Chapter 3
Algebraic Specification and Larch

3.1 Axioms, functions, and operations

In this chapter the family of algebraic specification languages will be considered, with Larch serving as the prime example. (In the Bibliographic References section at the end of the chapter, some of the other languages in this class are surveyed.)

Algebraic specification defines a data structure in terms of the operations executed, independent of any implementation or of any given notation for those operations. Instead, each operation is described in terms of its effect on the others. The operations are viewed as functions. The signatures of the functions describe the types of the arguments and the result. Relations among operations are presented in the form of equations, called axioms. These can be understood in several ways. One interpretation is as a statement that for some given state, if the operations on the left-hand side of the equation are executed, the resultant state is “equivalent” to that obtained by executing the operations on the right-hand side starting from the same state. Functional composition serves to order the execution of operations, with an outer function operating on the result of an inner one.

The classic example used to illustrate the algebraic method is a specification of a stack, with operations of push, pop, and top. If we
consider the functionality of the operations, \( \text{push} \) can be seen as a function from an existing stack and an input value to a new stack, while \( \text{pop} \) is a function from a stack to another stack (without the popped element). The key relationship is expressed by

\[
\text{pop}(\text{push}(s, i)) = s.
\]

This means that if an element \( i \) is \emph{pushed} onto a stack \( s \), and a \emph{pop} operation is executed on the result, then the original stack \( s \) is obtained. The last-in-first-out behavior of a stack is expressed by this axiom because the last element inserted, \( i \), is the one removed by the \emph{pop} operation. Similarly,

\[
\text{top}(\text{push}(s, i)) = i
\]

expresses that the last value inserted is the one to be returned by a \emph{top} operation.

Another way to view such a specification is as defining equivalence classes over an algebra of words, where the “words” are all of the possible combinations of operators and variables. Every combination is a well-formed expression and has a type that can be determined from the signatures of the operations.

A variable in an axiom represents any word whose result is of the same type as the variable. The first axiom above can be seen as a recursive definition, defining a more complex word that represents a stack (namely, \( \text{pop}(\text{push}(s, i)) \)) in terms of a simpler one (namely, \( s \)). To provide a base for the recursion, and allow listing the words that define a stack without using variables, a word that defines a stack without using \( s \) is needed. Here we will assume an additional initialization operation, called \emph{new}, that is a function with no arguments with a result of type \emph{stack}. A stack with two elements, say 5 and 7, could then correspond to the word \( \text{push}(\text{push}(\text{new}, 5), 7) \).

The surprising point of this approach is that the operations and the two axioms given are sufficient to completely define a stack. Moreover, use of the usual names for the operations is unimportant for the specification, although desirable as informal explanation to aid readability. If \( W, X, Y \), and \( Z \) had been used instead of \emph{push, pop, new and top,}
exactly the same operations and data structure would be defined by the axioms.

As an addition, we could indicate that new represents an empty stack, on which top and pop should not be attempted, by adding the axioms

\[ \text{top(new) = undefined} \]

and

\[ \text{pop(new) = undefined} \]

where undefined is a special constant used to denote error conditions. However, we shall see that these last two axioms are not really needed, since another approach to treating errors is more common in Larch.

Many questions are raised by this example. How do we know that the axioms are sufficient? In particular, why don’t we need one connecting top and pop directly? Can this simple example be expanded to more general specifications, and how natural is it to do so?

The Larch specification languages are an attempt to build a practical specification tool on the above ideas. Larch emphasizes the definition of new operations using the algebraic method in what is called the Larch Shared Language (also known as LSL or simply the Shared Language). These operations are then used to define a separate notation appropriate for each particular programming language, in interface languages. The core Larch Shared Language emphasizes redundancy in the specification, so that consistency checks can be made, and some guidelines on completeness can be given.

Most of the interface languages developed to date using Larch are notations for input-output assertions about the modules of the language, e.g., procedures. Basically, a Hoare logic is enriched with the operations and properties defined in the Shared Language. The use of algebraic specification in conjunction with a specification language for concurrency will be seen in the chapter on LOTOS. We shall emphasize LSL in the presentation here, and only briefly demonstrate the relation to interface languages.

The Larch notation and methodology does treat reactive systems, and new operations on data, in addition to pioneering ideas that encourage elegant and correct specification and design of systems.
3.2 Basic Elements of a Trait

The basic unit of LSL is the trait, which presents operators and described properties using axioms. Syntactically, a trait has an introduces section in which the operations and their functionality are listed, followed by an asserts section in which the algebraic axioms are given, along with additional assertions denoted by keywords to be explained later in the chapter.

