if the sort attribute (i.e. type) of the object is known. The object dependency relation is used to determine the objects that are implicitly asserted as being a part of the frame.

6.2 Theory of the Store

As is commonly done in the formalization of typed or multisorted languages, we begin by defining an unsorted model of the store (Section 6.2.1). Independently from the unsorted model we formalize the object dependency relation (Section 6.2.2) and sorts (Section 6.2.3). These three theories (defining the unsorted store, object dependencies

[3]A formal treatment of the meaning of the modifies clause is given in Section 8.6.

[4]The sort of the values contained in an object is called the object's *sort attribute*. See Section 6.2.3.

and sorts) are used in the definition of our sorted model of the store (Section 6.2.4). Finally, a theory scheme for defining a sorted projection of the sorted store is given in Section 6.2.5.

Why define a model of the store in terms of LSL sorts and not LCL types? An LCL type can be thought of as a set of values with an associated collection of operators [GH93, p. 4, p. 58]. In LCL, as in other Larch interface languages, each type is *based on* (or *associated with*) an LSL sort [GH93, p. 21, p. 58]: i.e., the "semantics of the type"—by which we mean the semantics of the operators defined over the type are provided by the LSL traits that define the sort on which the type is based. Since the semantics of each LCL type is defined in terms of the sort associated with the type, it becomes reasonable to base the storage model on sorts instead of types.

6.2.1 An Unsorted Model

We begin with two brief explanations of the purpose of the store. From a traditional computational perspective we can say that a store is used to store (i.e. record and preserve) values. Since we usually wish to store more than one value, we must have a convenient way of identifying the values that are stored so that we can later retrieve them: objects can be regarded as "labeled"[5] containers that serve this purpose.

From an object-oriented perspective, a store can be seen as a representation of a collection of objects. Execution of an object-oriented program results in the birth, mutation and death of objects. At any given moment there is never more than a finite number of live or *active* objects. Each active object has a collection of attributes. The identity of an object is a fundamental attribute: each object has a unique identity that is invariable over time and that is set at the birth of the object (or possibly even before). Other attributes include the value contained in an object and the sort of this value.

In either perspective, we see that the notion of object is central. *Obj* represents the set of all possible objects.

$[Obj]$

We require that there be an infinite supply of objects (our motives for doing so are

[5]The object identity serving as a label.

given in Section 6.2.3).

$$\forall\, objs : \mathbb{F}\, Obj \bullet$$
$$\quad \exists\, obj : Obj \bullet obj \notin objs$$

The values that can be stored in an object are from the given set U.

$$[U]$$

Our unsorted model of the store (captured by the *UStore* schema) identifies the objects that are active as well as the values contained in the active objects.

```
── UStore ──────────────────────────────
│ activeObjs : F Obj
│ val : Obj ⇸⇸ U
├────────────
│ activeObjs = dom val
└────────────────────────────────────────
```

As was mentioned earlier, there can only be a finite number of active objects—hence, *activeObjs* is a finite set. *val* is a finite partial function whose domain coincides with the set of active objects. Note that *UStore* is isomorphic to $Obj \twoheadrightarrow\!\!\!\!\!\!\!\!\;\shortmid\;\; U$.

6.2.2 Dependencies Between Objects

We say that the object x *depends on* the object x' if changing the value contained in x' may cause a change in the value contained in x. If x does not depend on x', then x is *independent of* x'. Object dependencies are modeled by means of the *depOn* relation.

```
│ depOn : Obj ↔ Obj
├────────────
│ depOn ∈ Reflexive
```

Every object depends on itself; hence *depOn* is reflexive. *depOn* need not be symmetric; asymmetric dependencies can arise with the use of abstract data types. The object dependency relation need not be transitive: e.g. given the declaration

```
int a[3];
```

a[1] depends on a and a depends on a[2], but a[1] and a[2] are independent.

The objects in a given indexed collection xs are *independent* (of each other, or *mutually independent*) if all objects from the collection are pair-wise independent (for pairs constructed from objects at different indices).

$$\begin{array}{|l}
indep : \mathbb{P}(\mathrm{seq}\ Obj) \\
\hline
xs \in indep \Leftrightarrow \\
\qquad (\forall i, j : \mathrm{dom}\, xs \mid i \neq j \bullet (xs\ i, xs\ j) \notin depOn)
\end{array}$$

$indep\ xs$ is true precisely when the objects in the sequence xs are independent. $indep$ is fully defined in terms of $depOn$.

For any object, x, the set of objects on which it depends is given by $depOn(\!|\{x\}|\!)$. The set of objects that depend on x is $depOn^{\sim}(\!|\{x\}|\!)$.

6.2.3 Sorts

Objects contain values from the unsorted domain U. Unsorted values are meaningless unless we know how to interpret them—just as a string of bits is meaningless unless we know what it is meant to represent. Thus, each object has a *sort attribute* which informs us of the intended interpretation of the unsorted values which it contains. Sorts are represented by their names.

$[SortNm]$

$$\begin{array}{|l}
sortAttr : Obj \twoheadrightarrow SortNm
\end{array}$$

Each object has a fixed sort attribute that is independent of the state. There is at least one object of each sort—hence $sortAttr$ has been declared as a surjective function.

We assume that there is an infinite supply of objects of each sort. Hence Obj must be infinite—this property is stated in Section 6.2.1. This assumption will simplify our model since we will not have to deal with the complications of computational failures due to insufficient storage.

$$\begin{array}{|l}
objsSort : SortNm \longrightarrow \mathbb{P}\ Obj \\
\hline
objsSort = relToFun\ sortAttr^{\sim} \\
objsSort\ sort \notin \mathbb{F}\ Obj
\end{array}$$

objsSort sort is the set of all objects containing values of sort *sort*. (The definition of *relToFun* is given in Appendix C.)

For each sort *sort* we assume the existence of an equality relation (over U) that holds when two unsorted values are considered equal when viewed as sorted values of sort *sort*. Not all unsorted values in U will represent a sorted value of sort *sort*. The set of unsorted values that are valid representations for values of sort *sort* is called the *representation set* of *sort*. The representation set coincides with the domain of the equality relation.

$$\begin{array}{|l}
\mathit{equal} : \mathit{SortNm} \longrightarrow (U \leftrightarrow U) \\
\mathit{repSet} : \mathit{SortNm} \longrightarrow \mathbb{P}\ U \\
\hline
\mathit{repSet}\ \mathit{sort} = \mathrm{dom}(\mathit{equal}\ \mathit{sort}) \\
\mathit{equal}\ \mathit{sort} \in \mathit{EquivalenceRel}[\mathit{repSet}\ \mathit{sort}]
\end{array}$$

Although *equal sort* is not an equivalence relation over U—since it will not be reflexive if $\mathit{repSet}\ \mathit{sort} \neq U$—it is an equivalence relation over its domain.

There are different kinds of equality. *equal sort* is an *existential* equality relation. Later, while defining the semantics, we will find it useful to have a *strong* equality relation. Strong equality over a sort *sort* differs from existential equality (over the same sort) in that strong equality also holds true for a pair of values (u, u') if neither u nor u' is a valid representation of a value of sort *sort*.

$$\begin{array}{|l}
\mathit{strongEq} : \mathit{SortNm} \longrightarrow (U \leftrightarrow U) \\
\hline
(u, u') \in \mathit{strongEq}\ \mathit{sort} \Leftrightarrow \\
\qquad (u, u') \in \mathit{equal}\ \mathit{sort} \lor \\
\qquad u \notin \mathit{repSet}\ \mathit{sort} \land u' \notin \mathit{repSet}\ \mathit{sort}
\end{array}$$

6.2.4 A Sorted Model

By making use of the basic theories for the unsorted store, object dependencies, and sorts we can provide a theory for "sorted stores" with dependent objects. The stores are "sorted" in that each object has a sort attribute. We also define an additional concept: well-definedness. An object is *well-defined* with respect to a given store if the value contained in the object (relative to that store) is in the representation set of the object's sort attribute. That is, if *sort* is the sort attribute of the object, then the value it contains must be a valid representation of a value of sort *sort*.

$$
\begin{array}{|l}
\textit{Store} \\
\hline
\textit{UStore} \\
\textit{wellDefObjs} : \mathbb{F}\ \textit{Obj} \\
\hline
x \in \textit{wellDefObjs} \Leftrightarrow \\
\qquad x \in \textit{activeObjs} \land \\
\qquad \textit{val}\ x \in \textit{repSet}(\textit{sortAttr}\ x) \\
\hline
\end{array}
$$

We define the empty store to be the store containing no active objects.

$$
\begin{array}{|l}
\textit{emptyStore} : \textit{Store} \\
\hline
\textit{emptyStore}.\textit{activeObjs} = \varnothing
\end{array}
$$

Every *Store* is uniquely determined by its underlying unsorted store and every unsorted store determines a unique *Store*.

$$(\lambda\ \textit{Store} \bullet \textit{UStore}) \in \textit{Store} \rightarrowtail\!\!\!\!\twoheadrightarrow \textit{UStore}$$

6.2.5 A Sorted Projection of the Store

The theory for the sorted model of the store provides us with all of the expressive power that we need—it can be used to provide a meaning for any[6] LCL specification—but we can increase the conciseness and clarity of the semantic definition if we exploit the fact that we are translating the non-interface part of LCL specifications into *LSL*. For example, the LCL statement

```
x^ = z'
```

is represented in LSL as[7]

[6]Actually, in the current version of the semantic definition, the model of the store can be used to provide a meaning for any LCL specification that only makes use of *static* object dependency relationships.

[7]The LSL notation used here is defined in Section 6.3.1 which presents the LSL formulation of the model of the store. The forward reference is necessary since the organization of the LSL formalization of the store depends on the knowledge that a trait defining sorted projections will be defined.

```
abs(sortAttr(x),val(pre,x)) = abs(sortAttr(z),val(post,z))
```

By making use of LSL operator overloading we can simplify this expression to

```
val(pre,x) = val(post,z)
```

The operators which allow us to achieve such conciseness are defined in a theory which provides what we call a *sorted projection of the store*. For every sort `Obj[S]` that is used in (the LSL formalization of) the non-interface part of an LCL specification, we define an S-sorted projection of the store. An S-sorted projection provides us with a view of the store that includes only those objects whose sort attribute is **S**.

In this section we define a theory for an **S**-sorted project of the store. In our Z formalization of the theory, the sort **S** is represented as a given set.

[S]

The object sort of **S**, usually written as `Obj[S]` in LSL, will be denoted by **Obj_S**. We let s be the name of the sort **S**.

| s : *SortNm*

In this theory we must

- establish the nature of the relationship that is to exist between the unsorted values in U and the sorted values in **S** (Section 6.2.5.1),
- define a bijection between the objects having sort attribute **S** (*objsSort* s) and the set of *object identifiers* **Obj_S** (Section 6.2.5.2),
- define the promoted store operations (Section 6.2.5.3).

We will call the elements of **Obj_S** *object identifiers* and the elements of *Obj objects* so as to make it intuitively clearer that the elements of **Obj_S** are not new objects but merely new labels for some of the objects in *Obj*.

6.2.5.1 Abstraction Function

We assume the existence of an abstraction function that maps unsorted values in U into sorted values in **S**.

$$abs_{\mathrm{s}} : U \twoheadrightarrow \mathbf{S}$$

For our purpose, a precise definition of *abs* is unnecessary. We need only specify the required relationship between the abstraction function and the equality relation *equal* s. Two unsorted values are equal (when regarded as values of sort **S**) if they are both in the domain of the abstraction function and if the abstraction function maps them to the same value.

$$\begin{aligned}&(u_1, u_2) \in \mathit{equal}\ \mathrm{s} \Leftrightarrow \\ &\qquad \{u_1, u_2\} \subseteq \operatorname{dom} abs_{\mathrm{s}} \land abs_{\mathrm{s}}\, u_1 = abs_{\mathrm{s}}\, u_2\end{aligned}$$

Since every value in **S** has a representation, abs_{s} is declared as a surjective function. abs_{s} need not be injective since there can exist more than one unsorted value representing any given sorted value. As a corollary to the relationship between abs_{s} and *equal* s specified above, we have:

$$\operatorname{dom} abs_{\mathrm{s}} =: \mathit{repSet}\ \mathrm{s}$$

6.2.5.2 Bijection Between Objects and Object Identifiers

The function *up* is a bijection between the objects whose sort attributes are **S** and the object identifiers **Obj_S**. A bijection can be regarded as a simple renaming scheme.

$$\begin{aligned}&up : \mathit{objsSort}\ \mathrm{s} \rightarrowtail\!\!\!\twoheadrightarrow \mathbf{Obj\_S} \\ &dwn : \mathbf{Obj\_S} \rightarrowtail\!\!\!\twoheadrightarrow \mathit{objsSort}\ \mathrm{s} \\ \hline &up = dwn^{\sim}\end{aligned}$$

We usually speak of "raising the level of abstraction of ...". Just as the abstraction function abs_{s} can be seen as "raising" unsorted values into values of sort s, *up* can be seen as "raising" *Objs*'s into **Obj_S**'s. Hence the name *up* for the bijection between the objects whose sort attributes are **S** and the object identifiers **Obj_S**, and the name and *dwn* (i.e. "down") for the inverse of *up*.

6.2.5.3 Promoted Operations

We say that an object identifier from **Obj_S** is active (well-defined) if the object, from *Obj*, that it represents is active (respectively, well-defined). The sorted value of a well-defined object identifier tx is the sorted value represented by the unsorted value contained in the object denoted by tx.

The schema *SProjStore* defines an **S**-sorted projection of the store with the promoted inspectors $activeObjs_{\mathbf{S}}$, $val_{\mathbf{S}}$ and $wellDefObjs_{\mathbf{S}}$.

$$
\begin{array}{|l}
\multicolumn{1}{l}{\textit{SProjStore}} \\
\hline
Store \\
activeObjs_{\mathbf{S}} : \mathbb{F}\, \mathbf{Obj\_S} \\
val_{\mathbf{S}} : \mathbf{Obj\_S} \mathrel{+\!\!\!\!\!\twoheadrightarrow} \mathbf{S} \\
wellDefObjs_{\mathbf{S}} : \mathbb{F}\, \mathbf{Obj\_S} \\
\hline
activeObjs_{\mathbf{S}} = \{\, tx : \mathbf{Obj\_S} \mid dwn\ tx \in activeObjs \,\} \\
wellDefObjs_{\mathbf{S}} = \{\, tx : \mathbf{Obj\_S} \mid dwn\ tx \in wellDefObjs \,\} \\
\mathrm{dom}\ val_{\mathbf{S}} = wellDefObjs_{\mathbf{S}} \\
\forall\, tx : wellDefObjs_{\mathbf{S}} \bullet val_{\mathbf{S}}\ tx = abs_{\mathbf{S}}(val(dwn\ tx)) \\
\hline
\end{array}
$$

We could have defined *SProjStore* more concisely as

$$
\begin{array}{|l}
\multicolumn{1}{l}{\textit{SProjStore\_AltDef}} \\
\hline
SProjStore \\
\hline
activeObjs_{\mathbf{S}} = up(\!|activeObjs|\!) \\
wellDefObjs_{\mathbf{S}} = up(\!|wellDefObjs|\!) \\
val_{\mathbf{S}} = abs_{\mathbf{S}} \circ val \circ dwn \\
\hline
\end{array}
$$

Every *SProjStore* is uniquely determined by its underlying unsorted store and every unsorted store determines a unique *SProjStore*.

$$(\lambda\, SProjStore \bullet UStore) \in SProjStore \rightarrowtail\!\!\!\!\!\twoheadrightarrow UStore$$

6.3 An LSL Formalization of the Store

Deriving an LSL formulation of the Z model of the store is rather straightforward. Z types and functions are usually mapped to similarly named LSL sorts and operators.

The most notable exceptions are functions with ranges that are power sets and relations over infinite sets. In these cases we usually represent the function or relation as a boolean operator. Furthermore, in the case of a function, we must explicitly constrain the corresponding boolean operator to be functional over its "domain".

There would appear to be no theoretical limitation preventing us from defining a sort that consists of possibly infinite subsets of values from another sort. On the other hand, such a formalization would allow higher-order functions to be defined—making it dubious to claim that LSL is a first-order formalism. We have chosen to avoid such murky waters.

6.3.1 Store and StorePOps Traits

We have adopted a *literate programming* [Knu92] approach for the documentation of the LSL traits that comprise the Z formalization of the store. That is, the LSL traits and this presentation of them are generated from a single source, thus eliminating the risk of inconsistencies between the two. We have made use of Norman Ramsey's `noweb` system [Ram94]. The theories for the unsorted and sorted models of the store are captured by the `Store` trait. The organization of `Store` follows the typical structure of an LSL trait.

94 ⟨*Store.lsl* 94⟩≡
```
Store: trait
  includes
    ⟨Store include 95b⟩
  introduces
    ⟨Store opsig 96c⟩
  asserts
    \forall ⟨Store assert var decl 95c⟩
        ⟨Store assert eqn 96a⟩
  implies
    ⟨Store implies gen/part 96b⟩
    \forall ⟨Store assert var decl 95c⟩
        ⟨Store implies eqn 97c⟩
```
What is given above is an outline of the contents of the `Store.lsl` file which defines the `Store` trait. The components, like ⟨*Store include* 95b⟩, are defined below.

We make use of an auxiliary trait named `StorePOps` to hold (only and all of) the declarations of the operators over the store that will be subject to promotion.

95a ⟨*StorePOps.lsl* 95a⟩≡
```
StorePOps(U,Obj): trait
  includes
    ⟨StorePOps include 95d⟩
  introduces
    ⟨StorePOps opsig 95f⟩
```

Having such an auxiliary trait will allow us to declare the promoted operators, in a sorted projection of the store, simply by including `StorePOps` with appropriate renamings for the sorts U and `Obj`. Since *Store* captures the theory of the store, it must necessarily include *StorePOps*.

95b ⟨*Store include* 95b⟩≡ (94) 97a▷
```
StorePOps,
```

Here are some of the variables that are used in this trait

95c ⟨*Store assert var decl* 95c⟩≡ (94) 95e▷
```
u, u', u'': U,
x, x': Obj,
any, any': Store,
```

The core material of the traits `Store` and `StorePOps` is presented next. For ease of comparison, we have organized the material under subsections that mimic the presentation of the Z theory (Section 6.2) that these traits are meant to encode.

6.3.1.1 Unsorted Store

We make use of the `Set` trait [GH93, §A.5] to define `Set[Obj]`, the sort representing finite sets of objects.

95d ⟨*StorePOps include* 95d⟩≡ (95a)
```
Set(Obj,Set[Obj])
```

95e ⟨*Store assert var decl* 95c⟩+≡ (94) ◁95c 97b▷
```
xs: Set[Obj],
```

The schemas types *UStore*, *Store* and *SProjStore* are all represented by the sort `Store`. The schema selectors are represented as LSL prefix operators; e.g., the Z expression *any.activeObjs* is written as `activeObjs(any)` in LSL.

95f ⟨*StorePOps opsig* 95f⟩≡ (95a) 99a▷

```
activeObjs: Store -> Set[Obj]
val: Store, Obj -> U
```

`activeObjs` and `val` are operators that will be subject to promotion and hence their declarations are contained in the `StorePOps` trait. In LSL, an operation denotes a *total* function. Thus, we "lose" the fact that (in the Z theory) `val` is a partial function with `activeObjs` as its domain, but we can achieve the same effect by explicitly encoding the following property: a store is fully determined by:

- the set of active objects it contains,
- the values contained in the active objects.

96a ⟨*Store assert eqn* 96a⟩≡ (94) 96d▷

```
(activeObjs(any) = activeObjs(any')
 /\ \A x:Obj (x \in activeObjs(any) =>
                    val(any,x) = val(any',x)))
=> any = any';
```

(The universal quantifier is represented as `\A` in LSL.) This property implies

96b ⟨*Store implies gen/part* 96b⟩≡ (94)

```
Store partitioned by activeObjs, val
```

which states that two stores are equal if they cannot be differentiated by means of the two listed inspectors.

6.3.1.2 Dependencies

The *depOn* and *indep* relations are represented as boolean operators.

96c ⟨*Store opsig* 96c⟩≡ (94) 97d▷

```
depOn: Obj, Obj -> Bool
indep: Seq[Obj] -> Bool
```

96d ⟨*Store assert eqn* 96a⟩+≡ (94) ◁96a 97g▷

```
depOn(x,x);
indep(xq) ==
  \A i \A j (
        i \in inds(xq) /\ j \in inds(xq)
     /\ i \neq j => \not depOn(xq[i],xq[j]));
```

The sort `Seq[Obj]` denotes the sort of finite sequences of objects and is defined in the `Seq` trait (Section 6.5.1).

97a ⟨*Store include* 95b⟩+≡ (94) ◁95b 97e▷
```
  Seq(Obj),
```

97b ⟨*Store assert var decl* 95c⟩+≡ (94) ◁95e 98a▷
```
  xq: Seq[Obj],
  i, j: Int,
```

Two objects (in a sequence) are independent if each is independent of the other.

97c ⟨*Store implies eqn* 97c⟩≡ (94)
```
  indep(empty |- x |- x') ==
     \not depOn(x,x') /\ \not depOn(x',x);
```

6.3.1.3 Sorts

Each object has a fixed sort attribute which is given by **`sortAttr`**. There is at least one object of each sort, i.e., **`sortAttr`** is surjective.

97d ⟨*Store opsig* 96c⟩+≡ (94) ◁96c 97f▷
```
  sortAttr: Obj -> SortNm
```

97e ⟨*Store include* 95b⟩+≡ (94) ◁97a
```
  Surjective(sortAttr,Obj,SortNm)
```

Since the range of *objsSort* is a power set, we will represent it as the relation *objsSameSort* which will hold true of a pair (*sort*, *objs*) if all of the objects in *objs* have *sort* as sort attribute.

$$objsSameSort : SortNm \longleftrightarrow \mathbb{F}\ Obj$$

$$\forall\, sort : SortNm;\ objs : \mathbb{F}\ Obj \bullet$$
$$\quad (sort, objs) \in objsSameSort \Leftrightarrow$$
$$\qquad \forall\, x : objs \bullet sortAttr\ x = sort$$

97f ⟨*Store opsig* 96c⟩+≡ (94) ◁97d 98c▷
```
  objsSameSort: SortNm, Set[Obj] -> Bool
```

97g ⟨*Store assert eqn* 96a⟩+≡ (94) ◁96d 98b▷
```
  objsSameSort(sn,xs) ==
    \A x (x \in xs => sortAttr(x) = sn);
```

98a ⟨*Store assert var decl* 95c⟩+≡ (94) ◁97b

```
sn: SortNm
```

There is an infinite supply of objects of each sort. That is, for any given sort sn and finite set of objects with sort attribute sn, we can always find an object with sort attribute sn that is outside of this set.

98b ⟨*Store assert eqn* 96a⟩+≡ (94) ◁97g 98d▷

```
objsSameSort(sn,xs) =>
   \E x (x \notin xs /\ objsSameSort(sn, {x} \cup xs));
```

Since the ranges of *equal*, *repSet* and *strongEq* are potentially infinite sets, we must change their representations. *equal* and *strongEq* will be represented by the ternary boolean predicates equal and strongEq respectively. *repSet* will be replaced by the binary predicate isRep.

98c ⟨*Store opsig* 96c⟩+≡ (94) ◁97f 99c▷

```
equal: SortNm, U, U -> Bool
isRep: SortNm, U -> Bool
strongEq: SortNm, U, U -> Bool
```

The unsorted value u is in the representation set of sn iff isRep(sn,u). As in the Z formalization of the store, the representation set corresponds to the "domain" of equal(sn,__,__).

98d ⟨*Store assert eqn* 96a⟩+≡ (94) ◁98b 98e▷

```
isRep(sn,u) == \E u' equal(sn,u,u');
```

equal(sn,__,__) is reflexive, symmetric and transitive over the representation set of sn.

98e ⟨*Store assert eqn* 96a⟩+≡ (94) ◁98d 98f▷

```
isRep(sn,u) => equal(sn,u,u);
isRep(sn,u) /\ isRep(sn,u') =>
  (equal(sn,u,u') => equal(sn,u',u));
isRep(sn,u) /\ isRep(sn,u') /\ isRep(sn,u'') =>
  (equal(sn,u,u') /\ equal(sn,u',u'')
    => equal(sn,u,u''));
```

Finally, we define strong equality.

98f ⟨*Store assert eqn* 96a⟩+≡ (94) ◁98e 99b▷

```
strongEq(sn,u,u') ==
    equal(sn,u,u') \/
    (~isRep(sn,u) /\ ~isRep(sn,u'));
```

6.3.1.4 Sorted Store

The only additional definitions that are required are for well-definedness

99a ⟨*StorePOps opsig* 95f⟩+≡ (95a) ◁95f

```
  wellDefObjs: Store -> Set[Obj]
```

99b ⟨*Store assert eqn* 96a⟩+≡ (94) ◁98f 99d▷

```
  x \in wellDefObjs(any) ==
          x \in activeObjs(any)
      /\ isRep(sortAttr(x), val(any,x));
```

and the empty store (which we call **empty** in LSL rather than *emptyStore*).

99c ⟨*Store opsig* 96c⟩+≡ (94) ◁98c

```
  empty: -> Store
```

99d ⟨*Store assert eqn* 96a⟩+≡ (94) ◁99b

```
  activeObjs(empty) == {};
```

6.3.2 SProjStore Trait

The trait SProjStore defines an S-sorted projection of the store. It has the usual overall structure.

99e ⟨*SProjStore.lsl* 99e⟩≡

```
  SProjStore(S): trait
    includes
      ⟨SProjStore include 99f⟩
    introduces
      ⟨SProjStore opsig 100b⟩
    asserts
      \forall ⟨SProjStore assert var decl 100a⟩
          ⟨SProjStore assert eqn 100c⟩
```

A sorted projection makes the link between the sorted model of the store and a projection (i.e. restricted view) of the store that contains only those objects whose sort attributes are S. Thus, it is necessary to include the theory of the sorted model.

99f ⟨*SProjStore include* 99f⟩≡ (99e) 102b▷

```
Store,
```

Here are the variables that are used in this trait.

100a ⟨*SProjStore assert var decl* 100a⟩≡ (99e) 102f▷

```
u, u':U,
s, s':S,
x, x': Obj,
tx, tx': Obj[S],
any: Store,
```

In the Z description of the sorted projection of the store we simply assumed that s was the element of *SortNm* naming S[8]. We follow another approach here: we define a constant function that maps all values of sort S into the sort name of S.

100b ⟨*SProjStore opsig* 100b⟩≡ (99e) 100e▷

```
sortNm: S -> SortNm
```

100c ⟨*SProjStore assert eqn* 100c⟩≡ (99e) 101a▷

```
sortNm(s) == sortNm(s');
```

100d ⟨*sn* 100d⟩≡ (101 102a)

```
sortNm(s)
```

It is necessary to assert that for each pair of distinct sorts S and S' the functions sortNm from the S and S'-sorted projections will have distinct values. This can be achieved by a global assertion or stated as a metalogical property as follows. For each pair of distinct sorts S and S',

$$\forall \texttt{ s:S, s':S' (sortNm(s)} \neq \texttt{sortNm(s'))}$$

holds under the theories of SProjStore(S) and SProjStore(S').

6.3.2.1 Abstraction Function

In LSL, the abstraction function **abs** is modeled as a total function.

100e ⟨*SProjStore opsig* 100b⟩+≡ (99e) ◁100b 101c▷

```
abs: U -> S
```

[8]We could follow this approach in the LSL formalization of the store but it leads to difficulties. With this approach, instantiation of SProjStore, would be achieved by renaming both S and s. This leads to difficulties, for example, when we attempt to define LSL traits defining C array or structure types.

In the Z theory of the store, *abs* is a (partial) surjective function: every sorted value has a pre-image in the domain of *abs*. Since the domain of *abs* coincides with the representation set of ⟨*sn* 100d⟩, we can express the surjectiveness of abs as follows

101a ⟨*SProjStore assert eqn* 100c⟩+≡ (99e) ◁100c 101b▷
```
 \A s \E u (isRep(⟨sn 100d⟩,u) /\ abs(u) = s);
```

Two unsorted values are equal (when viewed as values of sort S) if they are both representations of some value of sort S.

101b ⟨*SProjStore assert eqn* 100c⟩+≡ (99e) ◁101a 101d▷
```
 equal(⟨sn 100d⟩,u,u') ==
   isRep(⟨sn 100d⟩,u) /\ isRep(⟨sn 100d⟩,u') /\ abs(u) = abs(u');
```

6.3.2.2 Bijection on Obj_S

When the domain of up is restricted to the set of objects with sort attribute S, up is injective

101c ⟨*SProjStore opsig* 100b⟩+≡ (99e) ◁100e 101f▷
```
 up:  Obj -> Obj[S]
```

101d ⟨*SProjStore assert eqn* 100c⟩+≡ (99e) ◁101b 101e▷
```
     sortAttr(x) = ⟨sn 100d⟩
  /\ sortAttr(x') = ⟨sn 100d⟩
  /\ up(x) = up(x')   =>  x = x';
```

and surjective.

101e ⟨*SProjStore assert eqn* 100c⟩+≡ (99e) ◁101d 101g▷
```
 \A tx \E x (sortAttr(x) = ⟨sn 100d⟩ /\ up(x) = tx);
```

dwn is injective

101f ⟨*SProjStore opsig* 100b⟩+≡ (99e) ◁101c 102d▷
```
 dwn: Obj[S] -> Obj
```

101g ⟨*SProjStore assert eqn* 100c⟩+≡ (99e) ◁101e 101h▷
```
 dwn(tx) = dwn(tx') => tx = tx';
```

and its range is exactly the set of objects whose sort attribute is S.

101h ⟨*SProjStore assert eqn* 100c⟩+≡ (99e) ◁101g 102a▷
```
 sortAttr(dwn(tx)) = ⟨sn 100d⟩;
 sortAttr(x) = ⟨sn 100d⟩ => \E tx (dwn(tx) = x);
```

Furthermore, up and dwn are inverses

102a ⟨*SProjStore assert eqn* 100c⟩+≡ (99e) ◁101h 102c▷
```
  up(dwn(tx)) = tx;
  sortAttr(x) = ⟨sn 100d⟩ => dwn(up(x)) = x;
```

6.3.2.3 Promoted Operators

We declare the promoted operators simply by including StorePOps with an appropriate renaming.

102b ⟨*SProjStore include* 99f⟩+≡ (99e) ◁99f 103d▷
```
  StorePOps(S,Obj[S]),
```

Here are the definitions of the promoted operators.

102c ⟨*SProjStore assert eqn* 100c⟩+≡ (99e) ◁102a 102e▷
```
  tx \in activeObjs(any) == dwn(tx) \in activeObjs(any);
  tx \in wellDefObjs(any) == dwn(tx) \in wellDefObjs(any);
  tx \in wellDefObjs(any) =>
      val(any,tx) = abs(val(any,dwn(tx)));
```

6.3.2.4 Sugar for Sets and Sequences

In the translations of the non-interface part of LCL specifications into LSL traits we will often need to denote sets and sequences of objects of possibly different sorts. The following infix operations will allow us to write more concise expressions denoting such sets and sequences.

102d ⟨*SProjStore opsig* 100b⟩+≡ (99e) ◁101f 103a▷
```
  __ \ins __: Set[Obj], Obj[S] -> Set[Obj]
```

102e ⟨*SProjStore assert eqn* 100c⟩+≡ (99e) ◁102c 103b▷
```
  xs \ins tx == insert(dwn(tx),xs);
```

102f ⟨*SProjStore assert var decl* 100a⟩+≡ (99e) ◁100a 103c▷
```
  xs: Set[Obj],
```

The operator \ins can be used as a constructor for sets of objects. For example, we will be able to write

```
    empty \ins tx1 \ins tx2 \ins ... txk
```

instead of

```
    insert(dwn(txk), ...
                    insert(dwn(tx2),insert(dwn(tx1),empty))...)
```

Similarly, we define \apd to ease the writing of sequences of objects.

103a ⟨*SProjStore opsig* 100b⟩+≡ (99e) ◁102d

```
  __ \apd __: Seq[Obj], Obj[S] -> Seq[Obj]
```

103b ⟨*SProjStore assert eqn* 100c⟩+≡ (99e) ◁102e

```
  xq \apd tx == xq \postcat dwn(tx);
```

103c ⟨*SProjStore assert var decl* 100a⟩+≡ (99e) ◁102f

```
  xq: Seq[Obj]
```

Sequences are defined in the Seq trait given in Section 6.5.1.

103d ⟨*SProjStore include* 99f⟩+≡ (99e) ◁102b

```
  Seq(Obj)
```

6.4 Traits for LCL Types

The traits in this section provide a semantics for the sorts on which LCL exposed types are based[9].

6.4.1 IntTp Trait

IntTp is meant to be an axiomatization of the C int type. In the current version of the semantic definition we ignore the fact that the int type corresponds to a subrange of the integers and define it in terms of the LSL Integer trait.

103e ⟨*IntTp.lsl* 103e⟩≡

```
  IntTp: trait
    includes Integer(int)
```

[9]In the current release of the semantic model we only provide definitions for the sorts associated with the int and array types. Adding traits for the other basic types is straightforward.

6.4.2 ArrTp Trait

An object that has an array type contains values of sort Arr[E] called *array values*. As usual, such an object is of sort Obj[Arr[E]]. The ArrTp trait defines

- the Arr[E] sort,
- E-sorted and Arr[E]-sorted projections of the store,
- those array value operations that are "promoted" so that they can also be applied to array objects (which are of sort Obj[Arr[E]]), and
- the particular relationships (namely, dependencies, active/inactive status and well-definedness) that exists between array objects and the objects that are their members.

104a ⟨*ArrTp.lsl* 104a⟩≡
```
ArrTp(E): trait
  includes
    ⟨ArrTp include 104c⟩
  asserts
    \forall ⟨ArrTp assert var decl 104b⟩
        ⟨ArrTp assert eqn 106c⟩
  implies
    \forall ⟨ArrTp assert var decl 104b⟩
        ⟨ArrTp implies eqn 107c⟩
```

104b ⟨*ArrTp assert var decl* 104b⟩≡ (104a)
```
i: Int,
x: Obj,
tx: Obj[Arr[E]],
any: Store
```

A theory for array values is given in the Arr trait of Section 6.5.2.

104c ⟨*ArrTp include* 104c⟩≡ (104a) 105a▷
```
Arr(E),
```

The two sorted projections of the store are defined by appropriate trait references to SProjStore.

105a ⟨*ArrTp include* 104c⟩+≡ (104a) ◁104c 105b▷

```
  SProjStore(E),
  SProjStore(Arr[E]),
```

6.4.2.1 Array Operations on Array Objects

Some of the operators over array values are also defined over array objects. These operators, along with some of their basic properties, are contained in the ArrOps trait which is included in ArrTp[10] .

105b ⟨*ArrTp include* 104c⟩+≡ (104a) ◁105a

```
  ArrOps(Obj[E], Obj[Arr[E]])
```

The outline of ArrOps is

105c ⟨*ArrOps.lsl* 105c⟩≡

```
  ArrOps(E, C): trait
    includes
      Integer, Set(Int,Set[Int])
    introduces
      ⟨ArrOps opsig 105d⟩
    asserts
      \forall c:C
          ⟨ArrOps assert eqn 106a⟩
```

More specifically, the operators over array values that are extended to array objects are:

105d ⟨*ArrOps opsig* 105d⟩≡ (105c)

```
  inds: C -> Set[Int]
  __[__]: C, Int -> E
  dim: C -> Int
  maxIndex: C -> Int
```

The term tx[i] denotes the ith member object of the array tx. The other operators yield the index set, dimension and maximum index of the array value contained in

[10]We could not simply include Arr(Obj[E], Obj[Arr[E]] for Arr[Obj[E]]) to define the array operators over array objects since this would incorrectly assert that Obj[Arr[E]] was generated by the array constructor operators.

the array object. The properties of these operators are given in Section 6.4.2.3, with the exception of two general properties which are given next:

106a ⟨*ArrOps assert eqn* 106a⟩≡ (105c)
```
  dim(c) == size(inds(c));
  inds(c) \neq {} =>
     dim(c) = maxIndex(c) + 1;
```

Since we claim that we are defining "promoted" array value operators, then these operators must exists over array values. Furthermore, the general properties of the promoted operators must hold for the array value operators also. We express these requirements by including a reference to ArrOps in the implies section of ArrTp.

106b ⟨*Arr implies trait* 106b⟩≡ (109a)
```
  ArrOps(E, Arr[E])
```

6.4.2.2 Dependencies

The first thing to be defined about array objects is that nature of the dependencies that exist between arrays and their members. An array object depends on each of its member objects and *vice versa.*

106c ⟨*ArrTp assert eqn* 106c⟩≡ (104a) 106d▷
```
  \A i (i \in inds(tx) =>
                depOn(dwn(tx), dwn(tx[i]))
             /\ depOn(dwn(tx[i]), dwn(tx)));
```

Furthermore, an object x depends on an array object tx if and only if, x depends on at least one of the member objects of tx. Similarly, an array object tx depends on an object x, if and only if at least one of the member objects of tx depends on x.

106d ⟨*ArrTp assert eqn* 106c⟩+≡ (104a) ◁106c 107a▷
```
  depOn(x,dwn(tx)) ==
          \E i (i \in inds(tx) /\ depOn(x, dwn(tx[i])));
  depOn(dwn(tx),x) ==
          \E i (i \in inds(tx) /\ depOn(dwn(tx[i]), x));
```

6.4.2.3 Other Relationships Between Arrays and Their Member Objects

Since there exists dependencies between an array object and its member objects, we must clarify the nature of the relationships between the value, active/inactive status and well-definedness of an array and its members. An array object is considered to be active precisely when all of its members are active.

107a ⟨*ArrTp assert eqn* 106c⟩+≡ (104a) ◁106d 107b▷
```
  tx \in activeObjs(any) ==
          \A i (i \in inds(tx) => tx[i] \in activeObjs(any));
```

Consequently, trashing an array object also trashes all of its members and trashing a member renders the array (and hence all other array members) inactive.

The set of indices (and consequently the dimension) of an array object is the same as that for the array value that it contains.

107b ⟨*ArrTp assert eqn* 106c⟩+≡ (104a) ◁107a 107d▷
```
  inds(tx) == inds(val(any,tx));
```

As a consequence of these definitions we have,

107c ⟨*ArrTp implies eqn* 107c⟩≡ (104a)
```
  dim(tx) == dim(val(any,tx));
  dim(tx) > 0 =>
      maxIndex(tx) = maxIndex(val(any,tx));
```

The value contained in the *i*th member object of an array is always in agreement with the *i*th value of the array value contained in the array object. The `i`th member object of the array `tx` is `tx[i]` and the value contained in this object (in an arbitrary state) is `val(any,tx[i]))`. The array value contained in `tx` is `val(any,tx)` and the value associated with the `i`th entry is `val(any,tx)[i]`. Hence,

107d ⟨*ArrTp assert eqn* 106c⟩+≡ (104a) ◁107b 107e▷
```
  \A i (i \in inds(tx) =>
            val(any,tx)[i] = val(any,tx[i]));
```

An array object is well-defined when all of its members are well-defined.

107e ⟨*ArrTp assert eqn* 106c⟩+≡ (104a) ◁107d
```
  tx \in wellDefObjs(any) ==
          (\A i (i \in inds(tx) => tx[i] \in wellDefObjs(any)));
```

6.5 General Traits

6.5.1 Seq Trait

The trait Seq defines sequences, somewhat like Z sequences. Sequences are defined in terms of the *String* trait [GH93, p. 173]. Note that strings have a base index of 0.

108a ⟨*Seq.lsl* 108a⟩≡
```
Seq(E): trait
  includes
    String(E,Seq[E]), Set(Int,Set[Int])
  introduces
    ⟨Seq opsig 108c⟩
  asserts
    \forall ⟨Seq assert var decl 108b⟩
        ⟨Seq assert eqn 108d⟩
  implies
    \forall ⟨Seq assert var decl 108b⟩
        ⟨Seq implies eqn 108e⟩
```

108b ⟨*Seq assert var decl* 108b⟩≡ (108a)
```
e: E, s: Seq[E], i: Int
```

In addition to the operators over strings, we define inds(s) to be the set of indices of the elements of the sequence s[11].

108c ⟨*Seq opsig* 108c⟩≡ (108a)
```
inds: Seq[E] -> Set[Int]
```

108d ⟨*Seq assert eqn* 108d⟩≡ (108a)
```
inds(empty) == {};
inds(s |- e) == insert(len(s), inds(s));
```

The indices of a sequence are the integers between 0 and $len(s) - 1$, and there are as many indices as there are elements in the sequence.

108e ⟨*Seq implies eqn* 108e⟩≡ (108a)
```
i \in inds(s) ==  0 <= i /\ i < len(s);
size(inds(s)) == len(s);
```

[11] We provide a recursive definition for **inds** since this simplifies proofs (in particular those written using LP) involving **inds**.

6.5.2 Arr Trait

Our formalization of array values is mainly based on sequences as defined in the Seq trait.

109a ⟨*Arr.lsl* 109a⟩≡
```
Arr(E): trait
  includes
    Seq(E, Arr[E] for Seq[E])
  introduces
    ⟨Arr opsig 109b⟩
  asserts
    \forall ⟨Arr assert var decl 109c⟩
        ⟨Arr assert eqn 109d⟩
  implies
    ⟨Arr implies trait 106b⟩
```

In addition to the usual operator for extracting the value at the *i*th index of an array, `__[__]`, we also provide operators that yield the number of elements in an array (called the array *dimension*) and the maximum index of an array.

109b ⟨*Arr opsig* 109b⟩≡ (109a)
```
dim: Arr[E] -> Int
maxIndex: Arr[E] -> Int
```

The array dimension is the same as the length of the array when it is viewed as a sequence. If there is more than one element in an array then, the maximum index is one less than the dimension (since array indices start at 0).

109c ⟨*Arr assert var decl* 109c⟩≡ (109a)
```
a: Arr[E]
```

109d ⟨*Arr assert eqn* 109d⟩≡ (109a)
```
dim(a) == len(a);
dim(a) > 0 => maxIndex(a) = dim(a) - 1;
```

6.5.3 Properties of Functions

A function is surjective if every element in its range has a pre-image.

109e ⟨*Surjective.lsl* 109e⟩≡
```
Surjective(f,Dom,Ran):trait
```

```
introduces
  f: Dom -> Ran
asserts \forall d: Dom, r: Ran
  \A r \E d (f(d) = r)
```

6.6 Summary and Related Work

Our model of program states is conventional: the program state is partitioned into an environment and a store. Our model of the store is exceptional in that it covers object dependencies in their full static generality. Furthermore, an important characteristic of the model is that each object has a fixed, state-independent sort attribute. This is in contrast to C where objects are untyped; the type with which an object is viewed depends on the lvalue that is used to refer to the object [ISO, §3]. As a consequence of this, we represent (what would be considered in C as) a single object with several types by a collection of objects (each with a single type) related by certain object dependencies.

In the subsections that follow, we discuss the storage models for LCL (as defined by Tan), LCPP and LM3. Our discussion is guided by the requirements for the storage model that we identified in Section 6.1.

6.6.1 Tan's LCL Model of the Store

Tan informally describes the model of the store using the following domain equations [Tan94, §7.1]:

$$\begin{aligned}
&\mathit{values} \equiv \mathit{bvalues} \cup \mathit{objects}\\
&\mathit{states} \equiv \mathit{objects} \longrightarrow \mathit{values}\\
&\mathit{objects} \equiv \mathit{mutable\_objects} \cup \mathit{exposed\_objects}\\
&\mathit{exposed\_objects} \equiv \mathit{locations} \cup \mathit{structs} \cup \mathit{unions} \cup\\
&\qquad\qquad \mathit{arrays} \cup \mathit{pointers}
\end{aligned}$$

His formalization of the store is captured as two traits. Unfortunately no link is made between his model (described by domain equations) and the two traits. In particular, the fact that `objects` is a (disjoint) union of various kinds of object is not expressed in his formal model. Tan defines a trait named **state** that describes an unsorted model of the store. The trait typedObj provides sorted projections of the store [Tan94, §7.2]. Tan's model allows us to determine the set of active objects. It is not possible though to determine the sort associated with an object; it is not clear whether objects have a fixed (state independent) sort attribute—informally Tan states that they do, but this claim is not supported by his model.

Although Tan states that aggregate and union objects are viewed as collections of objects, he does not formally model this property. Dependencies between aggregate objects and their members are only partly modeled using the concept of *base objects*. The set of base objects of an object of a fundamental or an abstract type is the singleton set consisting of the object itself. The set of base objects of an aggregate object is defined as the union of the base objects of its members [Tan94, §7.4.1]. Tan does not model dependencies among the members of union types nor dependencies between instances of abstract types.

6.6.2 LCPP

The model of the store used for LCPP [LC95] is similar to Tan's model. Formally, the model of the store is captured by two traits. The State trait provides an unsorted model of the store and the TypedObj trait defines a sorted projection of the store. The LCPP model allows us to denote the set of all active objects and, as was the case with Tan's model, it is not possible to determine the sort of an object[12].

An "object", say x, of an array type is modeled as the collection of objects that constitute its members but x itself is not considered to be an object. On the other hand, an object of a structure or union type is consider to be an object.

A restricted form of object dependency relationships can be modeled in LCPP using an object containment relation. The object containment relation coincides with a transitive subrelation of the general object dependency relation. This approach does not model union types in their full generality. For example, it is not possible to explain the behavior of the call changeAorB(flag,&u.i,&u.c).

```
union { int i; char c; } u;

void changeAorB(Bool changeA, int *a, char *b) {
  modifies *a, *b;
```

[12]Of course the issue is more complicated in LCPP because of subtyping.

```
        ensures if   changeA
                then (*a)' != (*a)^
                  ∧ indep(*a,*b) ⇒ (*b)' = (*b)^
                else (*b)' != (*b)^
                  ∧ indep(*a,*b) ⇒ (*a)' = (*a)^;
    }
```

The LCPP model does not cope with dependencies between instances of abstract types.

6.6.3 LM3

Jones models the store as a collection of functions of the form

```
    σ: Obj[S], State → S
```

There is one such function for each LSL sort S that occurs in (the translation of) an LM3 specification. His formalization of the store is not based on an underlying unsorted model. Arrays (and presumably records too) are treated as "entire objects"; that is, an assignment to an array element is regarded as a change in value of the entire array object—such a model for array types is inadequate for LCL. No treatment of union objects is described.

There seems to be no notion of active or inactive objects. There is no formal expression corresponding to the set of all possible objects. The only property of an object that is modeled is the abstract value that it contains. Since the collection of functions defining the store are sorted, it is impossible to determine if an object is well-defined. Object dependency relationships are not modeled (neither for exposed nor abstract types).

Chapter 7

Semantic Objects

The semantic object that corresponds to an LCL specification is an *LCL environment* (also referred to simply as an *environment* when there is no possibility of confusion). An environment has two components one of which captures a specification's interface and the other assigns a meaning to the components that are part of the interface. The interface is documented as an *LCL signature* and the meanings of the interface components are captured in the form of an LSL trait body.

Since LCL specifications make use of LSL traits we must deal with interface components from two languages. Thus, an LCL signature consists of an LSL signature (*LSig*, Section 4.4.1) together with a semantic object—named an *ISig*—used to hold LCL specific information about the interface components.