Inside a trait, several properties may be grouped together for later use as part of more complex definitions. In addition, using Larch terminology, a new sort may be defined. The nature of such sorts and their close connection to data types will be seen when the interface languages are considered. The Larch trait stack that defines the sort $S$ is seen below. The lines that begin with % should be ignored for now.

stack: trait
includes Integer
introduces
  new: $\rightarrow S$
  push: $S, E \rightarrow S$
  top: $S \rightarrow E$
  pop: $S \rightarrow S$
  isEmpty: $S \rightarrow \text{Bool}$
asserts
% $S$ generated by new, push
$\forall e: E, \text{stk}: S$
  top(push(stk, e)) == e;
  pop(push(stk, e)) == stk;
  isEmpty(new) == true
  isEmpty(push(stk, e)) == false
% implies converts top, pop
% exempting top(new), pop(new)

The Larch version closely follows the general description of a stack given earlier, except for the trivial addition of an isEmpty operator. The Larch trait here does not treat error situations by using the value undefined, as described previously. Such a treatment is legal in Larch, but usually errors and required responses to them are left to the interface language, and are not considered in LSL traits.
3.2. BASIC ELEMENTS OF A TRAIT

A more complex specification can be seen in the trait `tablespec`:

```plaintext
tablespec: trait
  introduces
    new: → Table
    add: Table, Ind, Val → Table
    ∈: Ind, Table → Bool
  lookup: Table, Ind → Val
  isEmpty: Table → Bool
  size: Table → Int
  0, 1: → Int
  +: Int, Int → Int

asserts ∀ i, i1: Ind, val: Val, t: Table
  ¬(i ∈ new);
  i ∈ add(t, i1, val) ⇔ i = i1 ∨ i ∈ t;
  lookup(add(t, i, val), i1) ==
    if i = i1 then val else lookup(t, i1);
  size(new) == 0;
  size(add(t, i, val)) ==
    if i ∈ t then size(t) else size(t) + 1;
  isEmpty(t) == size(t) = 0
```

Here we see that the names of operators can be overloaded in different traits, so that `new` of `tablespec` will have the same general role as `new` of `stack`, but with different specific properties. The same will be true of `isEmpty`. In addition to the usual signature of functions, the specification also has an infix operator `∈`. This is merely for readability, so that the notation resembles familiar mathematics but in this case, the membership operation although its meaning is nonstandard.

Some superficial information about the operations can be obtained directly from the functionality. Only the first two operations, `new` and `add`, have a result of sort `Table`. Since the new sort `Table` is actually defined by this trait, these two operations are known as constructors of this sort. For the sort `S` defined in the `stack` trait, the constructors would be `push`, `pop`, and `new`.

The remaining operations of the trait examine the sort being defined, and provide information about it, in terms of other sorts. These operations are also known as observers. Of the four observer operations
of `tablespec`, two have a result of sort `Bool`, one has a result of sort `Int`, and one has a result of sort `Val`. For the `stack` trait, the observers are `top` and `isEmpty`.

The first two axioms of the `asserts` section define the `∈` operation. The first assertion is actually shorthand for

\[-(i \in \text{new}) = \text{true}.

In words, this asserts that for any `i` of sort `Ind`, and the `Table` obtained by executing `new`, the `∈` operation gives a result of `false`.

The second axiom relates `∈` to the result of applying `add` to arguments `t`, `j`, and `v`. The value of applying the `∈` will be `true` if either the `i` considered is equal to the `Ind` `j` just used for `add`, or if `i \in t` is `true`. Since `t` is also built up from `add` operations, the answer to this later question will recursively apply the two axioms until either a `true` result is obtained, or `t = \text{new}`, and a `false` result is obtained from the first axiom. This recursive definition will give the value `true` whenever the index tested has been used to add a value, and `false` otherwise.

The third axiom defines the relationship between `lookup` and `add`. If the index used for `lookup` is the same as the index just used to `add` a new value, the result will be the new value. Otherwise, the result is the same as that obtained by applying `lookup` to the `Table` before the most recent `add` operation, with the same index. Again, this can be used recursively on increasingly smaller words that represent a value of a `Table`, until the outermost `add` at index `j` is found. Note that if the index `j` does not appear in `t`, the actual value of `lookup(t, j)` cannot be obtained from this axiom. We will relate to this difficulty later on.

The next pair of axioms define `size` by relating it to `new` and `add`, respectively. Rather than having `size` represent the total number of values added to the structure, as we might do for a set, the operation here gives the number of different indices used in `add` operations. The recursive definition states that the size is not changed if a value is added to an index already used previously.

Finally, the `isEmpty` operation can be viewed as a derived predicate, using the `size` operation.
3.3 Initial or Final Algebras

Yet another view of a trait is as defining a \textit{theory}, namely a set of formulas (words) without free variables, in typed first-order logic with equality. The theory has

- all axioms of first-order logic with equality
- all assertions of the trait
- all logical consequences of the above

It is important to note that nothing else can be concluded about the theory of a trait. This answers the question of how should we relate to terms not explicitly connected by the axioms. Historically, two responses have been given to this question by researchers in algebraic specification.