7.1 Identifiers

In Table 2, we list the various entities that an identifier can denote in a C program after preprocessing has been performed [ISO, §6.1.2]. In addition, an identifier in an LCL specification can denote any one of the entities shown in Table 3. Identifiers can be classified by the context in which they are used, thus resulting in different *name spaces.* This allows for distinct entities to share the same identifier. LCL has the same identifier classification scheme as C: the name spaces are [ISO, §6.1.2.3]:

- *label names*,
- *tags* of structure, union and enumerated types,

An identifier can name ...	Name Space
enumeration constant	ordinary id
function	ordinary id
label	label name
object	ordinary id
structure or union member	member name
tag of a structure, union or enumeration	tag
`typedef` name	ordinary id

Table 2: Classification of C Identifiers

An identifier can name ...	Name Space
LCL constant	ordinary id
LCL abstract type name	ordinary id
logical variable	ordinary id
LSL operator	ordinary id
LSL sort	ordinary id

Table 3: Classification of LCL Specific Identifiers

- *member names* of structure and union types—each type corresponds to a distinct name space,
- the remaining identifiers are classified as *ordinary identifiers.*

The name spaces attributed to the uses of identifiers shown in Tables 2 and 3 are also given.

We will need to name entities unambiguously, hence we define *IdNmSp* to be an identifier together with the name space to which it is meant to belong.

```
IdNmSp
  id : Id
  nmSp : NmSp
```

In our informal discussions we overload the term "identifier" by using it to refer to members of *Id* as well as *IdNmSp*. The formal text should clarify which is being referred to. We need to provide names only for the name spaces consisting of tags and ordinary identifiers—label names are not used in LCL and member names are used in very specific (and easily distinguishable) contexts.

$$
\begin{aligned}
\mathit{NmSp} ::= \; & \mathit{OrdNmSp} \\
\mid \; & \mathit{TagNmSp}
\end{aligned}
$$

Since ordinary identifiers occur so frequently, it is useful to have a function that maps an ordinary *Id* into the corresponding *IdNmSp*:

$$
\begin{array}{|l}
\mathit{ordId} : \mathit{Id} \longrightarrow \mathit{IdNmSp} \\
\hline
\mathit{ordId}\; \mathit{id}' = (\mu\, \mathit{IdNmSp} \mid \\
\qquad \mathit{id} = \mathit{id}' \wedge \\
\qquad \mathit{nmSp} = \mathit{OrdNmSp})
\end{array}
$$

A declaration in an LCL specification can introduce an identifier as an abstract type, an LCL constant, an enumeration constant, a function, a logical variable, a **typedef** name, a variable, or a tag. Formally[1]

$$
\begin{aligned}
\mathit{IdKind} ::= \; & \mathit{AbsTpIdK} \\
\mid \; & \mathit{ConIdK} \\
\mid \; & \mathit{EnumCIdK} \\
\mid \; & \mathit{FunIdK} \\
\mid \; & \mathit{LVarIdK} \\
\mid \; & \mathit{TpDefIdK} \\
\mid \; & \mathit{VarIdK} \\
\mid \; & \mathit{TagIdK}
\end{aligned}
$$

7.2 Types

LCL types are represented as LSL sorts in LSL signatures and as LCL type denotations in *ISig*'s. Type denotations are discussed in Section 7.2.1 and sorts, as related to LCL types, are discussed in Section 7.2.5.

[1]Only the kinds *ConIdK*, *FunIdK*, *LVarIdK*, *VarIdK* are used in the current version of the semantic definition.

7.2.1 Type Denotations

A denotation for an unqualified type name is called an *unqualified type denotation*, and a denotation for a type name is called a *type denotation* (implicitly understood as being qualified). A type denotation consists of an unqualified type denotation and a set of type qualifiers.

$$\begin{array}{|l}
\hline \textit{TpDenG}[\textit{XUTpDen}] \\
\textit{tpQuals} : \mathbb{F}\ \textit{TpQual} \\
\textit{utd} : \textit{XUTpDen} \\
\hline
\end{array}$$

$$\textit{TpDen} \mathrel{\widehat{=}} \textit{TpDenG}[\textit{UTpDen}]$$

$$\begin{array}{|l}
\textit{utd2TD} : \textit{UTpDen} \longrightarrow \textit{TpDen} \\
\hline
\textit{utd2TD}\ \textit{utd} = (\textbf{let}\ \textit{tpQuals} == \varnothing \bullet \theta\,\textit{TpDen})
\end{array}$$

7.2.2 Unqualified Type Denotations

Each unqualified type has a denotation. Notice that the denotation of an array consists of a denotation for the element type and an optional LSL term: thus, the optional LCL expression is represented as an LSL term.

$$\begin{array}{lcl}
\textit{OptTerm} & ::= & \textit{NoTerm} \\
& | & \textit{OneTerm}\langle\!\langle \textit{Term} \rangle\!\rangle \\
\textit{UTpDen} & ::= & \textit{VoidUTD} \\
& | & \textit{BasicUTD}\langle\!\langle \textit{BasicTp} \rangle\!\rangle \\
& | & \textit{ArrUTD}\langle\!\langle \textit{TpDenG}[\textit{UTpDen}] \times \textit{OptTerm} \rangle\!\rangle \\
& | & \textit{FunUTD}\langle\!\langle \textit{FunUTDSch}[\textit{UTpDen}] \rangle\!\rangle \\
& | & \textit{ImmUTD}\langle\!\langle \textit{userSN} \rangle\!\rangle \\
& \vdots &
\end{array}$$

A denotation for a function type contains the denotation of the function return type, parameter types and the types of the variables in the global variable list.

$FunUTDSch0[XUTpDen]$

$retUTD : XUTpDen$

$prmTDs : \mathrm{seq}\ TpDenG[XUTpDen]$

$FunUTDSch[XUTpDen]$

$FunUTDSch0[XUTpDen]$

$prmIds,$

$gvarIds : \mathrm{seq}\ Id$

$gvarUTDs : \mathrm{seq}\ XUTpDen$

$\#prmIds = \#prmTDs$

$\#gvarIds = \#gvarUTDs$

7.2.3 Object, Function and Incomplete Types

In C, types are partitioned into [ISO, §6.1.2.5]:

- *object types*
- *function types*
- *incomplete types*

The object and function types are the types that objects and functions can have, respectively. The incomplete types are:

- `void`,
- array types for which the array dimension is not given,
- structure or union types with unspecified members; this occurs when only the tag is given for the type. Consider the following declarations for mutually recursive structures.

```
struct S;                                /* (a) */
struct T { struct S *ptr_to_s; ... };
struct S { struct T *ptr_to_t; ... };   /* (b) */
```

At point (a), the structure type '`struct S`' is incomplete but it is completed by (b).

All incomplete types, with the exception of `void`, can be completed and when this is done, the result is an object type. Hence, the collection of incomplete types other than `void` will be referred to as "incomplete object types". Formally, *objUTD* is the set of object types, and *incObjUTD* is the set of incomplete object types[2].

$$
\begin{array}{|l}
objUTD, \\
incObjUTD : \mathbb{P}\ UTpDen \\
\hline
objUTD = \\
\quad \operatorname{ran} BasicUTD \cup \\
\quad \{utd : \operatorname{ran} ArrUTD \mid \\
\quad\quad second(ArrUTD^{\sim}\ utd) \neq NoTerm\} \cup \\
\quad \operatorname{ran} ImmUTD \\
\\
incObjUTD = \{utd : \operatorname{ran} ArrUTD \mid \\
\quad\quad\quad second(ArrUTD^{\sim}\ utd) = NoTerm\}
\end{array}
$$

7.2.4 Type Denotation Components

We say that the unqualified type denotation utd_c is an *immediate type component* of *utd* if utd_c is used directly as a part of the type definition of *utd*: formally, iff $(utd, utd_c) \in cptUTD$. Only derived types have immediate type components, fundamental and abstract types do not.

$$
\begin{array}{|l}
cptUTD : UTpDen \leftrightarrow UTpDen \\
\hline
(VoidUTD, utd) \notin cptUTD \\
(BasicUTD\ tp, utd) \notin cptUTD \\
(ArrUTD(td, optTm), utd') \in cptUTD \Leftrightarrow utd' = td.utd \\
(FunUTD(\theta FunUTDSch, utd') \in cptUTD \Leftrightarrow \\
\quad utd' = retUTD \lor utd' \in \operatorname{ran} prmTDs \\
(ImmUTD\ nm, utd) \notin cptUTD
\end{array}
$$

The element type of an array type *utd* is the only proper component type of *utd*. The types of the members of a structure or union type *utd* are the component types

[2]The given definitions are tailored to the current version of the semantics.

of *utd*. The component types of a function type are the function return type and the types of the parameters (hence, excluding the types of the global variables listed in the function header).

In the usual way, we define the *type component* and *proper type component* relations as the reflexive-transitive and transitive closures of *cptUTD* respectively. That is, utd_c is a type component of *utd* if

- utd_c is utd, or
- utd_c is an immediate type component of utd' and utd' is a type component of utd;

that is, if $(utd, utd_c) \in cptUTD^*$. utd_c is a proper type component of *utd* if it is a type component of *utd* that is distinct from *utd*; i.e., if $(utd, utd_c) \in cptUTD^+$. The set of all type components of a given unqualified type *utd* is $cptUTD^* (\!| \{utd\} |\!)$.

7.2.5 Sorts and Types

For the purpose of defining a semantics for LCL, we assume that the set of sort names, *SortNm*, is defined as if by the following free type definition[3].

$$
\begin{aligned}
SortNm ::= {} & \langle\!\langle UserSN \rangle\!\rangle \\
| {} & BasicSN \langle\!\langle BasicTp \rangle\!\rangle \\
| {} & ObjSN \langle\!\langle SortNm \rangle\!\rangle \\
| {} & ArrSN \langle\!\langle SortNm \rangle\!\rangle \\
\vdots {} &
\end{aligned}
$$

In (partial) imitation of this definition, we define

$$
\begin{array}{|l}
userSN : \mathbb{P}\, SortNm \\
basicSN : BasicTp \rightarrowtail SortNm \\
objSN : SortNm \rightarrowtail SortNm \\
arrSN : SortNm \rightarrowtail SortNm \\
\hline
\text{disjoint } \langle userSN, \text{ran } basicSN, \text{ran } objSN, \text{ran } arrSN \rangle
\end{array}
$$

where

[3]The given free type definition is actually not legal Z since the first alternative does not have a 'constructor' name.

- *userSN* denotes the set of all sort names that do not correspond to names representing LCL exposed types,
- *basicSN* maps a basic type into its corresponding sort name,
- *objSN S* is the sort name for objects containing values of sort S,
- *arrSN S* is the sort name for array values—not array objects—with elements of sort S.

We require that every sort name built with these constructors be uniquely decomposable—hence *basicSN*, *objSN* and *arrSN* are defined as injective functions whose ranges are disjoint. Informally, we can define these functions as follows

$$\begin{aligned}&\mathit{basicSN}\ \mathit{IntTp} = \texttt{int}\\&\mathit{objSN}\ \texttt{S} = \texttt{Obj[S]}\\&\mathit{arrSN}\ \texttt{S} = \texttt{Arr[S]}\end{aligned}$$

7.2.6 Sort Name Components

The sort name $sort_c$ is an *immediate sort name component* of *sort*—formally $(sort, sort_c) \in cptSN$—if *sort* can be directly constructed from $sort_c$.

$$\begin{array}{|l}\mathit{cptSN} : \mathit{SortNm} \leftrightarrow \mathit{SortNm}\\\hline (sort, sort_c) \in \mathit{cptSN} \Leftrightarrow\\\qquad sort = \mathit{objSN}\ sort_c \lor\\\qquad sort = \mathit{arrSN}\ sort_c\end{array}$$

Hence

$$\mathit{cptSN} = \mathit{objSN}^{\sim} \cup \mathit{arrSN}^{\sim}$$

As was done in Section 7.2.4, we define the *sort name component* relation as cptSN^* and the *proper sort name component* relation as cptSN^+. The set of all sort name components of *sort* is given by $\mathit{cptSN}^*(\!|\{sort\}|\!)$.

7.3 Signatures

An LCL signature consists of two LSL signatures and an *ISig*. One LSL signature (*localLSig*) contains entries for the declarations that are local to the interface. The other signature contains information for all declarations that are part of the interface, including those derived from imports and uses components.

Sig

$\mathit{isig} : \mathit{ISig}$
$\mathit{lsig},$
$\mathit{localLSig} : \mathit{LSig}$

ISig's are described in the next section and operations over LCL signatures are defined in Section 7.3.2.

7.3.1 *ISig*'s

An *ISig* is a signature that holds LCL specific information about the components in a specification. The main component of an *ISig* is a binding (b) from *IdNmSp* identifiers to their attributes.

ISig

$b : \mathit{IdNmSp} ⇻ \mathit{Attr}$
$\mathit{types} : \mathbb{F}\ \mathit{UTpDen}$

The *types* component is used to record the types that have been used in the specification. This component is necessary for ensuring that each complete type specifier denoting a structure or union type is assigned a unique type. For example, the variables **x1** and **x2**

```
struct { int i; } x1;
struct { int i; } x2;
```

have different types even though they are declared with the same type name[4].

The attributes associated with identifiers in an *ISig* are:

[4]See [ISO, §6.5.2.3] for more information concerning how types are assigned to structure and union type names.

- the use that is being made of the identifier (formally represented by an *IdKind*),
- the type associated with the identifier, and
- an indication of whether the identifier is **spec** or non-**spec**.

```
┌─ Attr ──────────────────────────────
│ idKind : IdKind
│ td : TpDen
│ dclQual : DclQual
└─────────────────────────────────────
```

For example, the specification

```
constant int N;
spec int a[N];
```

corresponds to the following signature

$$(\mu\, ISig \mid b = \{\mathtt{N} \mapsto N\_Attr, \mathtt{a} \mapsto a\_Attr\} \wedge$$
$$types = \{\mathtt{int}, \mathtt{int[N]}\})$$

where

$$N\_Attr = (\mu\, Attr \mid$$
$$idKind = ConIdK \wedge$$
$$td = utd2TD(BasicUTD\ IntTp) \wedge$$
$$dclQual = NotSpecQ)$$

$$a\_Attr = (\mu\, Attr \mid$$
$$idKind = VarIdK \wedge$$
$$td = utd2TD(ArrUTD(\mathtt{int}, OneTerm\ \mathtt{N})) \wedge$$
$$dclQual = SpecQual)$$

The empty *ISig* corresponds to the empty specification: no identifiers are declared and no types are in use.

```
│ emptyISig : ISig
├──────────────
│ emptyISig = (μ ISig | b = ∅ ∧ types = ∅)
```

The function *mkISig* constructs an *ISig* from a given identifier and attribute.

$$
\begin{array}{|l}
mkISig : IdNmSp \times Attr \longrightarrow ISig \\
\hline
mkISig(ip, attr) = \\
\quad (\mu\, ISig \mid \\
\quad\quad b = \{ip \mapsto attr\} \wedge \\
\quad\quad types = cptUTD^{*}(\!|\{attr.td.utd\}|\!))
\end{array}
$$

The binding contains only the given identifier/attribute pair and the types are the component types of the type denotation associated with the identifier.

Two *ISig*'s can be joined: the resulting *ISig* has all of the types and bindings of the two given signatures, unless some identifiers are bound by both signatures. If an identifier is present in the bindings of both signatures, then the binding in the second signature takes precedence.

$$
\begin{array}{|l}
addISig : ISig \times ISig \longrightarrow ISig \\
\hline
addISig = (\lambda\, ISig'; ISig'' \mid \\
\quad (\mu\, ISig \mid b = b' \oplus b'' \wedge \\
\quad\quad types = types' \cup types''))
\end{array}
$$

ISig forms a monoid with *emptyISig* as unit and *addISig* as signature composition operator.

Given an identifier *ip* that has a binding in *isig'*, then *isigExport isig' ip* is a new signature that differs (if at all) from *isig'* only in that the spec attribute of *ip* is set to *NotSpecQ*.

$$
\begin{array}{|l}
isigExport : ISig \longrightarrow IdNmSp \mathrel{+\!\!\!\rightarrow} ISig \\
\hline
isigExport\, \theta ISig' = \\
\quad (\lambda\, ip : IdNmSp \mid ip \in \operatorname{dom} b' \bullet \\
\quad (\mu\, ISig; Attr \mid \\
\quad\quad \theta Attr = b'\, ip \wedge \\
\quad\quad b = b' \oplus \{ip \mapsto \\
\quad\quad\quad (\text{let } dclQual == NotSpecQ \bullet \theta Attr)\} \wedge \\
\quad\quad types = types' \bullet \\
\quad\quad \theta ISig))
\end{array}
$$

It is sometimes necessary to restrict the visibility of global declarations. For example, when processing expressions in the body of a function specification, we limit access to the global variables that are in the global variable list of the function specification header. *isigHideIdNSs isig idNSs* is like the signature *isig* but with the declarations of the identifiers in *idNSs* hidden.

$$
\begin{array}{|l}
\mathit{isigHideIdNSs} : \mathit{ISig} \longrightarrow \mathbb{F}\ \mathit{IdNmSp} \rightarrowtail\!\!\!\!\rightarrow \mathit{ISig} \\
\hline
\mathit{isigHideIdNSs} = \\
\quad (\lambda\ \mathit{ISig}' \bullet \\
\quad (\lambda\ \mathit{idNSsToHide} : \mathbb{F}\ \mathit{IdNmSp} \mid \\
\qquad \mathit{idNSsToHide} \subseteq \mathrm{dom}\ b' \bullet \\
\qquad (\mu\ \mathit{ISig} \mid \\
\qquad\quad b = \mathit{idNSsToHide} \ntriangleleft b' \land \\
\qquad\quad \mathit{types} = \mathit{types}')))
\end{array}
$$

7.3.2 *Sig*'s

The empty LCL signature contains no entries.

$$
\begin{array}{|l}
\mathit{emptySig} : \mathit{Sig} \\
\hline
\mathit{emptySig} = (\mu\ \mathit{Sig} \mid \\
\qquad\quad \mathit{isig} = \mathit{emptyISig} \land \\
\qquad\quad \mathit{lsig} = \mathit{emptyLSig} \land \\
\qquad\quad \mathit{localLSig} = \mathit{emptyLSig})
\end{array}
$$

Two LCL signatures can be joined by joining their corresponding component signatures.

$$
\begin{array}{|l}
\mathit{addSig} : \mathit{Sig} \times \mathit{Sig} \longrightarrow \mathit{Sig} \\
\hline
\mathit{addSig} = (\lambda\ \mathit{Sig}'; \mathit{Sig}'' \bullet \\
\quad (\mu\ \mathit{Sig} \mid \\
\qquad \mathit{isig} = \mathit{addISig}(\mathit{isig}', \mathit{isig}'') \land \\
\qquad \mathit{lsig} = \mathit{addLSig}(\mathit{lsig}', \mathit{lsig}'') \land \\
\qquad \mathit{localLSig} = \mathit{addLSig}(\mathit{localLSig}', \mathit{localLSig}''))
\end{array}
$$

We define *sigResVar ips sig* to be the LCL signature which is obtained from *sig* by restricting the variable identifiers that it contains to those that occur in *ips*.

$$
\begin{array}{|l}
sigResVar : Sig \longrightarrow \mathbb{F}\ IdNmSp \rightarrowtail\!\!\!\!\!\!\!\rightarrow Sig \\
\hline
sigResVar = (\lambda\ Sig' \bullet \\
\quad (\lambda\ varsToShow : \mathbb{F}\ IdNmSp \mid \\
\quad\quad varsToShow \subseteq \mathrm{dom}\ b' \land \\
\quad\quad \text{“All } ip \in varsToShow \text{ are variables”} \bullet \\
\quad\quad \text{“let } varsToHide == \ldots;\ opsToHide == \ldots\text{”} \bullet \\
\quad\quad (\mu\ Sig \mid \\
\quad\quad\quad isig = isigHideIdNSs\ isig'\ varsToHide \land \\
\quad\quad\quad lsig = lsigHideOps\ lsig'\ opsToHide \land \\
\quad\quad\quad localLSig = lsigHideOps\ localLSig'\ opsToHide)))
\end{array}
$$

where “All $ip \in varsToShow$ are variables” is

$$\forall\ ip : varsToShow \bullet (isig'.b\ ip).idKind = VarIdK$$

and “**let** $varsToHide == \ldots;\ opsToHide == \ldots$” is

$$
\begin{array}{l}
\textbf{let}\ allVars == \{ip : \mathrm{dom}\ isig'.b \mid \\
\qquad\qquad (isig'.b\ ip).idKind = VarIdK\} \bullet \\
\textbf{let}\ varsToHide == allVars \setminus varsToShow \bullet \\
\textbf{let}\ opsToHide == \\
\qquad \text{“}\textbf{let}\ id2OpNS == \ldots\text{”} \bullet id2OpNS(\!|varsToHide|\!)
\end{array}
$$

We define “**let** $id2OpNS == \ldots$” as

$$
\begin{array}{l}
\textbf{let}\ id2OpNS == \\
\quad (\lambda\ ip : IdNmSp \bullet \\
\quad\quad \textbf{let}\ td == (isig'.b\ ip).td \bullet \\
\quad\quad \textbf{let}\ opSig == mkConOpSig(objSN(td2SN\ td)) \bullet \\
\quad\quad\quad ip.id \mapsto opSig)
\end{array}
$$

7.4 Environments

An LCL environment consists of an LCL signature and a “special” trait body.

$$\begin{array}{|l}
\hline \textit{Env} \\
\textit{Sig} \\
\textit{TBS} \\
\textit{sig} : \textit{Sig} \\
\hline
\textit{sig} = \theta \textit{Sig} \\
\hline
\end{array}$$

Each *Env* is uniquely determined by its *isig* and *tb* components since *tb* determines the value of all *Env* components other than *isig*.

$$(\lambda\, \textit{Env} \bullet (\textit{isig}, \textit{tb})) \in \textit{Env} \rightarrowtail\!\!\!\!\twoheadrightarrow (\textit{ISig} \times \textit{TraitBody})$$

The empty environment has an empty signature and an empty trait body.

$$\begin{array}{|l}
\textit{emptyEnv} : \textit{Env} \\
\hline
\textit{emptyEnv} = (\mu\, \textit{Env} \mid \\
\qquad\qquad \textit{isig} = \textit{emptyISig} \land \\
\qquad\qquad \textit{tb} = \langle\rangle)
\end{array}$$

From a given identifier/identifier-attribute pair and a trait body, *mkEnv* yields the environment constructed from these components:

$$\begin{array}{|l}
\textit{mkEnv} : (\textit{IdNmSp} \times \textit{Attr}) \times \textit{TraitBody} \longrightarrow \textit{Env} \\
\hline
\textit{mkEnv}(\textit{ip} \mapsto \textit{attr}, \textit{tb}') = \\
\qquad (\mu\, \textit{Env} \mid \textit{isig} = \textit{mkISig}(\textit{ip}, \textit{attr}) \land \textit{tb} = \textit{tb}')
\end{array}$$

Two environments can be joined by joining their component signatures and trait bodies.

$$\begin{array}{|l}
\textit{addEnv} : \textit{Env} \times \textit{Env} \longrightarrow \textit{Env} \\
\hline
\textit{addEnv} = (\lambda\, \textit{Env}'; \textit{Env}'' \bullet \\
\qquad (\mu\, \textit{Env} \mid \\
\qquad\qquad \textit{isig} = \textit{addISig}(\textit{isig}', \textit{isig}'') \land \\
\qquad\qquad \textit{tb} = \textit{tb}' \frown \textit{tb}''))
\end{array}$$

If *ip* is an identifier in *env*, then *envExport env ip* is the environment that differs (if at all) from *env* only in the spec attribute of *ip* (which is guaranteed to be *NotSpecQ*).

$$
\begin{array}{|l}
envExport : Env \longrightarrow IdNmSp \rightarrowtail\!\!\!\!\rightarrow Env \\
\hline
envExport = (\lambda\, Env' \mid \\
\quad (\lambda\, ip : IdNmSp \mid ip \in \mathrm{dom}\ isig'.b \bullet \\
\quad\quad (\mu\, Env \mid \\
\quad\quad\quad isig = isigExport\ isig'\ ip \land \\
\quad\quad\quad tb = tb'))
\end{array}
$$

Chapter 8

Semantic Rules and Semantic Functions

The semantics of LCL is given principally by means of an inference system. The inference rules allow us to establish the validity of *elaboration predicates* of the form

$$c \vdash a \Rrightarrow x \tag{2}$$

Such a predicate asserts that the syntactic object a corresponds to the semantic object x under the context c; we will also say "*a elaborates* to x under c." The context will usually be a signature or an environment containing the declarations under which elaboration is to be performed. The rules of inference, consisting of zero or more hypotheses and a conclusion, are written in the form

$$\frac{hyp_1 \quad hyp_2 \quad \ldots \quad hyp_n}{concl} \qquad \boxed{\text{Name}}$$

where 'Name' is the rule name. The conclusion will always be an elaboration predicate. A hypothesis can be any predicate—including an elaboration predicate.

The inference rules define a proof system that can be used to present deductions of elaboration predicates. It is also possible to view the meaning of an inference rule as the predicate

$$\begin{aligned} &\forall\, v_1 : V_1;\ \ldots;\ v_h : V_h \bullet \\ &\qquad (\exists\, w_1 : W_1;\ \ldots;\ w_k : W_k \bullet \textstyle\bigwedge_{i=1}^{n} hyp_i) \Rightarrow concl \end{aligned}$$

where $v_1, \ldots, v_h$ are the free variables that occur in the conclusion, and $w_1, \ldots, w_k$

are the free variables that occur in the hypotheses excluding those already present in the conclusion.

The elaboration predicate

$$c \vdash_{\mathrm{R}} a \Rrightarrow x$$

is actually a mix-fix representation of the ordinary Z predicate

$$(c, a, x) \in R$$

(Notice that in (2) we omitted the relation name. This will be done when it is clear from the context which relation is being referred to or if the relation name is irrelevant to the discussion.) An elaboration predicate is simply an assertion that a given triple is a member of an ternary *elaboration relation*

$$R : \mathbb{P}(C \times A \times X)$$

which will also be written in mix-fix notation as

$$C \vdash_{\mathrm{R}} A \Rrightarrow X \tag{3}$$

In the sections that follow, inference rules are used to define one or more elaboration relations for each major syntactic class.

8.1 The *Seq* Metarule

Given a relation R

$$C \vdash_{\mathrm{R}} A \Rrightarrow X$$

that defines the elaboration of A's into X's under the context of C's, then the relation *Seq R*

$$C \vdash_{\mathrm{Seq\,R}} \mathrm{seq}\, A \Rrightarrow \mathrm{seq}\, X$$

(which defines the elaboration of sequences of A's into sequences of X's under the same context as for R) is defined as follows.

$$\frac{}{c \vdash_{\mathrm{Seq\,R}} \langle\rangle \Rrightarrow \langle\rangle} \qquad \boxed{\mathit{Seq}\text{-b}}$$

$$\frac{\begin{array}{c} c \vdash_{\mathrm{R}} a \Rrightarrow x \\ c \vdash_{\mathrm{Seq\,R}} as \Rrightarrow xs \end{array}}{c \vdash_{\mathrm{Seq\,R}} \langle a \rangle \frown as \Rrightarrow \langle x \rangle \frown xs} \qquad \boxed{\textit{Seq-i}}$$

An example of the use of this meta rule is in the definition of the elaboration relation for sequences of expressions into sequences of terms

$$\mathit{Sig} \vdash_{\mathrm{Seq\,Expr}} \mathrm{seq}\,\mathit{Exp} \Rrightarrow \mathrm{seq}\,\mathit{Term}$$

from the elaboration relation for expressions

$$\mathit{Sig} \vdash_{\mathrm{Expr}} \mathit{Exp} \Rrightarrow \mathit{Term}$$

8.2 Types

The principal semantic object corresponding to a type name is a type denotation. Type names are represented as type denotations in *ISig*'s. There is a type denotation for every LCL type.

Every LCL *object type* is additionally associated with an LSL sort. The semantic functions that relate types to sorts are presented in Section 8.2.8. In some cases, several types will correspond to the same sort (i.e. the mapping from types to sorts is many to one). Thus, type information is "lost" when sorts are used to represent types in LSL traits. To compensate for this loss of information, declared identifiers must sometimes be subject to constraints which we will call *type constraints.* Type constraints are expressed in the form of LSL predicates. For example, the following array variable declaration

```
int a[3];
```

will be represented (in LSL) as the operator

```
a: → Arr[int]
```

This operator will be subject to the constraint that its dimension be 3

```
dim(a) = 3
```

8.2.1 On the Form of the Elaboration Relations

In this section we define the elaboration relations

$$Sig \vdash_{\mathtt{UTpNm}_r} UTpNm \Rrightarrow (UTpDen \times LSLPred)$$
$$Sig \vdash_{\mathtt{TpNm}_r} TpNm \Rrightarrow (TpDen \times LSLPred)$$

that associate a type name to a type denotation and a type constraint. Why define relations that yield *both* a type denotation *and* a type constraint? Why not two sets of elaboration relations? Because a simpler definition results if we elaborate the type denotation and the constraint together. Furthermore, we generally need both the type denotation and the constraint at the same time. Also note that we cannot derive the type constraint from the type denotation—in particular because of `typedef` names with constraints.

Why not define elaboration relations that relate a type name to a triple that would also include the sort name? Because not all type names are associated with a sort name—only object types have an associated sort. (Although it is also the case that not all type names have type constraints, the absence of a type constraint is conveniently represented by the predicate `true`.) If a type name has an associated sort name, then the sort name can be derived from the type denotation using the functions given in Section 8.2.8.

8.2.2 An Anonymous Logical Variable

When formulating a type constraint we are faced with a dilemma: we do not know how to name the entity to which the constraint is being applied. For this purpose we will make use of an anonymous logical variable that has a name that is different from any user definable name

$$anonLVarNm : LVarNm$$

If d is a declarator to which a given type constraint *lslPred* is to be applied, then *anonSubst d lslPred* is the predicate that results from the substitution of all occurrences of *anonLVarNm* by d in *lslPred*. For example, if *lslPred* is

$\mathtt{dim(}\underline{anonLVarNm}\mathtt{)} = 3$

and *d* is 'a' then *anonSubst d lslPred* is

```
dim(a) = 3
```

$$
\begin{array}{|l}
\textit{anonSubst} : \textit{Term} \longrightarrow \textit{Term} \longrightarrow \textit{Term} \\
\hline
\textit{anonSubst tm tm}' \in \\
\quad \text{let } \textit{anonLVarNS} == (\textit{anonLVarNm}, \textit{tmSort tm}) \bullet \\
\quad \text{let } \textit{subst} == \{\textit{anonLVarNS} \mapsto \textit{tm}\} \bullet \\
\qquad \textit{appSubst subst tm}'
\end{array}
$$

If the (qualified or unqualified) type name *tn* elaborates to the (qualified or unqualified) type denotation *td* with the type constraint *lslPred*, then all occurrences (if any) of the logical variable name *anonLVarNm* in *lslPred* will be of the sort of *tn*. For example, the type name `int[3]` will elaborate to the type denotation `int[3]` and the type constraint

dim(<u>*anonLVarNm*</u>:Arr[int]) = 3

The sort of the anonymous logical variable, which is shown by explicit qualification, is the sort of `int[3]`—i.e. `Arr[int]`. The type `obj int[3]` will elaborate to the type denotation `obj int[3]` and the constraint

dim(val(any,<u>*anonLVarNm*</u>:Obj[Arr[int]])) = 3

The sort of the anonymous logical variable, `Obj[Arr[int]]`, is the sort of `obj int[3]`. As this last example illustrates, type constraints may also make use of the "anonymous" state variable `any` of sort `Store`.

If there is no sort associated with a given type, then there will be no occurrences of the anonymous logical variable in its type constraint since the type constraint will be `true`.

8.2.3 Fundamental Types

Regardless of the context, fundamental types elaborate to their corresponding denotations. There is no type constraint associated with a fundamental type (hence the type constraint is represented as the predicate `true`).

$$\frac{}{sig \vdash_{\mathtt{UTpNm_r}} \mathit{VoidUTN} \Rrightarrow (\mathit{VoidUTD}, [\![\mathtt{true}():\mathtt{Bool}]\!])} \quad \boxed{\text{VoidUTN}}$$

$$\frac{}{sig \vdash_{\mathtt{UTpNm_r}} \mathit{BasicUTN}\ \mathit{basicTp} \Rrightarrow (\mathit{BasicUTD}\ \mathit{basicTp}, [\![\mathtt{true}():\mathtt{Bool}]\!])} \quad \boxed{\text{BasicUTN}}$$

8.2.4 Array Types

We define two rules for array types: one rule concerns array types with unspecified dimensions and the other rule concerns array types for which the dimension is given.

8.2.4.1 Rule: ArrUTN0

The rule ArrUTN0 concerns the elaboration, under the LCL signature *sig*, of the array type

$$utn = \mathit{ArrUTN}(\mathit{elemTN}, \mathit{NoExp})$$

with the element type *elemTN* and an unspecified dimension.

8.2.4.1.1 Hypotheses The array element type must elaborate under *sig*,

$$sig \vdash_{\mathtt{TpNm_r}} \mathit{elemTN} \Rrightarrow (\mathit{elemTD}, \mathit{elemLSLPred})$$

and it must be an object type.

$$\mathit{elemTD}.utd \in \mathit{objUTD}$$

8.2.4.1.2 Conclusion The type denotation corresponding to the array type *utn* is

$$utd = \mathit{ArrUTD}(\mathit{elemTD}, \mathit{NoTerm})$$

The array type constraint, which we name *lslPred*, is induced by the element type constraint *elemLSLPred*: every array element must be subject to the element type constraint.

$$sort = utd2SN\ utd$$
$$anonLVarTm = LVarTm(anonLVarNm, sort)$$

The sort of the anonymous logical variable, *sort*, is the sort of the array type *utd*. The set of indices of the array elements is `inds(`$\underline{anonLVarTm}$`)`. The array type constraint *lslPred* is

$$\forall\ \texttt{i:int}\ (\texttt{i:int} \in \texttt{inds(}\underline{anonLVarTm}\texttt{):Set[int]} \Rightarrow \underline{lslPred'})$$

where

$$elemSort = td2SN\ elemTD$$
$$tm' = [\![\underline{anonLVarTm}[\texttt{i:int}]\texttt{:}\underline{elemSort}]\!]$$
$$lslPred' = anonSubst\ tm'\ elemLSLPred$$

Informally, *tm′* is the LSL term $\underline{anonLVarNm}$`[i]` which denotes the ith element of the array value *anonLVarNm*. If *elemLSLPred* is `true`, then *lslPred* simplifies to `true`.

8.2.4.1.3 Rule Summary

$$\frac{\begin{array}{l} sig \vdash_{\texttt{TpNm}_r} elemTN \Rrightarrow (elemTD, elemLSLPred) \\ elemTD.utd \in objUTD \end{array}}{sig \vdash_{\texttt{UTpNm}_r} utn \Rrightarrow (utd, lslPred)} \qquad \boxed{\text{ArrUTN0}}$$

where

$$utn = ArrUTN(elemTN, NoExp)$$
$$utd = ArrUTD(elemTD, NoTerm)$$
$$sort = utd2SN\ utd$$
$$anonLVarTm = LVarTm(anonLVarNm, sort)$$
$$elemSort = td2SN\ elemTD$$
$$tm' = [\![\underline{anonLVarTm}[\texttt{i:int}]\texttt{:}\underline{elemSort}]\!]$$
$$lslPred' = anonSubst\ tm'\ elemLSLPred$$

and *lslPred* is

$$\forall\ \texttt{i:int}\ (\texttt{i:int} \in \texttt{inds(}\underline{anonLVarTm}\texttt{):Set[int]} \Rightarrow \underline{lslPred'})$$

8.2.4.2 Rule: ArrUTN1

The rule ArrUTN1 concerns the elaboration, under the LCL signature *sig*, of the array type

$$utn = ArrUTN(elemTN, OneExp\ exp)$$

with element type *elemTN* and with a dimension specified by the expression *exp*.

8.2.4.2.1 Hypotheses The hypotheses for rule ArrUTN1 include those of rule ArrUTN0. Thus, the element type must elaborate (under the given signature) to an object type denotation.

$$sig \vdash_{\mathrm{TpNm_r}} elemTN \Rrightarrow (elemTD, elemLSLPred)$$
$$elemTD.utd \in objUTD$$

In addition, the expression specifying the array dimension must elaborate to an LSL term under *sig*,

$$(NoCtx, sig) \vdash_{\mathrm{Expr}} exp \Rrightarrow tm$$

and this term must be of sort `int`

$$tmSort\ tm = basicSN\ IntTp$$

8.2.4.2.2 Conclusion The type denotation corresponding to the array type *utn* is

$$utd = ArrUTD(elemTD, OneTerm\ tm)$$

As is the case for rule ArrUTD0, we must impose the element type constraint on all of the array members. Since we know what the array dimension is, we must additionally constrain the array to be of the declared dimension.

$$sort = utd2SN\ utd$$
$$anonLVarTm = LVarTm(anonLVarNm, sort)$$

As before *anonLVarTm* represents an anonymous declarator of type *utd* to which the type constraint is to be applied. The type constraint, *lslPred*, is

$\texttt{dim(}\underline{anonLVarTm}\texttt{)} = \underline{tm} \wedge$
$\forall\ \texttt{i:int}\ (\texttt{i:int} \in \texttt{inds(}\underline{anonLVarTm}\texttt{):Set[int]} \Rightarrow \underline{lslPred'}\texttt{)}$

where

$elemSort = td2SN\ elemTD$
$tm' = [\![\underline{anonLVarTm}[\texttt{i:int}]\texttt{:}\underline{elemSort}]\!]$
$lslPred' = anonSubst\ tm'\ elemLSLPred$

If *elemLSLPred* is `true`, then *lslPred* simplifies to

$\texttt{dim(}\underline{anonLVarTm}\texttt{)} = \underline{tm}$

8.2.4.2.3 Rule Summary

$$\frac{\begin{array}{l} sig \vdash_{\mathbf{TpNm_r}} elemTN \Rrightarrow (elemTD, elemLSLPred) \\ elemTD.utd \in objUTD \\ (NoCtx, sig) \vdash_{\mathbf{Expr}} exp \Rrightarrow tm \\ tmSort\ tm = basicSN\ IntTp \end{array}}{sig \vdash_{\mathbf{UTpNm_r}} utn \Rrightarrow (utd, lslPred)} \quad \boxed{\text{ArrUTN!}}$$

where *lslPred* is

$\texttt{dim(}\underline{anonLVarTm}\texttt{)} = \underline{tm} \wedge$
$\forall\ \texttt{i:int}\ (\texttt{i:int} \in \texttt{inds(}\underline{anonLVarTm}\texttt{):Set[int]} \Rightarrow \underline{lslPred'}\texttt{)}$

and

$utn = ArrUTN(elemTN, OneExp\ exp)$
$utd = ArrUTD(elemTD, OneTerm\ tm)$
$sort = utd2SN\ utd$
$anonLVarTm = LVarTm(anonLVarNm, sort)$
$elemSort = td2SN\ elemTD$
$tm' = [\![\underline{anonLVarTm}[\texttt{i:int}]\texttt{:}\underline{elemSort}]\!]$
$lslPred' = anonSubst\ tm'\ elemLSLPred$

8.2.5 Function Types

We describe the elaboration of the function type

$$utn = FunUTN\ \theta FunUTNSch$$

under the signature *sig*.

8.2.5.1 Hypotheses

No two parameters can have the same name[1].

$$prmIds \in \mathrm{iseq}\, Id$$

The return, parameter and global variable types must elaborate under the given signature.

$$\begin{array}{l} sig \vdash_{\mathsf{UTpNm_r}} retUTN \Rrightarrow (retUTD, retPred) \\ sig \vdash_{\mathsf{Seq\,TpNm_r}} prmTNs \Rrightarrow prmTDPreds \\ prmTDs = first \circ prmTDPreds \\ sig \vdash_{\mathsf{Seq\,UTpNm_r}} gvarUTNs \Rrightarrow gvarUTNPreds \\ gvarUTDs = first \circ gvarUTNPreds \end{array}$$

Each parameter must have a type that is either an object, array or function type [ISO, §6.7.1].

$$\begin{array}{l} (\forall\, i : \mathrm{dom}\, prmTDs \bullet \\ \quad (prmTDs\ i).utd \in objUTD\ \cup \\ \qquad \mathrm{ran}\, ArrUTD\ \cup \\ \qquad \mathrm{ran}\, FunUTD) \end{array}$$

The return type must either be **void** or an LCL object type that is not an array type [ISO, §6.3.2.2, §6.5.3, §6.7.1].

$$\begin{array}{l} retUTD \in \{ VoidUTD \}\ \cup \\ \qquad objUTD \setminus (\mathrm{ran}\, ArrUTD) \end{array}$$

[1] $\mathrm{iseq}\, X$ is the set of all injective sequences of X's: each member of such a sequence has a unique value.

Each identifier in the global variable list must actually be declared in the global signature.

$$
\begin{array}{l}
\mathrm{ran}(\mathit{ordId} \circ \mathit{gvarIds}) \subseteq \mathrm{dom}\,\mathit{sig.isig.b} \\
(\mathbf{let}\ \mathit{attrs} == \mathit{sig.isig.b} \circ \mathit{ordId} \circ \mathit{gvarIds} \bullet \\
\quad (\forall\, i : \mathrm{dom}\,\mathit{attrs} \bullet \\
\qquad (\mathit{attrs}\ i).\mathit{idKind} = \mathit{VarIdK} \land \\
\qquad (\mathit{attrs}\ i).\mathit{td.utd} = \mathit{gvarUTDs}\ i))
\end{array}
$$

8.2.5.2 Conclusion

The result of the elaboration is the pair

$$\mathit{utd\_LSLPred} = (\mathit{FunUTD}\,\theta \mathit{FunUTDSch}, [\![\texttt{true():Bool}]\!])$$

Since a function type is not an object type, there is no type constraint.

8.2.5.3 Rule Summary

$$
\dfrac{
\begin{array}{l}
\mathit{prmIds} \in \mathrm{iseq}\,\mathit{Id} \\
\mathit{sig} \vdash_{\mathrm{UTpNm_r}} \mathit{retUTN} \Rrightarrow (\mathit{retUTD}, \mathit{retPred}) \\
\mathit{sig} \vdash_{\mathrm{Seq\,TpNm_r}} \mathit{prmTNs} \Rrightarrow \mathit{prmTDPreds} \\
\mathit{prmTDs} = \mathit{first} \circ \mathit{prmTDPreds} \\
\mathit{sig} \vdash_{\mathrm{Seq\,UTpNm_r}} \mathit{gvarUTNs} \Rrightarrow \mathit{gvarUTNPreds} \\
\mathit{gvarUTDs} = \mathit{first} \circ \mathit{gvarUTNPreds} \\
(\forall\, i : \mathrm{dom}\,\mathit{prmTDs} \bullet \\
\quad (\mathit{prmTDs}\ i).\mathit{utd} \in \mathit{objUTD} \cup \\
\qquad \mathrm{ran}\,\mathit{ArrUTD} \cup \\
\qquad \mathrm{ran}\,\mathit{FunUTD}) \\
\mathit{retUTD} \in \{\mathit{VoidUTD}\} \cup \\
\qquad \mathit{objUTD} \setminus (\mathrm{ran}\,\mathit{ArrUTD}) \\
\mathrm{ran}(\mathit{ordId} \circ \mathit{gvarIds}) \subseteq \mathrm{dom}\,\mathit{sig.isig.b} \\
(\mathbf{let}\ \mathit{attrs} == \mathit{sig.isig.b} \circ \mathit{ordId} \circ \mathit{gvarIds} \bullet \\
\quad (\forall\, i : \mathrm{dom}\,\mathit{attrs} \bullet \\
\qquad (\mathit{attrs}\ i).\mathit{idKind} = \mathit{VarIdK} \land \\
\qquad (\mathit{attrs}\ i).\mathit{td.utd} = \mathit{gvarUTDs}\ i))
\end{array}
}{
\mathit{sig} \vdash_{\mathrm{UTpNm_r}} \mathit{utn} \Rrightarrow \mathit{utd\_LSLPred}
}
\quad \boxed{\text{FunUTN}}
$$

where

$$utn = \mathit{FunUTN}\ \theta \mathit{FunUTNSch}$$
$$utd\_LSLPred = (\mathit{FunUTD}\ \theta \mathit{FunUTDSch}, [\![\texttt{true():Bool}]\!])$$

8.2.6 Abstract Types

The identifier used to name an abstract type must also be a valid sort name that does not correspond to a sort name of an exposed LCL type, i.e., the identifier must be what we have called a user sort name. There is no type constraint associated with an abstract type.

$$\frac{id \in userSN}{sig \vdash_{\mathrm{UTpNm_r}} \mathit{ImmUTN}\ id \Rrightarrow (\mathit{ImmUTD}\ id, [\![\texttt{true():Bool}]\!])} \qquad \boxed{\text{ImmUTN}}$$

8.2.7 (Qualified) Types

A qualified type consists of an unqualified type and a collection of qualifiers. We define two rules: one deals with qualified type names without qualifiers and the other deals with obj qualified type names[2].

In both of the rules that we present below, the type denotation of the qualified type consists of the type denotation of its underlying unqualified type along with the same collection of qualifiers (if any). The rules differ in the definition of the type constraint.

8.2.7.1 Rule: TpNm0

We begin by defining the elaboration of qualified type names without qualifiers. Elaboration is of the qualified type name

$$tn = \theta \mathit{TpNm}$$

under the LCL signature *sig*.

[2]In the current version of the semantics we do not provide the meaning of **const** and **volatile** qualified types.

8.2.7.1.1 Hypotheses The underlying unqualified type of *tn* must elaborate under *sig*

$$sig \vdash_{\text{UTpNm}_r} utn \Rrightarrow (utd, lslPred)$$

and *tn* must not have any qualifiers

$$tpQuals = \varnothing$$

8.2.7.1.2 Conclusion The type name elaborates to

$$(\theta TpDen, lslPred)$$

consisting of the type denotation of *utn* and the same qualifiers as *tn*. Since *utd* and *θTpDen* have the same sort, we simply "pass on" the type constraint of *utn*.

8.2.7.1.3 Rule Summary

$$\frac{sig \vdash_{\text{UTpNm}_r} utn \Rrightarrow (utd, lslPred) \quad tpQuals = \varnothing}{sig \vdash_{\text{TpNm}_r} \theta TpNm \Rrightarrow (\theta TpDen, lslPred)} \qquad \boxed{\text{TpNm0}}$$

8.2.7.2 Rule: TpNm-ObjQual

We define the elaboration of the `obj` qualified type name

$$tn = \theta TpNm$$

under the LCL signature *sig*.

8.2.7.2.1 Hypotheses The underlying unqualified type of *tn* must elaborate under *sig*

$$sig \vdash_{\text{UTpNm}_r} utn \Rrightarrow (utd, lslPred)$$

The only qualifier in *tn* is `obj`

$$tpQuals = \{ObjQual\}$$

Since this is an `obj` qualified type, the underlying unqualified type must be an object type or an incomplete object type

$$utd \in objUTD \cup incObjUTD$$

8.2.7.2.2 Conclusion Elaboration yields

$$td\_LSLPred = (\theta TpDen, lslPred')$$

The type denotation $\theta TpDen$ is the `obj` qualified type denotation of *utn*. The type constraint *lslPred'* is described next.

The type constraint of *utn* is expressed as a predicate over an anonymous declarator of the sort of *utn*

$$sort = utd2SN\ utd$$

A declarator of type *tn* will represent an object containing values of sort *sort*. In any state, the values contained in this object must satisfy the type constraint of *utn*. Thus, the type constraint of *tn*, *lslPred'*, is simply

$$\begin{aligned}&lslPred' = anonSubst\ tm\ lslPred\\&sort' = td2SN\ \theta TpDen = objSN\ sort\\&anonLVarTm = LVarTm(anonLVarNm, sort')\\&tm = [\![\mathtt{val}(\mathrm{any}{:}\mathtt{Store}, \underline{anonLVarTm}){:}\underline{sort}]\!]\end{aligned}$$

where *anonLVarTm* represents the anonymous declarator denoting an object containing values of sort *sort*. *tm* is the LSL term representing the value contained in this object in an arbitrary state any.