The \textit{initial algebra} approach is that such terms are different. In this view, only what must be equivalent according to the axioms is considered equivalent. If two words cannot be proven to be equivalent from the theory, they must be different. For the \textit{tablespec} example, it is intuitively clear that if the same value is added at index $i$, and then at index $j$, the result is the same as if those operations had been done in the opposite order. But there is no way to prove this directly from the axioms given in \textit{tablespec}. Thus in the initial algebra approach the two orders of execution must be considered significant, and this is equivalent to the statement

$$\text{add}(\text{add}(t, i, v), j, v) \neq \text{add}(\text{add}(t, j, v), i, v)$$

On the other hand, the \textit{final algebra} approach would identify such terms as equivalent, as long as no explicit axiom to the contrary is given. That is, whatever will not contradict anything in the axioms can be considered equivalent. Under this assumption,

$$\text{add}(\text{add}(t, i, v), j, v) = \text{add}(\text{add}(t, j, v), i, v)$$

follows from \textit{tablespec}, because it does not contradict anything there.
Consider the assertion
\[
\text{add}(\text{add}(t, i, v), j, u) = \text{add}(\text{add}(t, j, u), i, v)
\]
This differs from the previous equation in that two values \(u\) and \(v\) are being added. Thus the assertion is that the order of adding two different elements is irrelevant, and that the final Table is the same. This would not be correct for the theory defined by \text{tablespec}, even if the final algebra approach were to be taken: if the indices \(i\) and \(j\) happen to be the same, the result of applying \text{eval} to the two values would be different. If, however, we assume that \(i \neq j\), then the above assertion would hold for a final algebra interpretation but not for an initial algebra view.

Larch adopts neither the initial nor the final algebra view. If nothing is explicitly declared about how to interpret the axioms, neither equality nor inequality could be concluded for the examples above. The reason for this design decision about the Shared Language is to encourage modularity. Only this assumption guarantees that all theorems proven about an incomplete specification remain valid when it is extended with additional axioms or operations. The additional axioms could make more words equivalent, so an earlier conclusion that they were distinct would now be invalid. Similarly, two words identified under a final algebra approach might now be distinguished by the axioms. So it indeed is best to reach no conclusion about such values. However, there are situations requiring more powerful theories that do allow concluding equivalence or nonequivalence implicitly. Larch provides extra facilities for creating such theories.

### 3.4 Additional features in LSL

In addition to the functionality of the operations, and the equational axioms, LSL provides three types of additional features:

- Creating stronger theories, using \text{generated by} and \text{partitioned by}. These allow concluding additional theorems, using assumptions related to those of initial or final algebras.
- Combining theories, using \text{includes} and renaming conventions.
• Checking consistency, using terms such as implies, converts, exempting, or assumes.

The first addition to the Shared Language has the form

\[
\text{sort generated by operator-list}
\]

The generated by clause means that every value of the named sort (represented by a word with no variables of that sort) is equivalent to a word that only contains the operators in the list. Another way to say the same thing is that every value of the sort can be obtained by composing only operators in the list. For the trait `stack`, it is reasonable to add the assertion

\[
\text{S generated by new, push}
\]

to the other parts of the asserts clause. This conforms to our intuition that every stack can be obtained by creating a new one, and putting in the appropriate values, and that the final stack state obtained by a long succession of push-es and pop-s is equivalent to the stack state obtained just by push-ing in the remaining values.

A generated by clause defines an induction rule for the sort in the clause: to prove a property for values of the sort, use induction only on the words generated by the operators in the list of the clause. For example, in `tablespec`, it is reasonable to add

\[
\text{Table generated by new, add}
\]

and then use that to prove

\[
\forall t:\text{Table}, i:\text{Ind} \quad ((i \in t) \Rightarrow (\text{size}(t) > 0))
\]

The base of the induction uses new in place of \( t \), and it is easy to see from the other axioms that the left side of the implication is then false for any \( i \), making the assertion true. The inductive case assumes the assertion for \( t \), and then must prove it true when \( \text{add}(t, j, v) \) is substituted for \( t \) in the assertion.

The generated by clause defines a subclass of the constructors, namely the generators. The rest of the constructors are called extensions. So pop in the `stack` trait is an extension. When a trait is used to define a sort, it often is sufficient to provide axioms that apply each observer
(e.g., top) or extension (e.g., pop) to each generator (push and new). Exceptions to this guideline will be shown later.