8.2.7.2.3 Rule Summary

$$\frac{\begin{array}{l}sig \vdash_{\mathrm{UTpNm}_r} utn \Rrightarrow (utd, lslPred)\\ tpQuals = \{ObjQual\}\\ utd \in objUTD \cup incObjUTD\end{array}}{sig \vdash_{\mathrm{TpNm}_r} tn \Rrightarrow td\_LSLPred} \qquad \boxed{\text{TpNm-ObjQual}}$$

where

$$\begin{aligned}&tn = \theta TpNm\\&td\_LSLPred = (\theta TpDen, lslPred')\\&sort = utd2SN\ utd\\&lslPred' = anonSubst\ tm\ lslPred\\&sort' = td2SN\ \theta TpDen = objSN\ sort\\&anonLVarTm = LVarTm(anonLVarNm, sort')\\&tm = [\![\mathtt{val}(\mathrm{any}{:}\mathtt{Store}, \underline{anonLVarTm}){:}\underline{sort}]\!]\end{aligned}$$

8.2.8 Types to Sorts

The partial functions that we define next map object and array type denotations into the sort names to which they correspond. Why not define a function from type names into sort names? Because the same type name can correspond to different types and hence to different sorts; this can be the case for structure or union types.

The function *utd2SN* maps an unqualified type denotation into its sort[3]. The object sort of a type *utd* can be obtained by an application of *objSN* to the sort of *utd*.

$$
\begin{array}{|l}
utd2SN : UTpDen \mathrel{\mapstochar\rightarrow} SortNm \\
\hline
\operatorname{dom} utd2SN = objUTD \cup \operatorname{ran} ArrUTD \\[6pt]
utd2SN(BasicUTD\ t) = basicSN\ t \\
utd2SN(ArrUTD(td, optTm)) = arrSN(td2SN\ td) \\
utd2SN(ImmUTD\ sort) = sort
\end{array}
$$

The sort associated with a type denotation *td* will be the same as the sort of its component unqualified type denotation (*td.utd*) unless the `obj` qualifier is present. If the `obj` qualifier is present, then the sort associated with *td* will be the object sort of *td.utd*.

$$
\begin{array}{|l}
td2SN : TpDen \mathrel{\mapstochar\rightarrow} SortNm \\
\hline
td2SN = (\lambda\ TpDen \mid utd \in \operatorname{dom} utd2SN \bullet \\
\qquad \mathbf{let}\ sort == utd2SN\ utd \bullet \\
\qquad\qquad \mathbf{if}\ ObjQual \in tpQuals \\
\qquad\qquad\qquad \mathbf{then}\ objSN\ sort \\
\qquad\qquad\qquad \mathbf{else}\ sort)
\end{array}
$$

8.2.9 Trait References for Sorts of LCL Exposed Types

In the semantic definition, we provide traits that define the meanings of the sorts on which LCL exposed types are based. The use of an LCL exposed type in a specification usually requires the inclusion of one of these traits. The function *sort2Ref* maps each

[3] *utd2SN* yields what Tan [Tan94] calls the *value sort* of the type. We call *utd2SN utd* the sort *associated with utd* or, more simply, *the* sort of *utd*.

nonuser sort into a trait reference which, if included, will define the theory associated with that sort.

$$\begin{array}{|l} sort2Ref : SortNm \rightarrowtail\!\!\!\!\!\!\!\rightarrow TraitRef \\ \hline \mathrm{dom}\ sort2Ref = SortNm \setminus userSN \end{array}$$

The `int` sort is defined by the `IntTp` trait.

$$sort2Ref(basicSN\ IntTp) = traitNm2Ref\ \mathtt{IntTp}$$

An array sort $sort = \mathtt{Arr[S]}$ with element sort `S`, is defined by trait `ArrTp(S for E)`.

$$\begin{array}{l} sort2Ref(arrSN\ eltSN) = \\ \quad (\mu\ TraitRef \mid \\ \quad\quad traitNm = \mathtt{ArrTp} \land \\ \quad\quad ren = (\mu\ Ren \mid \\ \quad\quad\quad sortPRen = \{\mathtt{E} \mapsto eltSN\} \land \\ \quad\quad\quad lvarPRen = \varnothing \land \\ \quad\quad\quad opPRen = \varnothing)) \end{array}$$

The same trait reference is used to define `Obj[Arr[S]]`.

$$sort2Ref(objSN(arrSN\ eltSN)) = sort2Ref(arrSN\ eltSN)$$

The object sort `Obj[S']` of any non-array sort `S'` is defined by the trait reference `SProjStore(S' for S)`.

$$\begin{array}{l} sort \notin \mathrm{ran}\ arrSN \Rightarrow sort2Ref(objSN\ sort) = \\ \quad (\mu\ TraitRef \mid \\ \quad\quad traitNm = \mathtt{SProjStore} \land \\ \quad\quad ren = (\mu\ Ren \mid \\ \quad\quad\quad sortPRen = \{\mathtt{S} \mapsto sort\} \land \\ \quad\quad\quad lvarPRen = \varnothing \land \\ \quad\quad\quad opPRen = \varnothing)) \end{array}$$

8.3 Expressions

We define the elaboration of LCL expressions into LSL terms under a context identifier and an LCL signature.

$$(ExpCtx \times Sig) \vdash_{\text{Exp}_r} Exp \Rrightarrow Term$$

$$ExpCtx ::= NoCtx \mid GenericCtx \mid PreCtx \mid PostCtx$$

Some LCL operators implicitly refer to the generic, pre-, or post-states. Hence we must restrict the use of these operators to those contexts in which it is sensible to refer to any one of these states. Details are given in Section 8.3.2.

8.3.1 Logical Variables

The elaboration of an expression consisting of a logical variable is the term denoting that logical variable provided the logical variable is in the vocabulary of the LSL signature component of the LCL signature. The *ExpCtx* is irrelevant.

$$\frac{tm = LVarTm\ lvarNS \quad tm \in wfTm\ lsig}{(expCtx, \theta Sig) \vdash_{\text{Exp}_r} LVarExp\ lvarNS \Rrightarrow tm} \qquad \boxed{\text{LVarExp}}$$

8.3.2 Operators

The elaboration of an expression that is an application of an operator depends on whether the operator is an LCL or an LSL operator. The LCL operators are given in the first column of Table 4. This collection of operators is named *LCLOpNSs*. The rule AppExp-LSLOp defines the elaboration of LSL operators.

$$\frac{\begin{array}{l} opNmSig \notin LCLOpNSs \\ expCtx, \theta Sig \vdash_{\text{Seq Exp}_r} exps \Rrightarrow tms \\ tm = AppTm(opNmSig, tms) \quad tm \in wfTm\ lsig \end{array}}{expCtx, \theta Sig \vdash_{\text{Exp}_r} AppExp(opNmSig, exps) \Rrightarrow tm} \qquad \boxed{\text{AppExp-LSLOp}}$$

An expression consisting of the application of an LCL operator is translated into an LSL term according to Table 4: if *opNmSig* is an operator that occurs in the first

LCL Operator (*OpNmSig*)	Contexts	LSL Term Schema
depOn:Obj[S_1],Obj[S_2] $\rightarrow$ Bool	all	depOn(__,__)
indep:Obj[S_1],..., Obj[S_n] $\rightarrow$ Bool	all	indep(empty $\vdash$ __ $\vdash$ ... __)
__\pre, __^: Obj[S] $\rightarrow$ S	pre	val(pre,__)
__\\activePre: Obj[S] $\rightarrow$ S	pre	__ $\in$ activeObjs(pre)
__\\wellDefPre: Obj[S] $\rightarrow$ S	pre	__ $\in$ wellDefObjs(pre)
__\post, __': Obj[S] $\rightarrow$ S	post	val(post,__)
__\\activePost: Obj[S] $\rightarrow$ S	post	__ $\in$ activeObjs(post)
__\\wellDefPost: Obj[S] $\rightarrow$ S	post	__ $\in$ wellDefObjs(post)
fresh: Obj[S] $\rightarrow$ Bool	post	$\forall$ x:Obj ((x $\in$ activeObjs(pre) $\wedge$ x $\in$ activeObjs(post)) $\Rightarrow$ indep(empty $\vdash$ x $\vdash$__))
trashed: Obj[S] $\rightarrow$ Bool	post	__ $\notin$ activeObjs(post)
__\any, __$^\bullet$: Obj[S] $\rightarrow$ S	generic	val(any,__)
__\\activeAny: Obj[S] $\rightarrow$ S	generic	__ $\in$ activeObjs(any)
__\\wellDefAny: Obj[S] $\rightarrow$ S	generic	__ $\in$ wellDefObjs(any)

Table 4: LCL Operator Translations

column of the table, then *LCLOpCtxs opNmSig* is the set of *ExpCtx*'s (from the second column in the same row) in which it can appear and *LCLOpTrans*(*opNmSig*, *tms*) denotes the term obtained from the third column (of the same row) by replacing the occurrences of '__' by the terms denoting the arguments of the operator. The left-to-right order of the arguments is preserved.

$$\frac{\begin{array}{l} opNmSig \in LCLOpNSs \quad expCtx \in LCLOpCtxs\ opNmSig \\ expCtx, \theta Sig \vdash_{\mathrm{SeqExpr}} exps \Rrightarrow tms \\ tm = LCLOpTrans(opNmSig, tms) \quad tm \in wfTm\ lsig \end{array}}{expCtx, \theta Sig \vdash_{\mathrm{Expr}} AppExp(opNmSig, exps) \Rrightarrow tm} \quad \boxed{\text{AppExp-LCLOp}}$$

8.4 Declarations

We now define the elaboration relation for declarations

$$Env \vdash_{\mathrm{Dcl}_r} Dcl \Rrightarrow Env$$

It relates an "global" environment and a declaration to an environment *increment*, that is an environment that contains *only* the given declaration (as opposed to the global environment enriched with the declaration). Defining incremental, rather than cumulative, elaboration rules make it easier to provide the semantics of local declarations.

A declaration is a kind of specification component; recall from Section 5.5

$$\begin{aligned} &Cpt ::= DclCpt\langle\!\langle DclQual \times Dcl\rangle\!\rangle \\ &\qquad\vdots \end{aligned}$$

A declaration specification component (*DclCpt*) consists of a declaration qualifier and a declaration. The declaration qualifier identifies whether the declaration is a **spec** component or not. One of the attributes of a declarator that is stored in an environment is the value of the declaration qualifier. Notice that a declaration (*Dcl*) does not contain the value of the declaration qualifier. Consequently, we elaborate declarations under the assumption that they are **spec** declarations. Elaboration of declaration specification components is covered in Section 8.5.1.

8.4.1 Constants

The elaboration of the constant declaration

$$dcl = ConDcl\,\theta UDcl$$

under the LCL environment θEnv is defined next.

8.4.1.1 Hypotheses

Since LCL constants are classified as ordinary identifiers, there must not already exist an ordinary identifier with the name *id*.

$$\begin{aligned} &ip = ordId\; id \\ &ip \notin \mathrm{dom}\, isig.b \end{aligned}$$

The unqualified type of the constant declaration must elaborate under the signature of θEnv and the resulting type denotation must be an object type.

$$\begin{aligned} &\theta Sig \vdash_{\mathrm{UTpNm_r}} utn \Rrightarrow (utd, lslPred) \\ &utd \in objUTD \end{aligned}$$

8.4.1.2 Conclusion

The result of the elaboration is an environment increment containing only the declaration *dcl*.

$$env = mkEnv(ip\_and\_Attr, tb)$$

An environment is uniquely determined by its *ISig* and trait body components: these are defined next.

8.4.1.2.1 *ISig* When defining an *ISig* increment, we need only define the value of the binding in the *ISig*. The *ISig* increment contains a single binding for the declared identifier

$$ip\_and\_Attr = ip \mapsto \theta Attr$$

with the following attributes

$$dclQual = SpecQual$$
$$idKind = ConIdK$$
$$td = utd2TD\ utd$$

That is, *ip* is a spec LCL constant of type *utd*.

8.4.1.2.2 Trait Body For the trait body we must identify the included traits, the declared operations and the assertions. Let *sort* be the sort associated with the type of the declared constant. For each sort component of *sort* that is the sort of an LCL exposed type, we must include the trait that defines it. For example, given the sort `Arr[int]` we must include the traits defining `Arr[int]` and `int`. The set of all trait references is *refs*.

$$sort = utd2SN\ utd$$
$$cptSorts = cptSN^{*}(\!|\{sort\}|\!)$$
$$refs = sort2Ref(\!|cptSorts|\!)$$

The operator introduced is a constant operator named *id* of sort *sort*

$$\underline{id}: \rightarrow \underline{sort}$$

Formally, the declared operator is *opNmSig*

$$opNmSig = id \mapsto mkConOpSig\ sort$$

The only assertion to be made, which we will call *lslPred′* is mandated by the type constraint: we assert that the type constraint *lslPred* applies to the operator representing the declared constant. When used to form a term, the constant operator is applied to the empty argument list.

$$tm = AppTm(opNmSig, \langle\rangle)$$
$$lslPred' = anonSubst\ tm\ lslPred$$

Given that *lvarNSs* is the set of logical variables that appear in *lslPred′*, then the trait body *tb* is defined as follows.

$$lvarNSs = tmLVarNSs\ lslPred'$$
$$tb \in mkTB(refs, lvarNSs, \{opNmSig\}, \{lslPred'\})$$

8.4.1.3 Rule Summary

$$\frac{\begin{array}{l} ip \notin \mathrm{dom}\ isig.b \\ \theta Sig \vdash_{\mathrm{UTpNm_r}} utn \Rrightarrow (utd, lslPred) \\ utd \in objUTD \end{array}}{\theta Env \vdash_{\mathrm{Dcl_r}} dcl \Rrightarrow env} \qquad \boxed{\text{ConDcl}}$$

where

$$\begin{aligned}
&dcl = ConDcl\,\theta UDcl\\
&ip = ordId\; id\\
&env = mkEnv(ip\_and\_Attr, tb)\\
&ip\_and\_Attr = ip \mapsto \theta Attr\\
&dclQual = SpecQual\\
&idKind = ConIdK\\
&td = utd2TD\; utd\\
&sort = utd2SN\; utd\\
&cptSorts = cptSN^{*}(\!|\{sort\}|\!)\\
&refs = sort2Ref(\!|cptSorts|\!)\\
&opNmSig = id \mapsto mkConOpSig\; sort\\
&tm = AppTm(opNmSig, \langle\rangle)\\
&lslPred' = anonSubst\; tm\; lslPred\\
&lvarNSs = tmLVarNSs\; lslPred'\\
&tb \in mkTB(refs, lvarNSs, \{opNmSig\}, \{lslPred'\})
\end{aligned}$$

8.4.2 Variables

The elaboration of the variable declaration

$$dcl = VarDcl\,\theta QDcl$$

under the LCL environment θEnv is defined next.

8.4.2.1 Hypotheses

Identifiers denoting variables are part of the name space of ordinary identifiers. Thus, the given environment must not already contain a declaration of an ordinary identifier with the name *id*.

$$\begin{aligned}
&ip = ordId\; id\\
&ip \notin \mathrm{dom}\; isig.b
\end{aligned}$$

The type of the declared variable must elaborate under the signature of θEnv and the resulting type denotation must be an object type or an incomplete object type.

$$\theta Sig \vdash_{\mathrm{TpNm}_r} tn \Rrightarrow (td, lslPred)$$
$$td.utd \in objUTD \cup incObjUTD$$

Furthermore, in the current release of the semantics, we only define the elaboration of unqualified variable declarations.

$$td.tpQuals = \varnothing$$

8.4.2.2 Conclusion

The result of the elaboration is an environment increment containing only the declaration *dcl*.

$$env = mkEnv(ip\_and\_Attr, tb)$$

An environment is uniquely determined by its *ISig* and trait body components.

8.4.2.2.1 *ISig* When defining an *ISig* increment, we need only provide the value of the binding in the *ISig*. The *ISig* increment contains a single binding for the declared identifier

$$ip\_and\_Attr = ip \mapsto \theta Attr$$

with the following attributes

$$dclQual = SpecQual$$
$$idKind = VarIdK$$

That is, *ip* is a spec variable of type *td*.

8.4.2.2.2 Trait Body For the trait body we must identify the included traits, the declared operations and the assertions. The given variable will be bound to an object containing values of the sort associated with *td*. Thus, *sort*, the sort of the variable, will be the object sort of *td*.

$$sort = objSN(td2SN\ td)$$

For each sort component of *sort* that is the sort of an LCL exposed type, we must include the trait that defines it. The set of all trait references is *refs*.

$$cptSorts = cptSN^{*}(\!| \{sort\} |\!)$$
$$refs = sort2Ref(\!| cptSorts |\!)$$

The operator introduced is a constant operator named *id* of sort *sort*

$$\underline{id}\colon \rightarrow \underline{sort}$$

Formally, the declared operator is *opNmSig*. When represented in a term, the operator is applied to the empty argument list.

$$opNmSig = id \mapsto mkConOpSig\ sort$$
$$tm = AppTm(opNmSig, \langle\rangle)$$

We must make the following three assertions.

- The type constraint *lslPred* applies to the operator representing the declared variable.

  $$lslPred' = anonSubst\ tm\ lslPred$$

- The object to which the variable is bound is active an all program states. This assertion, *lslPred''*, is

  $$\underline{tm} \in \texttt{activeObjs(any:Store):Set[}\underline{sort}\texttt{]}$$

- The object bound to the variable is independent of all objects bound to previously declared variables. This assertion, named *lslPred'''*, is defined below.

The set of all previously declared variables, represented as LSL operators, is given by *varsAsOpNSs*. The same set of variables represented as the corresponding set of LSL terms is given by *varsAsTms*.

$$\begin{aligned} varsAsOpNSs = \{\ & ip' : IdNmSp;\ Attr' \mid \\ & ip' \in \mathrm{dom}\, isig.b \land \\ & isig.b\ ip' = \theta Attr' \land idKind' = VarIdK \bullet \\ & \quad ip'.id \mapsto mkConOpSig(objSN(td2SN\ td'))\ \} \end{aligned}$$

$$\begin{aligned} varsAsTms = \{\ & opNmSig : varsAsOpNSs \bullet \\ & AppTm(opNmSig, \langle\rangle)\ \} \end{aligned}$$

lslPred‴ is

$\texttt{indepObjs(empty:Seq[Obj]} \vdash \underline{tm} \vdash \underline{tm_1} \vdash \ldots \vdash \underline{tm_n}\texttt{):Bool}$

where each member of the sequence tm_1, tm_2, ..., tm_n is a unique member of *varsAsTms* and the sequence includes all terms in *varsAsTms*. Finally, the trait body is defined as follows.

$$\begin{aligned}
&asns = \{lslPred', lslPred'', lslPred'''\}\\
&lvarNSs = \bigcup tmLVarNSs(\!|asns|\!)\\
&tb \in mkTB(refs, lvarNSs, \{opNmSig\}, asns)
\end{aligned}$$

8.4.2.3 Rule Summary

$$\frac{\begin{array}{l} ip \notin \mathrm{dom}\, isig.b \\ \theta Sig \vdash_{\mathsf{TpNm_r}} tn \Rrightarrow (td, lslPred) \\ td.utd \in objUTD \cup incObjUTD \\ td.tpQuals = \varnothing \end{array}}{\theta Env \vdash_{\mathsf{Dcl_r}} dcl \Rrightarrow env} \quad \boxed{\text{VarDcl}}$$

where *lslPred″* is

$\underline{tm} \in \texttt{activeObjs(any:Store):Set[}\underline{sort}\texttt{]}$

lslPred‴ is

$\texttt{indepObjs(empty:Seq[Obj]} \vdash \underline{tm} \vdash \underline{tm_1} \vdash \ldots \vdash \underline{tm_n}\texttt{):Bool}$

and finally,

$$
\begin{aligned}
&dcl = VarDcl\,\theta QDcl\\
&ip = ordId\ id\\
&env = mkEnv(ip\_and\_Attr, tb)\\
&ip\_and\_Attr = ip \mapsto \theta Attr\\
&dclQual = SpecQual\\
&idKind = VarIdK\\
&sort = objSN(td2SN\ td)\\
&cptSorts = cptSN^{*}(\!|\{sort\}|\!)\\
&refs = sort2Ref(\!|cptSorts|\!)\\
&opNmSig = id \mapsto mkConOpSig\ sort\\
&tm = AppTm(opNmSig, \langle\rangle)\\
&lslPred' = anonSubst\ tm\ lslPred\\
&varsAsOpNSs = \{\ ip' : IdNmSp;\ Attr' \mid\\
&\qquad ip' \in \mathrm{dom}\ isig.b\ \wedge\\
&\qquad isig.b\ ip' = \theta Attr' \wedge idKind' = VarIdK\ \bullet\\
&\qquad\qquad ip'.id \mapsto mkConOpSig(objSN(td2SN\ td'))\ \}\\
&varsAsTms = \{\ opNmSig : varsAsOpNSs\ \bullet\\
&\qquad AppTm(opNmSig, \langle\rangle)\ \}\\
&asns = \{lslPred', lslPred'', lslPred'''\}\\
&lvarNSs = \bigcup tmLVarNSs(\!|asns|\!)\\
&tb \in mkTB(refs, lvarNSs, \{opNmSig\}, asns)
\end{aligned}
$$

8.5 Specification Components

The elaboration relation for specification components

$$Env \vdash_{\mathsf{Cpt_r}} Cpt \Rrightarrow Env$$

relates a global environment and a component to an environment increment. In the current version of the semantic definition, a specification component is either a declaration or a function specification. Declaration components are described next. Function specifications are covered in Section 8.6.

8.5.1 Declaration Components

If the declaration *dcl* elaborates to the environment increment *env'* under *env*, then so does the spec qualified declaration component consisting of *dcl*.

$$\frac{env \vdash_{\mathsf{Dcl_r}} dcl \Rrightarrow env'}{env \vdash_{\mathsf{Cpt_r}} DclCpt(SpecQual, dcl) \Rrightarrow env'} \qquad \boxed{\text{SpecDcl}}$$

This is because the elaboration rules for declarations assume that the declarations are spec qualified.

On the other hand, if the declaration component is not qualified with spec, then we must change its status from spec to non-spec.

$$\frac{\begin{array}{c} env \vdash_{\mathsf{Dcl_r}} dcl \Rrightarrow env'; \\ \{ip\} = \mathrm{dom}\, env'.isig.b \end{array}}{env \vdash_{\mathsf{Cpt_r}} DclCpt(NotSpecQ, dcl) \Rrightarrow envExport\ env'\ ip} \qquad \boxed{\text{NonSpecDcl}}$$

ip is the identifier being declared; it will be the only identifier in the environment increment *env'*. In the current version of the semantic framework, we do not provide a semantics for type declarations (abstract types or typedef's). Hence all types are non-spec. When type declarations are covered, a hypothesis will have to be added to the NonSpecDcl rule that will ensure that a non-spec component does not contain uses of a spec type.

8.6 Function Specifications

We define the elaboration of the function specification

$FSCpt(\theta FSHeader, \theta FSBody)$

under the environment θEnv. The definition of the elaboration of function specifications is quite involved. Consequently, it is not practical to present all of the elaboration rule hypotheses in isolation from the conclusion.