The second strengthening of a theory can be done using the clause

\texttt{sort partitioned by operator-list}

This means that two values of \texttt{sort} will be different only if different results are obtained when some combination of operators from the list is applied to each of the values. Again using the correspondence between words and configurations, if two words are not equivalent, then that fact can be seen by applying only the operators in the list to the words. Note that this means that if the words can \textit{not} be distinguished by applying the operators in the list, they must be equivalent. Thus something approaching the final algebra approach is being taken, where a deduction of equivalence can be made because words cannot be distinguished by existing axioms.

Consider the trait \texttt{settrait} in Figure 3.1 that defines classic mathematical sets in the sort \texttt{Set} (and is analyzed in the Exercises).

We can add to the \texttt{asserts} section the clause

\texttt{Set partitioned by \in}

where \(\in\) is the membership operation of sets. Such an assertion is justified because if the elements are the same for two finite sets, so are the sets, no matter in what order the elements were inserted. If sets differ, there is some element in one that is not in the other, and applying the membership operation for that element to each set will give different results.

We can also add to the assertions of \texttt{tablespec}

\texttt{Table partitioned by \in, lookup}

Now the reasonable equalities mentioned earlier can be proven by deduction. For example, the same results are obtained by applying any \(\in\) or \texttt{lookup} operations to \texttt{add}((add(t, i, v), j, v)) and to \texttt{add}((add(t, j, v), i, v)), so they can be proven equivalent. This clause really does strengthen a Larch trait, since otherwise, nothing can be concluded about words not provably equivalent or provably nonequivalent using direct application of the equalities.

The next type of addition to Larch traits allows defining operators separately, to encourage modularity and reuse. The clause
settrait: trait
introduces
\{\}\: \rightarrow \text{Set}
_ \in _: \text{E}, \text{Set} \rightarrow \text{Bool}
insert: \text{E}, \text{Set} \rightarrow \text{Set}
delete: \text{E}, \text{Set} \rightarrow \text{Set}
size: \text{Set} \rightarrow \text{Int}
_ \cup _: \text{Set}, \text{Set} \rightarrow \text{Set}
_ \cap _: \text{Set}, \text{Set} \rightarrow \text{Set}

asserts
\forall \text{e, e1: E, s, s1: S}
\neg (\text{e} \in \{\});
\text{e} \in \text{insert(e1, s)} = \text{e} = \text{e1} \lor \text{e} \in \text{s};
\text{size(\{\})} = 0;
\text{size(insert(e, s))} =
\text{if e} \in \text{s} \text{ then size(s) else size(s) + 1};
\text{delete(e, \{\})} = \{\};
\text{delete(e, insert(e1, s))} =
\text{if e} = \text{e1} \text{ then delete(e, s) else insert(e1, delete(e, s))};
\text{s} \cup \{\} = \text{s};
\text{s} \cup \text{insert(e, s1)} = \text{insert(e, s} \cup \text{s1});
\text{s} \cap \{\} = \{\};
\text{s} \cap \text{insert(e, s1)} =
\text{if e} \in \text{s} \text{ then insert(e, s} \cap \text{s1) else s} \cap \text{s1};

Figure 3.1: A definition of sets
reflexive: trait
  introduces \texttt{\_\_ \_ \_ \_ \_ \_ \_}: T, T \rightarrow \text{Bool}
  asserts \forall x: T
  \hspace{1cm} x \leftrightarrow x

transitive: trait
  introduces \texttt{\_\_ \_ \_ \_ \_ \_ \_}: T, T \rightarrow \text{Bool}
  asserts \forall x, y, z: T
  \hspace{1cm} (x \leftrightarrow y \land y \leftrightarrow z) \Rightarrow x \leftrightarrow z

symmetric: trait
  introduces \texttt{\_\_ \_ \_ \_ \_ \_ \_}: T, T \rightarrow \text{Bool}
  asserts \forall x, y: T
  \hspace{1cm} x \leftrightarrow y \equiv y \leftrightarrow x

\begin{figure}[h]
\begin{center}
\begin{tabular}{l}
\text{Equiv: trait}
\begin{tabular}{l}
includes \text{trait}
\end{tabular}
\end{tabular}
\end{center}
\end{figure}

Figure 3.2: Some properties of $\leftrightarrow$

includes \text{trait}

means that the theory will be the union of the trait included and the one containing the clause. In fact we have been using operators from other traits, even in the simple traits used here as examples. The +, 0 and 1, used in defining \textit{tablespec} are really from the trait \textit{Integer}. Larch implicitly assumes that \textit{Integer} and \textit{Boolean} traits are included in every trait where the usual operators and constants of those traits are used.

The three trivial traits in Figure 3.2 express the properties of reflexivity, transitivity, and symmetry. They show that not every trait defines a data type. A new trait can be built from those in the figure merely by writing

\begin{verbatim}
Equiv: trait
includes reflexive, transitive, symmetric
\end{verbatim}

Naturally, it is unreasonable to assume that the same names and operation symbols will be used in all traits that we may wish to later include together as defining some properties in a more complex trait. It is therefore possible to \textit{rename} any sort or operator of an included trait by adding a list
where *new name* is the desired new terminology for the original *old name* in the included trait.