8.6.1 Selected Hypotheses

Since function identifiers are part of the name space of ordinary identifiers, we must ensure that there is no ordinary identifier in θEnv that has already been declared with the identifier *id*.

$$ip = ordId\ id$$
$$ip \notin \mathrm{dom}\ isig.b$$

The unqualified type name *utn* must be a function type. It is not sufficient to require that the type denotation of *utn* be a function type since *utn* could be a `typedef` name—in which case the function type category would be 'inherited' from the `typedef`. This is not permitted in C [ISO, §6.7.1].

$$utn = FunUTN\ \theta FunUTNSch$$

The function type *utn* must elaborate under the global signature. Function types do not contribute type constraints, hence the type constraint is `true`.

$$\theta Sig \vdash_{\mathrm{UTpNm}_f} utn \Rrightarrow (utd, [\![\mathtt{true}():\mathtt{Bool}]\!])$$
$$utd = FunUTD\ \theta FunUTDSch$$

In the current version of the formal definition of LCL we do not provide a semantics for qualified parameters. Therefore, all parameters must be unqualified.

$$\forall\, i : \mathrm{dom}\ prmIds \bullet (prmTNs\ i).tpQuals = \varnothing$$

8.6.2 Conclusion and Remaining Hypotheses

Elaboration yields the environment increment $\theta Env'$ which is uniquely defined by its $isig'$ and tb' components; these components are defined next.

8.6.3 *ISig*

Uses: *ip* (p. 155), *utd* (p. 155).

Defines: $isig'$.

The function specification declares *id* to be a (non-`spec`) function with type *utd*.

$$dclQual = NotSpecQ$$
$$idKind = FunIdK$$
$$td = utd2TD\ utd$$

Thus, the *ISig* increment will contain this single declaration with the mentioned attributes

$$isig' = mkISig(ip, \theta Attr)$$

8.6.4 Trait Body

Uses: *id* (p. 154), *opSig* (p. 157), *opDef* (p. 157).

Defines: *tb′*.

The meaning of the function specification is captured in the form of an LSL trait body containing a single operator declaration and an assertion defining the meaning of this operator. It is also necessary to include the traits that define the theories over the sorts of the LCL exposed types that are used in the function specification. The necessary traits are derived from the LSL predicate used to define the operator. We also obtain from this predicate the names and sorts of the logical variables that must be declared in the trait.

$$\begin{aligned} tb' \in (\textbf{let}\ & refs == sort2Ref(\!| tmSorts\ opDef |\!) \cup \\ & \qquad \{traitNm2Ref\ \texttt{LCLAux}\}; \\ & lvarNSs == tmLVarNSs\ opDef; \\ & opNmSigs == \{id \mapsto opSig\}; \\ & lslPreds == \{opDef\} \bullet \\ & mkTB(refs, lvarNSs, opNmSigs, lslPreds)) \end{aligned}$$

The trait `LCLAux` provides auxiliary operators needed to define the meaning of a function specification. It is defined in Appendix D.

8.6.4.1 Function Specification Operator Signature

Uses: *retUTD* (p. 155), *effRetSort* (p. 170), *effPrmSorts* (p. 168).

Defines: *opSig*.

If the function return type is not `void`, then the declared operator has the signature

$\underline{S_1}$, ..., $\underline{S_n}$, `Store`, `Store`, $\underline{R}$ → `Bool`

where S_i is the effective sort[4] associated with the ith parameter and R is the effective sort of the pseudovariable `result`. If the function return type is `void` then the operator signature becomes

$\underline{S_1}$, ..., $\underline{S_n}$, `Store`, `Store` → `Bool`

Formally, the operator signature, *opSig*, is defined as follows.

```
maybeRetSort =
    if $retUTD = VoidUTD$
    then $\langle\rangle$
    else $\langle effRetSort \rangle$

opSig =
    $(\mu\ OpSig\ |$
        $opDom = effPrmSorts\ ^\frown$
                 $\langle \texttt{Store}, \texttt{Store} \rangle\ ^\frown$
                 $maybeRetSort\ \wedge$
        $opRan = \texttt{Bool})$
```

8.6.4.2 Function Specification Operator Definition

Uses: *id* (p. 154), *prmIds* (p. 155), *effPrmSorts* (p. 168), *retUTD* (p. 155), *effRetSort* (p. 170), *precond* (p. 158), *postcond* (p. 159).

Defines: *opDef*.

The definition of the operation is derived principally from the function specification body but it is also affected by the function type. This is because the type declarations of the parameters and the function return type can contribute implicit constraints. Given that

$$\begin{aligned} prm = (&\lambda\, i : \operatorname{dom} prmIds \bullet \\ &LVarTm(prmIds\, i, effPrmSorts\, i)\end{aligned}$$

the operation definition *opDef*, is

[4]Effective types and sorts are defined in Section 8.6.4.4.1.

$\underline{id}$`(`$\underline{prm\ 1}$`, ..., `$\underline{prm(\#prmIds)}$`,`
`pre:Store, post:Store, result:`$\underline{effRetSort}$`):Bool`
`==` $\underline{precond} \Rightarrow \underline{postcond}$

if the function return type is not `void`. If the return type is `void`, then the **`result`** argument is not included as part of the argument list of *id*. The operation precondition and postcondition are defined in Sections 8.6.4.2.1 and 8.6.4.2.2, respectively.

8.6.4.2.1 Precondition

Uses: *prmIds* (p. 155), *reqExp* (p. 154), *effPrmSorts* (p. 168), *effPrmLSLPreds* (p. 168), *preSig* (p. 167).

Defines: *precond*.

The precondition is defined to be the conjunction of the requires clause predicate and the implicit constraints derived from the parameter declarations.

$$precond = [\![\underline{reqTm} \wedge \underline{preImpLSLPred}]\!]$$

reqTm is the LSL predicate corresponding to the LCL predicate in the requires clause. *reqTm* is obtained by elaboration under *PreCtx* and the signature *preSig*. *preSig* is defined in Section 8.6.4.3.

$$(PreCtx, preSig) \vdash_{\text{Expr}} reqExp \Rrightarrow reqTm$$
$$tmSort\ reqTm = \texttt{Bool}$$

Elaborating under *PreCtx* ensures that *reqExp* does not contain LCL operators that make implicit references to the generic and post-states.

The implicit constraint imposed by the function parameters (*preImpLSLPred*) is the conjunction of the constraints contributed by each parameter

$$\underline{preImpLSLPreds\ 1} \wedge \underline{preImpLSLPreds\ 2} \wedge \ldots$$
$$\wedge\ \underline{preImpLSLPreds(\#preImpLSLPreds)}$$

The implicit constraint contributed by a parameter is obtained as follows. In each parameter type constraint, *effPrmLSLPreds i*, we replace all occurrences of

- the anonymous variable by the parameter name (represented as a term),

- the logical variable **any** (denoting an arbitrary state) by **pre** (denoting the pre-state)

$$
\begin{aligned}
&\#preImpLSLPreds = \#prmIds \\
&(\forall\, i : \mathrm{dom}\, prmIds \bullet \\
&\qquad \mathbf{let}\ subst == \{(\mathtt{any}, \mathtt{Store}) \mapsto LVarTm(\mathtt{pre}, \mathtt{Store})\} \bullet \\
&\qquad \mathbf{let}\ lslPred == appSubst\ subst\ (effPrmLSLPreds\ i); \\
&\qquad\quad tm = LVarTm(prmIds\ i, effPrmSorts\ i) \bullet \\
&\qquad\quad\ preImpLSLPreds\ i = anonSubst\ tm\ lslPred)
\end{aligned}
$$

As a concrete example, consider a parameter named `p` with the effective type `obj int[3]`. The type constraint derived from this type is

```
dim(val(any:Store,anonLVarNm:Obj[Arr[int]]):Arr[int]) = 3:int
```

Hence, the implicit constraint contributed by the parameter `p` will be

```
dim(val(pre:Store,p:Obj[Arr[int]]):Arr[int]) = 3:int
```

8.6.4.2.2 Postcondition

Uses: *prmIds* (p. 155), *ensExp* (p. 154), *modLSLPred* (p. 160), *trashLSLPred* (p. 164), *postImpLSLPred* (p. 165), *retImpLSLPred* (p. 166), *postSig* (p. 167).

Defines: *postcond*.

The postcondition is defined in terms of the meaning of the modifies, trashes, and ensures clauses, and the implicit constraints derived from the parameter declarations and the function return type.

$$
\begin{aligned}
postcond = [\![&\underline{modLSLPred} \land \\
&\underline{trashLSLPred} \land \\
&\underline{ensTm} \land \\
&\underline{postImpLSLPred} \land \\
&\underline{retImpLSLPred}]\!]
\end{aligned}
$$

ensTm is described next. The other *postcond* conjuncts are described in the sections that follow.

ensTm is the LSL predicate corresponding to the LCL predicate in the ensures clause obtained by elaboration under *PostCtx* and the signature *postSig*. *postSig* is defined in Section 8.6.4.3.

$$(\mathit{PostCtx}, \mathit{postSig}) \vdash_{\mathtt{Expr}} \mathit{ensExp} \Rrightarrow \mathit{ensTm}$$
$$\mathit{tmSort}\ \mathit{ensTm} = \mathtt{Bool}$$

Elaborating under *PostCtx* ensures that *ensExp* does not contain any occurrences of the LCL operators that make implicit reference to the generic state.

8.6.4.2.3 *modLSLPred*

Uses: *preSig* (p. 167), *modExps* (p. 154).

Defines: *modLSLPred*.

Each expression in the modifies clause must elaborate under *PreCtx* and the signature *preSig* and it must denote an object.

$$(\mathit{PreCtx}, \mathit{preSig}) \vdash_{\mathtt{SeqExpr}} \mathit{modExps} \Rrightarrow \mathit{modTms}$$
$$\forall\, \mathit{tm} : \operatorname{ran} \mathit{modTms} \bullet \mathit{tmSort}\ \mathit{tm} \in \operatorname{ran} \mathit{objSN}$$

modTms is the sequence of LSL terms corresponding to the list of LCL expressions given in the modifies clause. The contribution of the modifies clause to the postcondition, *modLSLPred*, is defined as

$$\mathtt{modAtMost(}\underline{\mathit{tms}}\mathtt{,pre:Store,post:Store):Bool}$$

where

$$\mathit{tms} == \mathit{mkSetDObjTm}\ \mathit{modTms}$$

The function `modAtMost` is described in Section 8.6.4.2.4 and *mkSetDObjTm* is defined next.

For a given sequence of terms $\mathit{tm}_1, \ldots, \mathit{tm}_k$, where each term tm_i is of an object sort $\mathit{objSN}\ \mathit{sort}_i$ (for some sort sort_i), $\mathit{mkSetDObjTm}\ \langle \mathit{tm}_1, \ldots, \mathit{tm}_k \rangle$ yields the LSL expression

```
empty \ins tm_1 \ins ... tm_k
```

representing the set of objects (of sort Obj) denoted by the terms in the sequence. The \ins operator is defined in Section 6.3.2.4.

$$
\begin{array}{|l}
\mathit{mkSetDObjTm} : \mathrm{seq}\ \mathit{Term} \twoheadrightarrow \mathit{Term} \\
\hline
\mathrm{dom}\ \mathit{mkSetDObjTm} = \{\mathit{tms} : \mathrm{seq}\ \mathit{Term} \mid \\
\qquad \forall\, \mathit{tm} : \mathrm{ran}\ \mathit{tms} \bullet \mathit{tmSort}\ \mathit{tm} \in \mathrm{ran}\ \mathit{objSN}\} \\
\\
\mathit{mkSetDObjTm}\ \langle\rangle = [\![\mathtt{empty():Set[Obj]}]\!] \\
\mathit{mkSetDObjTm}(\mathit{tms} \frown \langle \mathit{tm} \rangle) = \\
\qquad [\![(\underline{\mathit{mkSetDObjTm}\ \mathit{tms}}\ \mathtt{\backslash ins}\ \underline{\mathit{tm}})\mathtt{:Set[Obj]}]\!]
\end{array}
$$

8.6.4.2.4 Modified Clause The collection of objects denoted by the lvalues listed in the modifies clause is often called a *frame*. We will refer to it as a *modifies* frame to distinguish it from the trashes frame introduced in Section 8.6.4.2.5. In the presence of dependencies, we define the modifies frame to be the set of all objects explicitly listed in the modifies clause as well as all objects that are related to them by means of the object dependency relation.

The meaning of the modifies clause is often informally given as follows: every object that is outside the modifies frame must have the same abstract value in the pre- and post-states. A few points are worth highlighting. Firstly, phrasing the meaning of the modifies clause in terms of abstract values allows implementations of function specifications to have benevolent side-effects [Tan94, §2.4]. For example, an implementation of the following specification of member

```
mutable type intSet;
...
bool member(int i, intSet s) {
  ensures result = (i ∈ s);
}
```

would be permitted to changed the representation of s—e.g. to make subsequent invocations of member more efficient—provided it preserved the abstract value of s. Secondly, requiring that *every* object outside the frame have the same abstract value is senseless because not all objects outside of the frame contain values—an object may be inactive, or it may be active but not well-defined. Hence, we must focus our attention on objects that are active in the pre- and post-states. To make our final point we rephrase the meaning of the modifies clause in more general terms as follows: any object outside the modifies frame that is active in the pre- and post-states must not have its state changed in any way that is visible to the caller (of the specified function). Thus, if an active object outside the modifies frame is:

- not well-defined in the pre-state, then it must remain not well-defined in the post-state
- well-defined in the pre-state, then it must remain well-defined in the post-state, and furthermore, the abstract pre- and post-state values must be the same.

We formalize the meaning of the modifies clause, in Z, as follows.

$$
\begin{array}{|l}
\mathit{modAtMost} : \mathbb{F}\ \mathit{Obj} \longrightarrow (\mathit{Store} \leftrightarrow \mathit{Store}) \\
\hline
(\mathit{pre}, \mathit{post}) \in \mathit{modAtMost}\ \mathit{objs} \Leftrightarrow \\
\quad \textbf{let}\ \mathit{frame} = (\mathit{depOn}^{\sim} \cup \mathit{depOn})(\!|\mathit{objs}|\!) \bullet \\
\quad \forall x : \mathit{Obj} \mid \\
\qquad\quad x \in \mathit{pre.activeObjs} \land \\
\qquad\quad x \notin \mathit{frame} \land \\
\qquad\quad x \in \mathit{post.activeObjs} \bullet \\
\qquad (\mathit{pre.val}\ x, \mathit{post.val}\ x) \in \mathit{strongEq}(\mathit{sortAttr}\ x)
\end{array}
$$

If *objs* is the set of objects explicitly referenced in the modifies clause, then the modifies frame is the union of $\mathit{depOn}^{\sim}(\!|\mathit{objs}|\!)$, the set of objects whose values depend on the objects in *objs*, and $\mathit{depOn}(\!|\mathit{objs}|\!)$, the set of objects that the objects in *objs* depend on. The LSL version of *modAtMost* is represented as the operator

⟨*ModAndTrash opsig*⟩≡

```
  modAtMost: Set[Obj], Store, Store -> Bool
```

and is defined as

⟨*ModAndTrash assert eqn*⟩≡

```
  modAtMost(xs,pre,post) ==
    \A x: Obj (
          x \in activeObjs(pre)
       /\ ⟨x is outside the frame⟩
```

If the object x is ...				Then
active in the pre-state	active in the post-state	in the trashed frame	in the modifies frame	x is ...
no	no	–	–	inactive
no	yes	–	–	new
yes	no	no	–	*not possible*
yes	no	yes	–	trashed
yes	yes	–	no	persistent, unchanged
yes	yes	–	yes	persistent, may change

Figure 13: Modified and Trashed Objects in the Pre- and Post-states

```
    /\ x \in activeObjs(post)
    =>
      ⟨pre- and post-state values of x are strongly equal⟩) ;
```

This definition will be a part of the ModAndTrash trait (see Section 8.6.4.2.6). Another way of saying that x is outside the frame is "x is independent of the objects in xs."

⟨x *is outside the frame*⟩≡

```
  \A x1 (x1 \in xs =>
    ~depOn(x,x1) /\ ~depOn(x1,x))
```

⟨*pre- and post-state values of* x *are strongly equal*⟩≡

```
  strongEq(sortAttr(x), val(pre,x), val(post,x))
```

Also see Figure 13.

8.6.4.2.5 *trashLSLPred*

Uses: *preSig* (p. 167), *trashExps* (p. 154).

Defines: *trashLSLPred*.

The meaning of the trashes clause can be informally described as follows: any client-visible object that is active in the pre-state must be active in the post-state if it is outside of the trashes frame. The *trashes frame* is the set of all objects explicitly referenced in the trashes clause as well as all objects that are related to them by means of the object dependency relation. Each expression in the trashes clause must elaborate under *PreCtx* and the signature *preSig* and it must denote an object.

$$(PreCtx, preSig) \vdash_{\mathrm{Seq\ Expr}} trashExps \Rrightarrow trashTms$$
$$\forall\, tm : \mathrm{ran}\ trashTms \bullet tmSort\ tm \in \mathrm{ran}\ objSN$$

trashTms is the sequence of LSL terms corresponding to the list of LCL expressions given in the trashes clause.

The LSL predicate capturing the meaning of the trashes clause, *trashLSLPred*, is defined as

```
trashAtMost(tms,pre:Store,post:Store):Bool
```

where

$$tms == mkSetDObjTm\ trashTms$$

The function *mkSetDObjTm* is described in Section 8.6.4.2.3. If *objs* is the set of objects explicitly referenced in the trashed clause, then *trashAtMost objs* is a relation on pre- and post-state pairs that holds for those pairs that satisfy the constraint imposed by the trashes clause.

$$\begin{array}{|l}
trashAtMost : \mathbb{F}\ Obj \longrightarrow (Store \leftrightarrow Store) \\
\hline
(pre, post) \in trashAtMost\ objs \Leftrightarrow \\
\quad \mathbf{let}\ frame = (depOn^{\sim} \cup depOn)(\!|objs|\!) \bullet \\
\quad \forall\, x : Obj \mid \\
\qquad\quad x \in pre.activeObjs \land \\
\qquad\quad x \notin frame \bullet \\
\qquad x \in post.activeObjs
\end{array}$$

Also see Figure 13. We define an LSL version of the *trashAtMost* operator.

⟨*ModAndTrash opsig*⟩+≡
```
  trashAtMost: Set[Obj], Store, Store -> Bool
```
⟨*ModAndTrash assert eqn*⟩+≡
```
  trashAtMost(xs,pre,post) ==
    \A x: Obj (
         x \in activeObjs(pre)
      /\ ⟨x is outside the frame⟩
      =>
         x \in activeObjs(post));
```

This definition will be a part of the `ModAndTrash` trait (see Section 8.6.4.2.6).

8.6.4.2.6 `ModAndTrash` Trait

The trait `ModAndTrash` is used to hold the LSL definitions of the functions `modAtMost` (Section 8.6.4.2.4) and `trashAtMost` (Section 8.6.4.2.5). The overall structure of the trait is

⟨*ModAndTrash.lsl*⟩≡

```
ModAndTrash: trait
  includes
    ⟨ModAndTrash include⟩
  introduces
    ⟨ModAndTrash opsig⟩
  asserts
    \forall ⟨ModAndTrash assert var decl⟩
        ⟨ModAndTrash assert eqn⟩
```

The theory over the `Store` sort is defined in the `Store` trait which we must include

⟨*ModAndTrash include*⟩≡

```
Store, Set(Obj,Set[Obj])
```

Here are the variable declarations that are relevant to this trait.

⟨*ModAndTrash assert var decl*⟩≡

```
x, x1: Obj, xs: Set[Obj], pre, post: Store
```

8.6.4.2.7 *postImpLSLPred*

Uses: *prmIds* (p. 155), *effPrmLSLPreds* (p. 168).

Defines: *postImpLSLPred*.

postImpLSLPred is the contribution to the function postcondition of the implicit constraints derived from the parameter type constraints. It is defined as

$$\underline{\mathit{postImpLSLPreds}\ 1} \ \wedge\ \underline{\mathit{postImpLSLPreds}\ 2} \ \wedge\ \ldots$$
$$\wedge\ \underline{\mathit{postImpLSLPreds}(\#\mathit{postImpLSLPreds})}$$

For the *i*th parameter, *postImpLSLPreds i* is derived from the *i*th parameter constraint, *effPrmLSLPreds i*, by replacing all occurrences of

- the anonymous variable by the parameter name (represented as a term),

- the logical variable any (denoting an arbitrary state) by post (denoting the post-state)

$$
\begin{array}{l}
\#postImpLSLPreds = \#prmIds \\
(\forall\, i : \mathrm{dom}\, prmIds \bullet \\
\quad \mathbf{let}\ subst == \{(\mathtt{any}, \mathtt{Store}) \mapsto LVarTm(\mathtt{post}, \mathtt{Store})\} \bullet \\
\quad \mathbf{let}\ lslPred == appSubst\ subst\ (effPrmLSLPreds\, i); \\
\qquad tm = LVarTm(prmIds\, i, effPrmSorts\ i) \bullet \\
\qquad\ postImpLSLPreds\ i = anonSubst\ tm\ lslPred)
\end{array}
$$

8.6.4.2.8 *retImpLSLPred*

Defines: *retImpLSLPred*.

If the function return type is not `void` then the type constraint of the return type must be imposed on `result`. If the function return type is `void`, then the type constraint is `true`—i.e., there is no constraint. In the current version of the definition, it is not possible for the function return type to contribute implicit constraints, hence we provide a simplified definition

$$retImpLSLPred = [\![\mathtt{true():Bool}]\!]$$

8.6.4.3 Signatures

Uses: *gvarIds* (p. 155), *prmSigInc* (p. 168), *retUTD* (p. 155), *retSigInc* (p. 170), *0Sig* (p. 154).

Defines: *preSig*, *postSig*.

The expressions that occur in a function specification body can only contain variable identifiers denoting either

- a variable that appears in the global variable list of the function specification header,
- one of the function parameters, or
- the pseudovariable `result`.

Only the ensures clause predicate can refer to `result`.

The elaboration of the expressions in the requires, modifies and trashed clauses is done under the *preSig* signature which is obtained as follows:

- we obtain a restricted version of the global signature *θSig* by hiding the global variables that do not appear in the global variable list of the function specification header

  $$restrictedSig = sigResVar\ \theta Sig\ (\mathrm{ran}(ordId \circ gvarIds))$$

- the resulting environment is enriched with the function parameter declarations

  $$preSig == addSig(restrictedSig, prmSiginc)$$

If a parameter has the same name as a globally declared ordinary identifier then the parameter declaration will mask the global declaration (as is the case in C). That is, a function specification opens a new scope.

Elaboration of the predicate in the ensures clause is done under *postSig*. If the function return type is `void`, then *postSig* is *preSig*, otherwise *postSig* is obtained by enriching *preSig* with the declaration of `result`.

```
postSig ==
    if $retUTD = VoidUTD$
    then $preSig$
    else $addSig(preSig, retSigInc)$
```

8.6.4.4 Function Parameters and `result`

8.6.4.4.1 Effective Types The following discussion is phrased in terms of function parameters but the statements that are made also apply to the pseudovariable `result`.

Function parameters are modeled as logical variables in both *ISig*'s and LSL signatures. Within an LSL signature a parameter is usually given the sort of the parameter type. There are two exceptions to this rule: parameters of array types and mutable abstract types are associated with the object sort of the parameter type. Within a given LCL signature *sig*, a logical variable of type *td* in *sig.isig* must be mirrored (without exception) by a logical variable of the sort of *td* in the LSL signature *sig.lsig*.

Since parameters of array types and mutable abstract types are associated with the object sort of the parameter type, we must change the type associated with these parameters in the *ISig*. Thus arises the concept of the *effective type*, in contrast to the declared type, of a parameter.

The effective type of an array or mutable abstract type `T` is `obj T`. The effective type of any other type `T` is `T`. $td \in dclTpIsEffTp$ is true if the effective type of the type denotation td is td[5].

$$\begin{array}{|l}
dclTpIsEffTp : \mathbb{P}\ TpDen \\
\hline
td \in dclTpIsEffTp \Leftrightarrow \\
\qquad td.utd \notin \operatorname{ran} ArrUTD
\end{array}$$

The function $addObjQ2TN$ can be used to add an `obj` qualifier to a type name.

$$\begin{array}{|l}
addObjQ2TN : TpNm \longrightarrow TpNm \\
\hline
addObjQ2TN\ \theta TpNm' = \\
\qquad (\mu\ TpNm \mid tpQuals = tpQuals' \cup \{ObjQual\} \land \\
\qquad\qquad utd = utd')
\end{array}$$

Note that in the prototype of a function specification we preserve the declared types of the parameters. Effective types are only relevant when elaborating expressions occurring in a function specification body.