So if *symmetric* had been defined using $S$ rather than $\leftrightarrow$, e.g., as,

```plaintext
symmetric: trait
  introduces _S_ : T, T \rightarrow Bool
  asserts \forall x, y: T
     x S y \iff y S x
```

then we still could use it to define *Equiv* with a more standard equivalence notation, by writing

```plaintext
includes reflexive($\equiv$ for $\rightarrow$), transitive($\equiv$ for $\rightarrow$),
symmetric($\equiv$ for $S$)
```

It is also possible to rename just some of the operations or sorts. For example, to obtain more standard terminology, we might define

```plaintext
Sparse: trait
includes tablespec(Array for Table, _[ ]_ for lookup)
```

When it is likely that some names will commonly be renamed when a trait is included in other traits, a formal parameter notation can be used. That is, when the trait is defined, list the likely names as formal parameters of the trait. Then those can be renamed by listing the actual new name when the trait is included, without the “for”. If we had defined *tablespec* as

```plaintext
tablespec(Ind, Val): trait
```

then a more particular theory could have

```plaintext
includes tablespec(Integer, Real)
```

Similarly, it would be reasonable to define *Equiv*, and then perhaps include *Equiv* in one trait and *Equiv* in another.

The remainder of LSL provides facilities for extra checks and implications of a theory, to aid in debugging the specification. Already built into every trait is an implicit claim of *consistency*. That is, the theory defined by the trait cannot ever allow proving *true* = *false*. The
specification checker tool to be discussed below will attempt to determine consistency, but this question is undecidable for most nontrivial theories, such as those including the integers. It is thus an example of a specification requirement (of the Larch system itself) that we cannot fully implement.

For any equation or other LSL clause \( P \),

\[ \text{implies } P \]

can be added to the theory. This does not in any way change the theory defined, but merely claims that \( P \) is a theorem of the already defined theory. Such claims are useful primarily to show that the trait corresponds to the desired intuition, and that the intended object is defined with the desired properties. Of course, once such an \text{implies} statement has been shown to be true, the claim can be used to aid in reasoning about the trait, when it is included in more complex contexts. When a Shared Language specification with an \text{implies} clause is checked by the tools, a new theorem is created, and an automatic proof that \( P \) follows from the existing theory is attempted. Failures in such proofs can be valuable for detecting errors in the specification.

Among the allowable assertions in an implication, is a statement

\[ \text{implies converts operation-list} \]

This is only used as an additional check, and means that every operation in the list, when applied to a legal argument, has only one interpretation that satisfies the axioms, assuming that the other operations have fixed interpretations. Intuitively, the operations are sufficiently defined. For example,

\[ \text{implies converts isEmpty} \]

for the \text{tablespec} trait would mean that \text{isEmpty} applied to a word of sort \text{Table} is always equivalent to one of the constant values of its result, namely \text{true} or \text{false}.

It might seem that \text{lookup} could also be added to the list of a \text{converts} assertion, but as already noted, \text{lookup(new, i)} is not equivalent to any word not involving \text{lookup}, since there is no axiom relating \text{lookup} and \text{new}. One way to solve the problem would be to add an axiom
lookup(new,i) = undefined

However, as already noted, error situations are more commonly treated in the interface languages, without relating to such values in the Shared Language. By adding a phrase

\texttt{exempting } \forall i: \text{Ind} \ \texttt{lookup(new,i)}

terms of the form \texttt{lookup(new,i)} can be left as they are, without being equivalent to any shorter word or constant, and \texttt{lookup} only has to be uniquely defined relative to some interpretation of those exempted words.

Similarly, in the \textit{stack} trait, it is natural to write in the \texttt{implies} section

\begin{verbatim}
converts isEmpty, top, pop
  exempting top(new), pop(new)
\end{verbatim}

The keyword \texttt{assumes} of the Larch Shared Language is the last we consider that requires checking when a specification is submitted to the associated tools. Within a trait B the statement \texttt{assumes A}

may appear. This statement asserts that the trait A will be defined in any context where the including trait B is itself included or assumed. For the purposes of the analysis of B itself, the theory of B has that of A in it, just as if A appeared in an \texttt{includes} statement. As an example of its use, we could have a trait \texttt{totalorder} that contains \texttt{assumes Equiv}

When a higher level trait, say, \texttt{Sortintegers}, contains \texttt{includes totalorder}

the \texttt{assumes} clause implicitly asserts that \textit{Equiv} is already defined in that context, and there then is an obligation to check that it is. The only difference between \texttt{assumes} and \texttt{includes} is in the additional assertion that the context already defines a theory that implies the assumed trait.
3.5 Style in algebraic specifications

All of the essential features of the Larch Shared Language have now been presented, although there are several additional syntactic conveniences that are not given here, for the sake of brevity. Beyond simple understanding of the components in the language, there seem to be distinctive styles for writing LSL traits. As in any language, there are recurrent paradigms in algebraic specifications that are the idioms of the language. Once these are recognized, understanding and writing such specifications becomes far easier.