8.6.4.4.2 Function Parameters

Uses: *prmIds* (p. 155), *prmTNs* (p. 155), *prmTDs* (p. 155), *θSig* (p. 154).

Defines: *effPrmSorts*, *effPrmLSLPreds*, *prmSigInc*.

The sequences of effective parameter type names is given by *effPrmTNs*. The corresponding sequence of effective type denotation and effective type constraint pairs is *effPrmTDPreds*.

[5]The given definition is tailored for the current version of the definition of LCL.

$$
\begin{array}{|l}
\mathit{effPrmTNs} : \mathrm{seq}\ \mathit{TpNm} \\
\mathit{effPrmTDPreds} : \mathrm{seq}(\mathit{TpDen} \times \mathit{LSLPred}) \\
\mathit{effPrmTDs} : \mathrm{seq}\ \mathit{TpDen} \\
\mathit{effPrmLSLPreds} : \mathrm{seq}\ \mathit{LSLPred} \\
\mathit{effPrmSorts} : \mathrm{seq}\ \mathit{SortNm} \\
\hline
\#\mathit{effPrmTNs} = \#\mathit{prmTNs} \\
(\forall i : \mathrm{dom}\ \mathit{effPrmTNs} \bullet \\
\quad \mathit{effPrmTNs}\ i = \\
\quad\quad \textbf{if}\ \mathit{prmTDs}\ i \in \mathit{dclTpIsEffTp} \\
\quad\quad \textbf{then}\ \mathit{prmTNs}\ i \\
\quad\quad \textbf{else}\ \mathit{addObjQ2TN}(\mathit{prmTNs}\ i)) \\
\theta \mathit{Sig} \vdash_{\mathrm{SeqTpNm}_r} \mathit{effPrmTNs} \Rrightarrow \mathit{effPrmTDPreds} \\
\mathit{effPrmTDs} = \mathit{first} \circ \mathit{effPrmTDPreds} \\
\mathit{effPrmLSLPreds} = \mathit{second} \circ \mathit{effPrmTDPreds} \\
\mathit{effPrmSorts} = \mathit{td2SN} \circ \mathit{effPrmTDs}
\end{array}
$$

The effective type, type constraint and sort of the ith parameter are $\mathit{effPrmTDs}\ i$, $\mathit{effPrmLSLPreds}\ i$, and $\mathit{effPrmSorts}\ i$, respectively.

For each parameter we define an LCL signature increment ($\mathit{prmSigIncs}\ i$) to hold the declaration of that parameter as a spec logical variable with the appropriate effective type.

$$
\begin{array}{|l}
\mathit{prmSigIncs} : \mathrm{seq}\ \mathit{Sig} \\
\hline
\#\mathit{prmSigIncs} = \#\mathit{prmIds} \\
\forall i : \mathrm{dom}\ \mathit{prmIds};\ \mathit{Sig};\ \mathit{TBS} \mid \\
\quad \mathit{prmSigIncs}\ i = \theta \mathit{Sig} \bullet \\
\quad (\text{“}\textbf{let}\ \ \text{local}\ \mathit{prmSigIncs}\ \text{defs”} \bullet \\
\quad\quad \mathit{isig} = \mathit{mkISig}(\mathit{ordId}(\mathit{prmIds}\ i), \theta \mathit{Attr}) \wedge \\
\quad\quad \mathit{tb} \in \mathit{mkTB}(\mathit{refs}, \mathit{lvarNSs}, \mathit{opNmSigs}, \mathit{lslPreds}))
\end{array}
$$

Each LCL signature increment ($\mathit{prmSigIncs}\ i$) is defined in terms of its component *ISig* and LSL signatures. Each *ISig* increment contains the declaration of a parameter as a spec logical variable. The LSL signatures are derived from a special trait body defined in terms of a trait body *tb*. For the ith parameter, the trait body *tb* contains the declaration of the parameter as a logical variable of sort *effPrmSort i* and any trait references necessary to define the theories associated with the sort of the parameter. We define "**let** local *prmSigIncs* defs" as

$$\begin{array}{l}
\textbf{let}\ \mathit{dclQual} == \mathit{SpecQual};\\
\mathit{idKind} == \mathit{LVarIdK};\\
\mathit{td} == \mathit{effPrmTDs}\ i;\\
\mathit{cptSorts} == \mathit{cptSN}^{*}(\!|\{\mathit{effPrmSorts}\ i\}|\!) \bullet\\
\textbf{let}\ \mathit{refs} == \mathit{sort2Ref}(\!|\mathit{cptSorts}|\!);\\
\mathit{lvarNSs} == \{\mathit{prmIds}\ i \mapsto \mathit{effPrmSorts}\ i\};\\
\mathit{opNmSigs} == \varnothing;\\
\mathit{lslPreds} == \varnothing
\end{array}$$

The LCL signature increment that combines all of the parameter declarations is obtained by combining the individual parameter environment increments

$$\begin{array}{|l}
\mathit{prmSigInc} : \mathit{Sig}\\
\hline
\mathit{prmSigInc} == \mathit{foldLL}\ \mathit{addSig}\ \mathit{emptySig}\ \mathit{prmSigIncs}
\end{array}$$

8.6.4.4.3 Pseudovariable result

Uses: *retUTN* (p. 155), *retUTD* (p. 155), *θSig* (p. 154).

Defines: *effRetSort*, *effRetLSLPred*, *retSigInc*.

If the function return type is not `void`, then an environment increment containing the declaration of the pseudovariable `result` will be needed. The effective type name, type denotation, type constraint and sort of `result` are defined as follows.

$$
\begin{array}{|l}
retTN, \\
effRetTN : TpNm \\
retTD, \\
effRetTD : TpDen \\
effRetLSLPred : LSLPred \\
effRetSort : SortNm \\
\hline
retTN = (\mu\ TpNm \mid tpQuals = \varnothing \wedge utn = retUTN) \\
retTD = utd2TD\ retUTD \\
effRetTN = \textbf{if}\ retTD \in dclTpIsEffTp \\
\qquad\qquad \textbf{then}\ retTN \\
\qquad\qquad \textbf{else}\ addObjQ2TN\ retTN \\
\\
\theta Sig \vdash_{\mathrm{TpNm}_r} effRetTN \Rrightarrow (effRetTD, effRetLSLPred) \\
effRetSort = td2SN\ effRetTD
\end{array}
$$

The LCL signature increment (*retSigInc*) is defined in terms of its component *ISig* and LSL signatures. The *ISig* increment contains the declaration of `result` as a `spec` logical variable. The LSL signatures are derived from a special trait body defined in terms of the trait body *tb*. The trait body *tb* contains the declaration of `result` as a logical variable of sort *effRetSort* and any trait references necessary to define the theories associated with the sort of `result`.

$$
\begin{array}{|l}
retSigInc : Sig \\
\hline
\exists\, Sig;\ TBS \mid \\
\quad retSigInc = \theta Sig \bullet \\
\quad (\text{“}\textbf{let}\ retSigInc\ \text{local defs”} \bullet \\
\qquad isig = mkISig(ordId\, \texttt{result}, \theta Attr) \wedge \\
\qquad tb \in mkTB(refs, lvarNSs, opNmSigs, lslPreds))
\end{array}
$$

where "**let** *retSigInc* local defs" is

$$
\begin{array}{l}
\textbf{let } dclQual == SpecQual; \\
idKind == LVarIdK; \\
td == effRetTD; \\
cptSorts == cptSN^{*}(\!|\{effRetSort\}|\!) \bullet \\
\textbf{let } refs == sort2Ref(\!|cptSorts|\!); \\
lvarNSs == \{\texttt{result} \mapsto effRetSort\}; \\
opNmSigs == \varnothing; \\
lslPreds == \varnothing
\end{array}
$$

8.7 Specifications

The elaboration of a specification is done against a global environment and it yields the environment increment corresponding to the given specification.

$$Env \vdash_{\mathsf{Spec}_r} Spec \Rrightarrow Env$$

Either a specification is empty, in which case it corresponds to the empty environment increment,

$$\frac{}{env \vdash_{\mathsf{Spec}_r} \langle\rangle \Rrightarrow emptyEnv} \qquad \boxed{\text{Spec0}}$$

or, it contains at least one component *cpt* followed by the rest of the specification *spec*.

Let *env* denote the global environment. If *cpt* elaborates to the environment increment *env'* under *env*, and *spec* elaborates to the environment increment *env''* under *env* enriched with the declaration of *cpt*, then the environment increment corresponding to $\langle cpt\rangle \frown spec$ is the combined environment increments of *cpt* and *spec*:

$$\frac{\begin{array}{c} env \vdash_{\mathsf{Cpt}_r} cpt \Rrightarrow env'; \\ addEnv(env, env') \vdash_{\mathsf{Spec}_r} spec \Rrightarrow env'' \end{array}}{env \vdash_{\mathsf{Spec}_r} \langle cpt\rangle \frown spec \Rrightarrow addEnv(env', env'')} \qquad \boxed{\text{Spec}}$$

8.8 Related Work

8.8.1 LCL

In his thesis, Tan rigorously describes the "interesting aspects of the semantics of LCL" [Tan94, Chapter 7]. Although Tan gives a meaning to some of the components of a specification, he does not address the issue of the meaning of a specification in its entirety. Constant and variable declarations are briefly covered. We treat two problematic aspects of Tan's formalization of function specifications next.

8.8.1.1 Implicit Pre- and Postconditions

Tan defines the meaning of a function specification by a predicate schema of the form[6]

$$R \Rightarrow (M \wedge E)$$

where R and E stand for the requires and ensures clauses respectively, and M is a translation of the modifies clause [Tan94, §7.4.1]. This definition is incorrect since there may be implicit constraints due to parameters (Section 2.4.2) and, in LCL 2.4, non-trashed objects (Section 2.4.3). The meaning of a function specification is in fact of the form

$$(R \wedge I) \Rightarrow (M \wedge E \wedge J)$$

where I and J represent the implicit part of the pre- and postconditions of a function specification (Section 8.6.4.2).

Although Tan omits the implicit constraints from his definition of the meaning of a function specification he recognizes two kinds of implicit condition associated with function specifications. The first is the implicit condition, discussed in Section 2.4.3, concerning the non-trashing, under certain circumstances, of some of the objects listed in the modifies clause. The second concerns constraints that can be associated with **typedef** names[7] [Tan94, §7.4.2]. We have shown, in Section 2.4.2, that there are also implicit conditions related to function parameters and the pseudovariable result. The InitMod convention[8] contributes implicit constraints as well (although the convention

[6]In LCL 2.4 we see the introduction of **checks** and **claims** clauses as a part of function specifications [Tan94, §7.4]. We do not consider the meanings of specifications that contain these clauses.

[7]Constraints related to **typedefs** are not formalized in this version of the semantic definition.

[8]The InitMod convention states that if a module **M** has a module initialization function named **initMod_M**, then there is the implicit constraint that **initMod_M** must be the first function from **M** to be invoked [GH93, §5.3], [Tan94, §7.5], [Cha94].

can be defined as a syntactic sugar).

8.8.1.2 Meaning of the Modifies Clause

Tan formalizes the meaning of the modifies clause as

```
∀ i: AllObjects ((i ∈ domain(pre) ∧ i ∉ modifiedObjs)
                      ⇒ i' = i^)                                    (*)
```

where "AllObjects is the disjoint sum of the object sorts of the mutable types", and modifiedObjs is the union of the set of base objects (Section 6.6.1) of the expressions listed in the modifies clause [Tan94, §7.4.3]. The expression domain(pre) represents the set of objects visible to the client in the pre-state. There are two main problems with this formalization. Firstly, AllObjects is only defined informally: Tan does not show how this "disjoint sum" can be expressed as a part of the formal model. Secondly, the use of base objects does not adequately model object dependencies (as related to the meaning of the modifies clause) as we illustrate next.

From the following specification

```
int a[10];
void f(void) int a[10]; {
  modifies a[1];
  ensures  a[1]^ ≠ a[1]';
}
```

we can conclude that:

- the array a and its members are among the objects that are visible to clients

  `{a, a[0], a[1], ..., a[9]} ⊆ domain(pre)`

- f can only modify a[1]

  `modifiedObjs = {a[1]}`

  because a[1] is the only expression listed in the modifies clause and since the set of base objects of a[1] is {a[1]}.

Since a is active in the pre-state and it is not in `modifiedObjs`, by Tan's formalization of the meaning of the modifies clause (*), we may conclude that a, and hence all of its member objects, must have the same values in the pre- and post-states. Formally,

```
    a ∈ domain(pre)  ∧  a ∉ modifiedObjs
 ⇒  a^ = a'                                     [By (*)]
 ⇒  a[1]^ = a[1]'                  [Since (a•)[1] = (a[1])•]
```

which contradicts the predicate in the ensures clause; no such contradiction should exist. The problem is that a should also be in `modifiedObjs`.

8.8.2 LM3

The idea of defining the (part of the) meaning of a Larch interface language in terms of LSL was originally proposed by Jones in his work on the semantics of LM3 [Jon92]. As we have done, Jones represents LCL constants and global variables by LSL constant operators. He also defines a representation for Modula-3 objects.

By means of an example (written in C), we briefly explain his representation for function specifications followed by arguments stating that, although his translation is suitable for type checking, it is inadequate for reasoning about function specifications.

In his LSL translation of the meaning of the (non-interface part of the) following LCL function specification

```
int f(int pv) int gv; {
  required 0 < pv;
  modifies gv;
  ensures  result = gv^ ∧ gv' = pv;
}
```

Jones would introduce the operators

```
f_Id: → Proc
f_pv: → int
gv: → int_loc
f_RESULT: → int
```

along with the assertion

```
0 < f_pv ⇒ (isMod(gv)
               ∧ f_RESULT = σ(gv,pre)
               ∧ σ(gv,post) = f_pv)
```

In our model, int_loc corresponds to Obj [int] and σ(x,any) is written as val(any,x). The constant operator f_Id of sort Proc represents the specifi [illegible]n. Unfortunately, the Proc sort and the isMod operator are not defined. [illegible]d operator is to be used to encode the meaning of the modifies clause; the absence of its definition is a major omission. Function parameters and the pseudovariable result are modeled by constant operators. This makes reasoning about the meaning of function specifications very impractical since, to prove properties of specific calls to a function, we must instantiate the function parameters. Since function parameters are modeled as operators we encounter difficulties when we must reason about two (or more) instantiations, as would be required in the proof of the following claim[9]

```
claims Property_of_f {
  body    { gv = f(f(gv)); }
  ensures gv' = gv^;
}
```

[9]The given claim cannot be written directly, as it is presented, in LCL 2.4 since the body clause can only contain a single function call but it could be written indirectly by using an auxiliary function specification.

Chapter 9

Summary and Future Work

9.1 Summary

Formal methods have been described as the applied mathematics of computer systems engineering [Cra89]. Although immature in some important aspects, a judicious use of formal methods has been shown to contribute to the timely production of quality systems [BH94]. Underlying every formal method is a mathematically based notation—usually a specification language. Specification languages can be used during the entire software development process to document requirements, designs and the interface specifications for modules and program components. The specialization of a specification language to a particular programming language is an important characteristic of module interface specification languages (MISL's). The only MISL's are the Larch interface languages (among these LCL would seem to be the most developed and used) and an adaptation of the language used with the Trace Assertion Method (TAM) [PW89].

Our efforts to elaborate a semantic model for LCL lead to the identification of inadequacies and insufficiencies in the language and its informal definition. In particular, by introducing the concept of object dependency we illustrate, by means of realistic examples, that there is a need for LCL language constructs that would allow specifiers to describe and reason about object dependencies. We argue that the meaning of a function specification is affected by implicit parameter constraints and that these constraints have been poorly documented. The constraints are shown to be problematic—in particular. they are ambiguous and in some cases overly constraining. We show that the definition of the meaning of a function specification relative to trashed and non-trashed objects leads to a violation of the principle of referential transparency. We also argue that issues related to the trashing of aggregate and union objects have not been fully defined.

The version of LCL described in this thesis is named LCL′. The LCL′ language differs from its predecessor, LCL 2.4, principally in that:

- new operators have been added for describing object dependencies,
- the implicit constraints over pointer and array parameters have been dropped and new language operators have been added that allow specifiers to assert whether or not an object is active or well-defined,
- an optional trashes clause has been added to function specification bodies.

These changes increase the expressiveness of LCL′ and allow us to overcome the identified shortcomings of LCL 2.4. In particular, we eliminate the instance of referential opacity. One of our major design decisions has been to use LL, the logic underlying LSL, as the logical foundation for LCL.

The main contribution of this thesis is a semantic model within which a core of the stable aspects of LCL have been formally defined and the unsettled aspects of the language have been highlighted. The formally defined core consists of constant declarations, variable declarations and function specifications. The semantics of LCL is given in the form of a natural semantics [Kah87]. The meaning of the non-interface part of an LCL specification is captured in the form of an LSL trait body. There are significant advantages to this approach to the formal definition of LCL. Firstly, since the semantics is expressed at a "higher" level of abstraction, it is considerably simpler than would be, e.g., a corresponding denotational semantics. Secondly, use of LSL to describe the non-interface part of LCL specifications should make the semantics more accessible to the Larch community in general and to LCL users in particular. Finally, the inference rules that are used to define the semantics can be readily implemented. The result would be a tool that translates the non-interface part of an LCL specification into an LSL trait. Such a tool would allow specifiers to capitalize on other Larch tools (such as the LSL checker and the Larch Prover). The semantics is written in the Z specification language. We have chosen Z because it is an expressive and mature language that is in widespread use and for which tools are available. Although our model of program states is conventional, our model of the store is exceptional in that it supports the representation of object dependencies in their full static generality and of objects containing undefined values. Another important characteristic of our storage model is that each object has a fixed, state-independent sort attribute (in contrast to C where objects are untyped). Our model can none-the-less be used to represent an object that is referenced by lvalues of different types. Such an object is modeled by a collection of mutually dependent objects (where each object has a single type).

To increase our confidence in the accuracy of the semantic definition we have compiled and applied a suite of tests (Appendix A). Each test includes an LCL specification along with the LSL trait that is meant to capture the meaning of the non-interface part of the LCL specification.

We have argued that MISL's are an excellent way of introducing formal methods into industrial settings. MISL's can be gradually integrated at various levels of rigor into new software projects as well as retroactively introduced into projects developed without formal methods. Furthermore, automated tools can perform more checks with the use of interface specifications than with the use of other kinds of specification.

Finally, we note that the utility of our semantic model extends beyond its use in the formal definition of LCL. The semantic model (particularly the storage model) can serve as a base for the formal definition of other imperative programming languages and MISL's—especially the Larch interface languages LCPP and LM3. In fact, changes to LCPP have already been made in response to the research results reported in this thesis.

9.2 Future Work

A natural succession to this work is the formal definition of the remaining LCL constructs. This is likely to induce further changes to the language and may require LCL user input to determine which language features are needed most. Even within the bounds of the LCL constructs that are covered in the current version of the LCL semantics, we take note of the following issues that require further investigation.

- We have chosen to drop the implicit constraints on function parameters of pointer and array types. Consequently, specifiers must explicitly document whether or not the objects that can be referred to by function parameters of these types are active or well-defined. The explicit statement of these properties unnecessarily clutters the requires clauses of most function specifications. The use of parameter qualifiers (e.g. Ada-like qualifiers `in`, `out`, and `inout`) with appropriately defined semantics would seem to offer an acceptable solution to this problem.

- As we have noted in Section 3.2.1, we model *static* dependency relationships between objects. Will the model be adequate in practice or will we need to model dynamic dependencies too? If dynamic dependencies are to be supported then appropriate language constructs will have to be devised.

- The role of LCL constants is not clear: are LCL constants values that are computable at compile-time or run-time? In the Larch book, Guttag and Horning claim that an LCL constant can be implemented by a `const` variable [GH93, §5]. If this is true, then clients will not be able to use an LCL constant in those situations where a constant expression[1] is required. It is also unclear what kinds of restriction should be imposed on the types of constants. For example, should array constants be permitted? Such constants can be given a meaning under the semantics (as we have done), but does this meaning correspond to a practical need? Are there implementation limitations that need to be taken into account? Another interesting question to consider is: should spec constants be permitted to have `obj`-qualified types?

- We have in effect defined LCL expressions as LSL terms. As a consequence, LCL operators are treated like LSL operators. Hence, users can redefine the meanings of LCL operators (by imposing additional constraints) or they can make use of operator overloading to define completely unrelated operations that merely use LCL operator names. Is such freedom really beneficial?

Our long-term goal is to achieve the complete definition of a formal system within which it would be possible to formally prove the correctness of implementations of LCL specifications. Preliminary definitions of correctness have already been sketched.

[1]The value of a constant expression must be computable at compile-time.

Finally, one of our short-term goals is to complete an SML implementation of the semantic definition. In addition to performing type checking, this tool will generate the LSL traits corresponding to the non-interface parts of given LCL specifications. Eventually this functionality could be incorporated into LCLint.

Bibliography

[Abr91] J.-R. Abrial. The B method for large software, specification, design and coding (abstract). In *[PT91a]*, pages 398–405, 1991.

[AW82] E. A. Ashcroft and W. W. Wadge. Prescription for semantics. *ACM TOPLAS*, 4(2):283–294, April 1982.

[B+92] E. Börger et al., editors. *Computer science logic : 6th Workshop, CSL '92*, volume 702 of *Lecture Notes in Computer Science*. Springer-Verlag, 1992.

[BH94] Jonathan P. Bowen and Michael G. Hinchey. Ten commandments of formal methods. Technical Report 350, University of Cambridge Computer Laboratory, September 1994. (Also IEEE Computer, Vol. 28, No. 4, April 1995.).

[BVNW87] F.J. Brandenburg, G. Vidal-Naquet, and M. Wirsing, editors. *STACS 87: 4th Annual Symposium on Theoretical Aspects of Computer Science*, volume 247 of *Lecture Notes in Computer Science*. Springer-Verlag, 1987.

[CGR93] Dan Craigen, Susan Gerhart, and Ted Ralston. An international survey of industrial applications of formal methods volume 1 purpose, approach, analysis and conclusions. Technical Report NIST GCR 93/626-V1, National Institute of Standards and Technology, Springfield, Virginia 22161, U.S.A., 1993.

[Cha94] P. Chalin. A formal definition of the LCL module initialization convention. Technical Note 240, Computer Science Department, Concordia University, June 1994.

[Che89] Jolly Chen. The Larch/Generic interface language. S. B. Thesis, Department of Electrical Engineering and Computer Science, MIT, 1989.

[Cra89] Dan Craigen, editor. *Formal Methods for Trustworthy Computer Systems (FM89)—a Workshop on the Assessment of Formal Methods for Trustworthy Computer Systems, 23-27 July 1989, Nova Scotia, Canada.* Springer-Verlag, 1989.

[EGHT94] David Evans, John V. Guttag, James J. Horning, and Yang Meng Tan. LCLint: A tool for using specifications to check code. In *Symposium on the Foundations of Software Engineering*, December 1994.

[EHO94] George W. Ernst, Raymond J. Hookway, and William F. Ogden. Modular verification of data abstractions with shared realizations. *IEEE Transactions on Software Engineering*, 20(4):288–307, April 1994.

[Eva94] David Evans. Using specifications to check source code. MIT/LCS/TR 628, Laboratory for Computer Science, MIT, June 1994. S.M. Thesis.

[FJKRdL89] L.M.G. Feijs, H.B.M. Jonkers, C.P.J. Koymans, and G.R. Renardel de Lavalette. Formal definition of the design language COLD-K. Technical report, Philips Research Laboratories, October 1989. Revised Edition.

[GH91] John V. Guttag and James J. Horning. LCL: A Larch interface language for C. Technical Report 74, DEC Systems Research Center, July 1991.

[GH92] Yuri Gurevich and James K. Huggins. The semantics of the C programming language. In *[B^+92]*, pages 274–308, 1992.