The language encourages building up definitions in stages, identifying possible intermediate traits that may be reused in several contexts, and adding properties only as required for more complex traits. In Figure 3.3 three traits are defined, InsertGenerated, Container, and Queue. Variations of these traits have appeared in several papers on Larch. InsertGenerated is a minimalistic trait that is part of any sort that contains elements, and is built up by creating an empty structure, and somehow inserting elements. How such insertions occur, and the properties of the insert operation are completely open at this stage, except that values of C can be built up by using only empty and insert operations.

This is then used to define more complex structures. The Container trait includes InsertGenerated, but also defines isEnc, count, ε, and len. What is still missing is how elements in the structure can be deleted or examined, and particularly the relation between insert and these other operations. In the Queue trait, a FIFO queue is defined by adding particular remove and look operations. If these were defined by different axioms, stacks and other ordered data structures could be defined.

Traits similar to these and many other common traits appear in the Larch Handbook. This is a large collection of hopefully useful traits. Most standard data structures are included, as are properties such as equivalence and total order. Thus most traits that describe systems can build on these standard units, defining only the new components that differ from standard terminology.
3.5. **STYLE IN ALGEBRAIC SPECIFICATIONS**

InsertGenerated \((E, C)\): trait

% \(C\)'s contain finitely many \(E\)'s

introduces

- \(\text{empty} : \rightarrow C\)
- \(\text{insert} : E, C \rightarrow C\)

asserts

\(C\) generated by \(\text{empty}\), \(\text{insert}\)

Container \((E, C)\): trait

% enumerating contents of a \(C\)

includes InsertGenerated, Integer

introduces

- \(\text{isEmpty} : C \rightarrow \text{Bool}\)
- \(\text{count} : E, C \rightarrow \text{Int}\)
- \(\_ \in \_ : E, C \rightarrow \text{Bool}\)
- \(\text{len} : C \rightarrow \text{Int}\)

asserts

forall \(e, e_1 : E, c : C\)

- \(\text{isEmpty}(\text{empty})\);
- \(\neg\text{isEmpty}(\text{insert}(e, c))\);
- \(\text{count}(e, \text{empty}) == 0\);
- \(\text{count}(e, \text{insert}(e_1, c)) ==\)
  \(\text{count}(e, c) + (\text{if } e = e_1 \text{ then } 1 \text{ else } 0)\);
- \(e \in c == \text{count}(e, c) > 0\);
- \(\text{len}(\text{empty}) == 0\);
- \(\text{len}(\text{insert}(e, c)) == \text{len}(c) + 1\);

implies

converts \(\text{isEmpty}\), \(\text{count}\), \(\_ \in \_\)

Queue \((E, C)\): trait

includes Container

introduces

- \(\text{remove} : C \rightarrow C\)
- \(\text{look} : C \rightarrow E\)

asserts

\(C\) partitioned by \(\text{isEmpty}\), \(\text{look}\), \(\text{remove}\)

\(\text{look}(\text{insert}(e, c)) ==\)

- \(\text{if } \text{isEmpty}(c) \text{ then } e \text{ else } \text{look}(c)\)

\(\text{remove}(\text{insert}(e, c)) ==\)

- \(\text{if } \text{isEmpty}(c) \text{ then } \text{empty else } \text{insert}(e, \text{remove}(c))\)

\(\neg\text{isEmpty}(c) \Rightarrow\)

\(\text{count}(e, \text{insert}(\text{look}(c), \text{remove}(c))) == \text{count}(e, c)\)

implies

converts \(\text{look}\), \(\text{remove}\), \(\text{len}\)

exempting \(\text{look}(\text{empty})\), \(\text{remove}(\text{empty})\)

---

Figure 3.3: Containers and a Queue
3.6 Interface specifications

As already mentioned, LSL is used to define traits, operators, and sorts that are then used in specifying modules of systems. The actual module is specified in a Larch Interface Language. The idea is that each implementation language has its own terminology—its data types, keywords, error handling facilities, and modularization concepts. The interface language inherits all keywords of the implementation language, along with their semantics. Assuming that the semantics of the programming language are or can be formally defined, the interface language saves having to repeat the effort, and simply uses the result. Moreover, all terms from traits defined in LSL are available for use in specifications of the particular Larch Interface Language.