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

[GHM90] John V. Guttag, James J. Horning, and Andrés Modet. Report on the Larch Shared Language: Version 2.3. Technical Report 58, DEC Systems Research Center, April 1990.

[Heh93] Eric C. R. Hehner. *A Practical Theory of Programming.* Texts and Monographs in Computer Science. Springer-Verlag, 1993.

[HHJ$^+$87] C. A. R. Hoare, I. J. Hayes, He Jifeng, C. C. Morgan, A. W. Roscoe, J. W. Sanders, I. H. Sorensen, J. M. Spivey, and B. A. Sufrin. Laws of programming. *Communications of the ACM*, 30(8):672–686, August 1987.

[HMRC88] Richard C. Holt, Philip A. Matthews, J. Alan Rosselet, and James R. Cordy. *The Turing Programming Language: Design and Definition.* Prentice Hall, 1988.

[Hoa87] C.A.R. Hoare. An overview of some formal methods for program design. *IEEE Computer*, 20(9):85–91, September 1987.

[Hoa94] C.A.R. Hoare. Unified theories of programming. Monograph, Oxford University Computing Laboratory, July 1994.

[Hug95] Jim Huggins. Evolving algebras. World-wide Web, URL:`http://www.eecs.umich.edu/ealgebras`, August 1995.

[HW73] C.A.R. Hoare and N. Wirth. An axiomatic definition of the programming language Pascal. *Acta Informatica*, 2:335–355, 1973.

[IMPK93] Michal Iglewski, Jan Madey, David Lorge Parnas, and Philip C. Kelly. Documentation paradigms. CRL Technical Report 270, McMaster University, Telecommunications Research Institute of Ontario (TRIO), Ontario, Canada, July 1993.

[ISO] ISO/IEC 9899 : 1990 (E). *Programming languages—C.*

[JM94] C.B. Jones and C.A. Middelburg. A typed logic of partial functions reconstructed classically. *Acta Informatica*, 31(5):399–430, 1994.

[Jon90] Cliff B. Jones. *Systematic Software Development using VDM.* Computer Science Series. Prentice Hall International, second edition, 1990.

[Jon91] Kevin D. Jones. LM3: a Larch interface language for Modula-3: A definition and introduction. Technical Report 72, DEC Systems Research Center, June 1991.

[Jon92] Kevin D. Jones. A semantics for a Larch/Modula-3 interface language. In Ursula Martin and Jeannette M. Wing, editors, *First International Workshop on Larch.* Springer-Verlag, July 1992.

[Jon93] C.B. Jones. Reasoning about interference in an object-based design method. In *[WL93]*, pages 1–18, 1993.

[Kah87] Giles Kahn. Natural semantics. In *[BVNW87]*, pages 22–29, 1987.

[Kah93] Stefan Kahrs. Mistakes and ambiguities in the definition of Standard ML. Technical Report ECS-LFCS-93-257, Laboratory for Foundations of Computer Science, University of Edinburgh, April 1993.

[Knu92] Donald E. Knuth. *Literate Programming.* Number 27 in CSLI Lecture Notes. Center for the Study of Language and Information, 1992.

[KST94] Stefan Kahrs, Donald Sannella, and Andrzej Tarlecki. The definition of Extended ML. Technical Report ECS-LFCS-94-300, Laboratory for Foundations of Computer Science, University of Edinburgh, 1994.

[Lam89] Leslie Lamport. A simple approach to specifying concurrent systems. *Communications of the ACM*, 32(1):32–45, January 1989.

[LC95] Gary T. Leavens and Yoonsik Cheon. *Larch/C++ Reference Manual.* Department of Computer Science, Iowa State University, Ames, Iowa 50011-1040 USA, June 1995. Draft Revision: 1.117.

[Ler91] Richard Allen Lerner. *Specifying Objects of Concurrent Systems.* PhD thesis, School of Computer Science, Carnegie Mellon University, May 1991. TR CMU-CS-91-131.

[MdL94] C. A. Middelburg and G. R. Renardel de Lavalette. LPF and MPL$_\omega$ a logical comparison of VDM-SL and COLD K. In *[PT91b]*, pages 279 308, 1994.

[Mer74] The Merriam-Webster Dictionary. Pocket Books, New York, 1974.

[Mor90] Carroll Morgan. *Programming from Specifications.* Computer Science Series. Prentice Hall International, 1990.

[MT91] Robin Milner and Mads Tofte. *Commentary on Standard ML.* MIT Press, 1991.

[MTH90] Robin Milner, Mads Tofte, and Robert Harper. *The Definition of Standard ML.* MIT Press, 1990.

[MW85] Zohar Manna and Richard Waldinger. *The Logical Basis for Computer Programming—Volume 1: Deductive Reasoning.* Computer Science Series. Addison-Wesley, 1985.

[NHWG89] M. Neilson, K. Havelund, K. Wagner, and C. George. The RAISE language, method, and tools. *Formal Aspects of Computing*, 1:85 114, 1989.

[PM91] David Lorge Parnas and Jan Madey. Functional documentation for computer systems engineering (version 2). CRL Technical Report 237, McMaster University, Telecommunications Research Institute of Ontario (TRIO), Ontario, Canada, September 1991.

[PST91] Ben Potter, Jane Sinclair, and David Till. *An Introduction to Formal Specification and Z.* Computer Science Series. Prentice Hall, 1991.

[PT91a] S. Prehn and W.J. Toetenel, editors. *VDM'91: Formal Software Development Methods*, volume 552 of *Lecture Notes in Computer Science.* VDM Europe, Springer-Verlag, 1991. Volume 2: Tutorials.

[PT91b] S. Prehn and W.J. Toetenel, editors. *VDM'91: Formal Software Development Methods*, volume 551 of *Lecture Notes in Computer Science.* VDM Europe, Springer-Verlag, 1991. Volume 1: Conference Contributions.

[PW89] David Lorge Parnas and Yabo Wang. The trace assertion method of module interface specification. Technical Report 89-261, Queen's University at Kingston (Dept. of Computing and Information Science), Telecommunications Research Institute of Ontario (TRIO), Ontario, Canada, 1989.

[Ram94] Norman Ramsey. Literate programming simplified. *IEEE Software*, 11(5):97-105, September 1994.

[SK95] Kenneth Slonneger and Barry L. Kurtz. *Formal Syntax and Semantics of Programming Languages: A Laboratory Based Approach.* Addison-Wesley, 1995.

[Spi88] J.M. Spivey. *Understanding Z: A Specification Language and its Formal Semantics*, volume 3 of *Cambridge Tracts in Theoretical Computer Science.* Cambridge University Press, January 1988.

[Spi89] J.M. Spivey. *The Z Notation: A Reference Manual.* Computer Science Series. Prentice Hall International, 1989.

[Spi92] J.M. Spivey. *The Z Notation: A Reference Manual.* Computer Science Series. Prentice Hall International, second edition, 1992.

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

[Ten81] R. D. Tennent. *Principles of Programming Languages.* Computer Science Series. Prentice Hall International, 1981.

[Van94] Mark T. Vandevoorde. Exploiting specifications to improve program performance. MIT/LCS/TR 598, Laboratory for Computer Science, MIT, February 1994. Ph.D. Thesis.

[Win93] Glynn Winskel. *The Formal Semantics of Programming Languages: An Introduction.* Foundations of Computing Series. MIT Press, 1993.

[WL93] Jim C. P. Woodcock and P. G. Larsen, editors. *FME'93: Industrial-Strength Formal Methods*, volume 670 of *Lecture Notes in Computer Science.* Formal Methods Europe, Springer-Verlag, 1993.

[WZ91] Jeannette M. Wing and Amy Moormann Zaremski. Unintrusive ways to integrate formal specifications in practice. In *[PT91b]*, pages 545–569, 1991.

Appendix A

Tests

A semantic definition cannot be verified but it can be validated. For this purpose, we have created some simple tests. The tests are in the form of LCL specifications. The expected result of each test is given in the form of the C header file and the LSL trait that are to be generated from the LCL specification. The C header file contains a representation of the non-spec components that are a part of the *lSig* of the LCL signature corresponding to the test specification. These tests have been checked with LCLint and the LSL checker.

Test 002

LCL Code

⟨*Test002.lcl*⟩≡
```
 /* Test 002 */
 spec constant int c;
```

C Header

Empty

LSL Trait

⟨*Test002.lsl*⟩≡
```
 Test002: trait
   includes IntTp
   introduces
     c: -> int
```

Test 005

LCL Code

⟨*Test005.lcl*⟩≡
```
/* Test 005 */
int gv;
```

C Header

⟨*Test 005 Header*⟩≡
```
extern int gv;
```

LSL Trait

⟨*Test005.lsl*⟩≡
```
Test005: trait
  includes IntTp, SProjStore(int for S)
  introduces
    gv: -> Obj[int]
  asserts
    \forall any: Store

    % gv is always active
    gv \in activeObjs(any);

    % gv is independent of
    % every other global variable
    indep(empty \apd gv);
```

Test 022

LCL Code

⟨*Test022X.lcl*⟩≡
```
/* Test 022 */
constant int cA;
spec constant int cB[cA];
```

C Header

Empty

LSL Trait

⟨*Test022.lsl*⟩≡
```
Test022: trait
  includes IntTp, ArrTp(int)
  introduces
    cA: -> int
    cB: -> Arr[int]
  asserts equations
    dim(cB) == cA;
```

Test 025

LCL Code

⟨*Test025.lcl*⟩≡
```
/* Test 025 */
int gv[];
```

C Header

⟨*Test 025 Header*⟩≡
```
extern int gv[];
```

LSL Trait

⟨*Test025.lsl*⟩≡
```
Test025: trait
  includes IntTp, ArrTp(int)
  introduces
    gv: -> Obj[Arr[int]]

  asserts \forall any: Store
    gv \in activeObjs(any);
    indep(empty \apd gv);
```

Test 026

LCL Code

⟨*Test026.lcl*⟩≡
```
/* Test 026 */
constant int c;
int gv[c];
```

C Header

⟨*Test 026 Header*⟩≡
```
extern int gv[c];
```

LSL Trait

⟨*Test026.lsl*⟩≡
```
Test026: trait
  includes IntTp, ArrTp(int)
  introduces
    c: -> int
    gv: -> Obj[Arr[int]]

  asserts \forall any: Store

    gv \in activeObjs(any);
    indep(empty \apd gv);
    dim(gv) == c;
```

Test 027

LCL Code

⟨*Test027.lcl*⟩≡
```
/* Test 027 */
constant int cA, cB;
int gv[cA][cB];
```

C Header

⟨*Test 027 Header*⟩≡
```
extern int gv[cA][cB];
```

LSL Trait

⟨*Test027.lsl*⟩≡
```
Test027: trait
  includes IntTp, ArrTp(int), ArrTp(Arr[int])
  introduces
    cA,
    cB: -> int
    gv: -> Obj[Arr[Arr[int]]]

  asserts
    \forall any: Store, i: int
    gv \in activeObjs(any);
    indep(empty \apd gv);
    dim(gv) == cA;
   \A i (i \in inds(gv) => dim(gv[i]) = cB);
```

Test 100

LCL Code

⟨*Test100.lcl*⟩≡

```
  /* Test 100 */
  void skip(void) {
    requires true;
    modifies nothing;
  /*trashes  nothing;*/
    ensures  true;
  }
```

C Header

⟨*Test 100 Header*⟩≡

```
  extern void skip(void);
```

LSL Trait

⟨*Test100.lsl*⟩≡

```
  Test100: trait
    includes LCLAux
    introduces
      skip: Store, Store -> Bool

    asserts \forall pre, post: Store

      skip(pre,post) ==
          true =>
             modAtMost({}, pre, post)
             /\ trashAtMost({}, pre, post);
```

Test 101

LCL Code

⟨*Test101.lcl*⟩≡

```
  /* Test 101 */
```

```
void fp(int pv) {
  requires 0 < pv;
  modifies nothing;
/*trashes  nothing;*/
  ensures  true;
}
```

C Header

⟨*Test 101 Header*⟩≡

```
extern void fp(int pv);
```

LSL Trait

⟨*Test101.lsl*⟩≡

```
Test101: trait
  includes LCLAux, IntTp
  introduces
    fp: int, Store, Store -> Bool
  asserts
    \forall pre, post: Store, pv: int

    fp(pv,pre,post) ==
         0 < pv =>
            modAtMost({}, pre, post)
            /\ trashAtMost({}, pre, post);
```

Test 102

LCL Code

⟨*Test102.lcl*⟩≡

```
/* Test 102 */
int fr(void) {
  requires true;
  modifies nothing;
/*trashes  nothing;*/
  ensures  0 <= result /\ result <= 9;
}
```

C Header

⟨*Test 102 Header*⟩≡

```
extern int fr(void);
```

LSL Trait

⟨*Test102.lsl*⟩≡
```
Test102: trait
  includes LCLAux, IntTp
  introduces
    fr: Store, Store, int -> Bool
  asserts
    \forall pre, post: Store, result: int

    fr(pre,post,result) ==
        true =>
          (modAtMost({}, pre, post)
           /\ trashAtMost({}, pre, post)
           /\ 0 <= result /\ result <= 9);
```

Test 103

LCL Code

⟨*Test103.lcl*⟩≡
```
/* Test 103 */
int gv;

void fg(void) int gv; {
  requires 0 < gv^;
  modifies gv;
/*trashes  nothing;*/
  ensures  gv^ < gv';
}
```

C Header

⟨*Test 103 Header*⟩≡
```
extern int gv;
extern void fg(void);
```

LSL Trait

⟨*Test103.lsl*⟩≡
```
Test103: trait
  includes LCLAux, IntTp,
           Test005 % for gv
  introduces
    fg: Store, Store -> Bool
  asserts
```

```
    \forall pre, post: Store

    fg(pre,post) ==
        0 < val(pre,gv) =>
          (modAtMost({} \ins gv, pre, post)
           /\ trashAtMost({}, pre, post)
           /\ val(pre,gv) < val(post,gv));
```

Test 104

LCL Code

⟨*Test104.lcl*⟩≡

```
/* Test 104 */
int gv;

int fprg(int pv) int gv; {
  requires 0 < pv;
  modifies nothing;
/*trashes  nothing;*/
  ensures  result = gv^;
}
```

C Header

⟨*Test 104 Header*⟩≡

```
extern int gv;
extern int fprg(int pv);
```

LSL Trait

⟨*Test104.lsl*⟩≡

```
Test104: trait
  includes LCLAux, IntTp,
           Test005 % for gv
  introduces
    fprg: int, Store, Store, int -> Bool
  asserts
    \forall pre, post: Store, pv, result: int

    fprg(pv,pre,post,result) ==
        0 < pv =>
           (modAtMost({}, pre, post)
            /\ trashAtMost({}, pre, post)
            /\ result = val(pre,gv));
```

Test 205

LCL Code

⟨*Test205X.lcl*⟩≡
```
/* Test 205 */
int gv;

claims() {
  ensures  gv\activePre /\ gv\activePost;
}
```

C Header

⟨*Test 205 Header*⟩≡
```
extern int gv;
extern void Claim205(void);
```

LSL Trait

⟨*Test205.lsl*⟩≡
```
Test205: trait
  includes IntTp, Test005
  implies
    \forall pre, post: Store
        gv \in activeObjs(pre);
        gv \in activeObjs(post);
```

Test 226

LCL Code

⟨*Test226X.lcl*⟩≡
```
/* Test 226 */
constant int c;
int gv[c];

claims() {
  ensures  gv\activePre /\ gv\activePost
    /\ maxIndex(gv) = c - 1
    /\ \A i:int (i \in inds(gv) <=> 0 <= i /\ i < c)
    /\ \A i:int, j (i \in inds(gv) /\ j \in inds(gv) =>
            (i \neq j <=> indep(gv[i],gv[j])))
    /\ \A i:int (i \in inds(gv) =>
            depOn(gv,gv[i]) /\ depOn(gv[i],gv));
}
```

C Header

⟨*Test 226 Header*⟩≡
```
extern int gv[c];
```

LSL Trait

⟨*Test226.lsl*⟩≡
```
Test226: trait
  includes Test026
  implies
    \forall i,j:Int, pre, post: Store

    gv \in activeObjs(pre)  /\   gv \in activeObjs(post);
    maxIndex(gv) == c - 1;
    0 < c;
    \A i (i \in inds(gv) = 0 <= i /\ i < c);
    \A i \A j (i \in inds(gv) /\ j \in inds(gv) =>
         ((i \neq j) = indep(empty \apd gv[i] \apd gv[j])));
    \A i (i \in inds(gv) =>
         depOn(dwn(gv),dwn(gv[i]))
         /\ depOn(dwn(gv[i]),dwn(gv)));
```

Test 405

LCL Code

⟨*Test405X.lcl*⟩≡
```
/* Test 405 */
int gvA, gvB;

claims() {
  ensures  indep(gvB, gvA);
}
```

C Header

⟨*Test 405 Header*⟩≡
```
extern int gvA, gvB;
```

LSL Trait

⟨*Test405.lsl*⟩≡
```
Test405: trait
  includes IntTp, SProjStore(int for S)
  introduces
```

```
    gvA, gvB: -> Obj[int]
  asserts
    \forall any: Store

    gvA \in activeObjs(any);
    gvB \in activeObjs(any);
    indep(empty \apd gvA \apd gvB);
  implies equations
    indep(empty \apd gvA \apd gvB);
```

Test 426

LCL Code

⟨*Test426X.lcl*⟩≡

```
/* Test 426 */
constant int c;
int gvA[c], gvB;

claims {
  ensures  indep(gvB,gvA)
    /\ \A i:int (i \in inds(gvA) =>
           (gvA[i]\activePost /\ indep(gvA[i],gvB)))
    /\ gvA\wellDefPost <=>
        \A i:int (i \in inds(gvA) => gvA[i]\wellDefPost);
}
```

C Header

⟨*Test 426 Header*⟩≡

```
extern int gvA[c], gvB;
```

LSL Trait

⟨*Test426.lsl*⟩≡

```
Test426: trait
  includes IntTp, SProjStore(int for S),
           ArrTp(int)
  introduces
    c:   -> int
    gvA: -> Obj[Arr[int]]
    gvB: -> Obj[int]

  asserts
    \forall any: Store
  gvA \in activeObjs(any);
  gvB \in activeObjs(any);
  indep(empty \apd gvA \apd gvB);
  dim(gvA) == c;

implies
  \forall i: Int, post: Store

  indep(empty \apd gvA \apd gvB);
  \A i (i \in inds(gvA) =>
           (gvA[i] \in activeObjs(post)
            /\ indep(empty \apd gvA[i] \apd gvB)));
  gvA \in wellDefObjs(post) ==
     \A i (i \in inds(gvA) => gvA[i] \in wellDefObjs(post));
```

Appendix B

Concrete Syntax

The LCL grammar given here is essentially that defined by Tan [Tan94] for LCL 2.4. We have corrected a few minor errors.

```
interface          ::= {import | use | export | private | claim } *
import             ::= imports { id | " id " | < id > } +, ;
use                ::= uses traitRef+,;
export             ::= constDeclaration | varDeclaration | type | fcn
private            ::= spec { constDeclaration | varDeclaration | type }
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]
```

Figure 14: Concrete Syntax for LCL: Specifications

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* ;

Figure 15: Concrete Syntax for LCL: Functions

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

Figure 16: Concrete Syntax for LCL: Types

```
lclPredicate    ::= term
term            ::= if term then term else term | equalityTerm
                    | term logicalOp term
equalityTerm    ::= simpleOpTerm [ { eqOp | = } simpleOpTerm ]
                    | quantifier+ ( lclPredicate )
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 )
cLiteral        ::= intLiteral | stringLiteral | singleQuoteLiteral | floatLiteral
quantifier      ::= quantifierSym { varId : [ obj ] sortSpec }+,
varId           ::= id
fcnId           ::= id
sortId          ::= id
opId            ::= id
```

Figure 17: Concrete Syntax for LCL: Terms and Predicates

```
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*, )
```

Figure 18: Concrete Syntax for LCL: Claims

Appendix C

General Z Definitions

The relationship between the *pair*, and the projection functions *first*, and *second* is illustrated in Figure 19. The functions *first* and *second* are part of the standard mathematical toolkit [Spi89].

$[X, Y, X', Y']$

$pair : (X \times Y \longrightarrow X') \longrightarrow (X \times Y \longrightarrow Y') \longrightarrow X \times Y \longrightarrow X' \times Y'$

$pair\, \psi_1\, \psi_2 =$
$\quad (\lambda\, x : X;\ y : Y \bullet \psi_1(x, y) \mapsto \psi_2(x, y))$

$$\begin{tikzcd}
(x,y) \arrow[r, "g"] \arrow[d, "f"'] \arrow[dr, dotted, "pair\ f\ g"] & x' \\
y' & (x',y') \arrow[l, "second"] \arrow[u, "first"']
\end{tikzcd}$$

Figure 19: Relationship between *pair*, *first* and *second*.

Let '$\_\odot\_$' be a binary function (which we will write as an infix operator), then the value of $foldLL \odot b\, \langle x_1, \ldots, x_n \rangle$ is $((\ldots(b \odot x_1) \odot \ldots) \odot x_n$.

$[X]$

$foldLL : (X \times X \rightarrowtail\!\!\!\!\!\!\rightarrow X) \longrightarrow X \longrightarrow \mathrm{seq}\, X \rightarrowtail\!\!\!\!\!\!\rightarrow X$

$\forall f : X \times X \rightarrowtail\!\!\!\!\!\!\rightarrow X;\ b : X \bullet$
$\quad foldLL f\, b\, \langle\rangle = b \land$
$\quad foldLL f\, b\, (\langle x \rangle \frown xs) = foldLL f\, (f(b, x))\, xs$

For any set, X, *Reflexive*$[X]$ is the set of all reflexive relations over X's. Similarly, *Symmetric*$[X]$ and *Transitive*$[X]$ are the sets of symmetric and transitive relations over X's. The set of equivalence relations over X's is given by *EquivalenceRel*$[X]$.

$$\begin{array}{l}
\mathit{Reflexive}[X] == \{\, R : X \leftrightarrow X \mid \mathrm{id}\, X \subseteq R \,\} \\
\mathit{Symmetric}[X] == \{\, R : X \leftrightarrow X \mid \forall\, x, y : X \bullet \\
\qquad\qquad (x, y) \in R \Rightarrow (y, x) \in R \} \\
\mathit{Transitive}[X] == \{\, R : X \leftrightarrow X \mid \forall\, x, y, z : X \bullet \\
\qquad\qquad (x, y) \in R \wedge (y, z) \in R \Rightarrow (x, z) \in R \} \\
\mathit{EquivalenceRel}[X] == \mathit{Reflexive} \cap \mathit{Symmetric} \cap \mathit{Transitive}[X]
\end{array}$$

If R is a relation between X's and Y's, then *relToFun* R is a representation of R as a function that maps every x in the domain of R into the set of Y's related to x under R.

$$\begin{array}{|l}
[X, Y] \\
\hline
\mathit{relToFun} : (X \leftrightarrow Y) \longrightarrow (X \rightarrowtail\!\!\!\!\!+ \; \mathbb{P}\, Y) \\
\hline
\mathit{relToFun}\; R = \{\, x : \mathrm{dom}\, R \bullet x \mapsto R(\!|\{x\}|\!) \,\} \\
\hline
\end{array}$$

Appendix D

Supporting Traits

The `LCLAux` trait is meant to contain auxiliary definitions that are needed to support the definition of traits derived from LCL specifications.

⟨*LCLAux.lsl*⟩≡
```
LCLAux: trait
   includes
      ⟨LCLAux include⟩
```
Currently `LCLAux` contains only the definitions of `modAtMost` and `trashAtMost` which are defined in the trait `ModAndTrash` given in Section 8.6.4.2.6.

⟨*LCLAux include*⟩≡
```
ModAndTrash
```

Index

Index of Noweb Chunks

⟨*StorePOps.lsl* 95a⟩
⟨*Surjective.lsl* 109e⟩