The interface languages defined to date by the researchers in Larch have mainly been state based, relating to the state before a module is activated and the one after it terminates. They closely resemble the input/output specifications used by Hoare, with the added flexibility of terminology that can be expanded using the Shared Language. Rather than being algebraic, pre- and post-conditions are used. Among the interface languages to date are those for Pascal, CLU, and C++.

This separation of the algebraic stage from the input/output stage has several advantages. In the algebraic part, variables are logical and do not ever change values, while in the interface part, program variables have values before and after activation of a module. Keeping these views separate can avoid confusion.

The use of terminology from LSL is not restricted in either direction: some operators and sorts defined in LSL might not appear in interface specifications, and even those that are used may not be implemented as procedures or other modules, unless that is the particular requirement of the interface specification. In other algebraic approaches, there is a distinction between the operators intended for implementation and the “hidden” operators used only to help define the others. Here this issue is handled automatically by the separation of shared language traits that never have to be implemented as such, and the requirements of the interface language. In one interface specification a particular subset of the operators from a trait may be used, while in another, a second subset appears.
As already noted, treatment of errors is usually deferred to the interface specification. The particular facilities of the implementation language can then be exploited in defining the requirements. One language may have stronger type checking, another, more convenient interrupt and recovery possibilities. Moreover, particular limitations of resources such as memory may be relevant only temporarily, or in a particular environment. Such limitations are both difficult to treat and irrelevant in the algebraic definition. The LSL traits should be reusable in a variety of concrete systems, with differing error conditions.

Since the interface languages are particular to the appropriate programming language, and have many details irrelevant to the general principles involved, we only briefly describe a typical, Pascal-like interface specification language. The interface relates to procedures, and has the form

\begin{verbatim}
uses trait
procedure header
requires P
modifies L
ensures Q
\end{verbatim}

where $P$ is the usual precondition of an input/output assertion, in terms of formal parameters from the header, $L$ is a list of the changeable objects, and $Q$ is a usual postcondition, relating final values of some parameters to initial values. The final values, when the procedure terminates, are denoted using a \('\) after the variable name, while the unprimed version relates to the value when the procedure is called. The name of a function is used as a (primed) variable to relate to its final value.

Restrictions on input appear in $P$, while error conditions appear in $Q$. If the intent is to forbid calling the procedure with illegal values, the restriction should appear in $P$, while if the procedure may be called with those values, but an exceptional computation, such as printing an error message, is the result, then this should be specified in $Q$.

A collection of procedures related to the \texttt{set} trait defined earlier is shown in Figure 3.4. The first procedure initializes a variable $s$ of type \texttt{set}, and thus corresponds to the \{\} operation. The second seems to correspond to the \texttt{insert} operation, as long as the result will have no more
uses settrait

procedure setinit( var s: Set )
  modifies s
  ensures s’ = {}

procedure setinsert( var s: Set; e: integer )
  requires size( insert( e, s ) ) \leq 100
  modifies s
  ensures s’ = insert(e, s)

procedure setrem( var s: Set; var f: Bool; e: integer )
  modifies s, f
  ensures s’ = delete( e, s ) \land f’ = ( e \in s )

function choose( s: Set; var e: integer ): boolean
  modifies e, choose
  ensures if size(s) > 0 then (choose’ \land (e’ \in s))
  else (¬choose’ \land (e’ = e))

Figure 3.4: Interface specifications for set-related procedures
than 100 elements. On the other hand, the third procedure combines delete with a membership function, while choose does not correspond to any operation of the trait (and chooses an arbitrary element of the set). In other interface specifications it is even more striking that there need be no direct correspondence between the operations of the traits used and the procedures and functions of the implementation. The operations may be used indirectly to define the requirements of the modules, and never need to be implemented themselves.

### 3.7 Correctness of implementations

To complete the picture of Larch specifications and their connection to implementations, we must show what exactly is asserted by an interface specification of a procedure, and how we may determine whether the code in a procedure satisfies the specification. To do this, we need to connect the state of the implementation to values of sorts used in the requirements. We then will be able to use axioms from the definition of the sort in LSL traits, to help show the ensures clause correct after executing the code in the procedure.

However, implementations of Larch specifications are generally not formally verified. As noted in the Introduction, verification is only one possible benefit of specification, and often it is too demanding in resources to be practical. The automatic tools of the support system for Larch also do not directly support verification, although the theorem prover could be used to help in correctness proofs of implementations. The explanation below is thus intended to show the semantics of Larch—what exactly is being asserted. General techniques for verification of implementations with loops, and proper treatment of parameters, is beyond the scope of this book. Courses and books on correctness and verification should be consulted to receive a full picture of this aspect.

The connection between the state of the implementation and the abstract values of sorts (which can be viewed as the state of the specification) is made using a mapping function and an invariant. The invariant is established by an initialization, either in a procedure or the main program. Thereafter, it is assumed true before each procedure, and must be shown to hold for the values when the procedure is com-
completed. Consider an interface specification for a procedure \texttt{pcode} where a predicate \texttt{reqs} represents the input specification, \texttt{ensures} denotes the output specification, \texttt{impstate} denotes the state of the implementation, \texttt{I} is the invariant, and \texttt{M} is the mapping function. The specification is equivalent to the Hoare logic assertion
\[
\{I \land \texttt{reqs}(M(\texttt{impstate}))\} \texttt{pcode} \{I \land \texttt{ensures}(M(\texttt{impstate}))\}
\]
plus asserting termination of the procedure and that only variables in the \texttt{modifies} list are changed. It follows that \texttt{reqs} and \texttt{ensures} are assertions about the abstract specification states that are the results of applying the mapping function \texttt{M} to the implementation states before and after executing the implementation \texttt{pcode}.

The invariant \texttt{I} and mapping function \texttt{M} cannot in general be found automatically. They require understanding the key properties of the implementation. Consider the procedure \texttt{popimp} intended to implement a \texttt{pop} operation of a stack. The interface specification might have

```plaintext
procedure popimp(var st:stack);
  requires \rightarrow \texttt{isEmpty}(st)
  modifies st
  ensures st' = \texttt{pop}(st)
```

In terms of input/output assertions, this could be written as:

\[
\{I \land \neg\texttt{empty}(st)\} \texttt{popimp} \{I' \land st' = \texttt{pop}(st)\}
\]

A typical implementation of a stack might have an array \texttt{a} and an index \texttt{i} indicating the top element in the stack. To add a value, we would increase \texttt{i} by one, and put the value in \texttt{a[i]}. To \texttt{pop} the stack, we therefore need only decrease the index \texttt{i} by one (i.e., the code \texttt{i := i - 1}). The stack represented by a particular array value and index has the value in \texttt{a[1]} as the innermost element in the stack, the value in \texttt{a[2]} next, up to the top value, which is in \texttt{a[i]}. In the abstract sort \texttt{stack}, a value is an equivalence class of words. The mapping function \texttt{M} is thus from the concrete state \texttt{a} and \texttt{i}, to the abstract stack \texttt{st}, represented by \texttt{new}, and a series of \texttt{push}-es. The discussion above leads us to the recursive definition:

\[
M(a, 0) = \texttt{new}
\]
To make the mapping complete, we need to have that \( i \geq 0 \land \text{int}(i) \). This is the needed invariant (where \( \text{int} \) denotes that \( i \) is an integer). In terms of input/output assertions, using that \( st = M(a, i) \) and \( st' = M(a', i') \), we require:

\[
\{ i \geq 0 \land \text{int}(i) \land \neg\text{empty}(M(a, i)) \}
\]

\[
\{ i' \geq 0 \land \text{int}(i') \land M(a', i') = \text{pop}(M(a, i)) \}
\]

From the precondition, \( \text{empty} \) is not true for \( M(a, i) \), so it must have the form \( \text{push}(M(a, i - 1), a[i]) \). Otherwise, from the recursive definition, \( M(a, i) \) must equal \( \text{new} \), and in that case \( \text{empty} \) is true. Substituting for \( M(a, i) \) in the post-condition, we see that we require

\[
M(a', i') = \text{pop}(\text{push}(M(a, i - 1), a[i]))
\]

Now we may use the key axiom \( \text{pop}(\text{push}(s, e)) = s \), with \( M(a, i - 1) \) for \( s \). The remaining requirement is simply

\[
M(a', i') = M(a, i - 1).
\]

Since the intended code in \( \text{popimp} \) is \( i := i - 1 \), the final value of \( i \) is indeed one less than the initial value (i.e., \( i' = i + 1 \)), and \( a' \) equals \( a \). Thus the above equation will be true for the implementation, and we have established the specification. A similar proof for the implementation of \( \text{pushimp} \) is also easy to establish. Of course, when the implementation code is more complex, such a proof can be much more difficult.

### 3.8 Bibliographic remarks

Algebraic specification has a long history, both as a theoretical underpinning for the semantics of languages and data types, and as a specification method. Some of the basic ideas were suggested in [10]. The theoretical basis can be seen, for example, in the work of the ADJ
group [5, 6] and in [7]. The goal of much of that work was to provide a mathematical semantics for data structures and types in programming languages.

Early specification languages based on algebraic techniques include CLEAR[1] and OBJ[4]. The Larch language evolved from more theoretical work in the mid-70’s, and has since undergone significant revision and development [9, 8]. Other algebraic specification languages include ACT-ONE[2, 3], which serves as the algebraic level of the LOTOS specification method, and ASL [11].

The idea of dividing a specification into a Shared language part and an Interface part more oriented to the implementation is due to... Interface languages for object-oriented technologies have been developed by....

The Larch language has an active Web site, at www.larch.whereever.
Bibliography


