REPORT ON THE LARCH SHARED LANGUAGE*

J.V. GUTTAG
MIT Laboratory for Computer Science, Cambridge, MA 02139, U.S.A

J.J. HORNING
Digital Equipment Corporation, Systems Research Center, Palo Alto, CA 94301, U.S.A.

Communicated by M. Sintzoff
Received July 1984
Revised March 1985

Abstract. Each member of the Larch family of formal specification languages has a component derived from a programming language and another component common to all programming languages. We call the former interface languages, and the latter the Larch Shared Language.

This paper presents version 1.1 of the Larch Shared Language. It has two major sections. The first part starts with a brief introduction to the Larch Project and the Larch family of languages, and continues with an informal presentation of most of the features of the Larch Shared Language. It concludes with a brief discussion of how we expect Larch Shared Language Specifications to be used, a discussion of some of the more important design decisions, and a summary of the current status of the Larch project. The second part of this paper is a reference manual. A companion paper includes an extensive set of examples.

Introduction

0. The Larch family of languages

The Larch Project is developing tools and techniques intended to aid in the productive use of formal specifications of systems containing computer programs. Many of its premises and goals are discussed in [10]. A major component of the Larch Project is a family of specification languages. Each Larch language has a component particular to a specific programming language and another component common to all programming languages. We call the former interface languages, and the latter the shared language.

We use the interface languages to specify program modules. A specification of a program module should provide the information one needs to write programs that use the module. Specifications of the interface that one module presents to other

* This work was supported at MIT's Laboratory for Computer Science by DARPA under contract N00014-75-C-0661, and by the National Science Foundation under Grant MCS-8119846, by the Digital Equipment Corporation Systems Research Center, and at the Xerox Palo Alto Research Center by the Computer Science Laboratory.
modules often rely on notions specific to the programming language, e.g., its denotable values or its exception handling mechanisms. Thus the specifications are programming language dependent. Each interface language deals with what can be observed about the behavior of programs written in a specific programming language. Its simplicity or complexity is a direct consequence of the simplicity or complexity of the observable state and state transformations of that programming language.

The shared language is primarily algebraic. It is used to specify abstractions that are independent of both the program state and the programming language. The central syntactic domains of the shared language are operators and sorts. It is important not to confuse these with the domains procedure and type that are likely to occur in most interface languages. The operators and sorts of a shared language specification appear in specifications written in the interface languages, and in reasoning about such specifications, but they do not denote procedures or other objects that may appear in programs.

In some respects, the style of specification used in Larch resembles that used in operational specifications built upon abstract models [1]. It differs, however, in several important respects. The shared language is used to specify a theory rather than a model, and the interface languages are built around predicate calculus rather than around an operational notation. One consequence of these differences is that Larch specifications never exhibit 'implementation bias' [11].

Some important aspects of the Larch family of specification languages are:
- **Composability of specifications.** We emphasize the incremental construction of specifications from other specifications. The importance of such mechanisms is discussed in [2].
- **Emphasis on presentation.** Reading specifications is an important activity. To assist in this process, we use composition mechanisms defined as operations on specifications, rather than on theories or models.
- **Interactive and integrated with tools.** The Larch languages are designed for interactive use. They are intended to facilitate the interactive construction and incremental checking of specifications. The decision to rely heavily on support tools has influenced our language design in many ways.
- **Semantic checking.** It is all too easy to write specifications with surprising implications. We would like many such specifications to be detectably ill-formed. Extensive checking while specifications are being constructed is an important aspect of our approach. Larch was designed to be used with a powerful theorem prover for semantic checking to supplement the syntactic checks commonly defined for specification languages. We have been influenced here by our experience with Affirm [14].
- **Programming language dependencies localized.** We feel that it is important to incorporate programming-language-dependent features into our specification languages, but to isolate this aspect of specifications as much as possible. This prompted us to design a single shared language that could be incorporated into different interface languages in a uniform way.
- **Shared language based on equations.** The shared language has a simple semantic basis taken from algebra. Because of the emphasis on composability, checkability and interaction, however, it differs substantially from the 'algebraic' specification languages we have used in the past.
- **Interface languages based on predicate calculus.** Each interface language is based on assertions written in typed first-order predicate calculus with equality, and incorporates programming-language-specific features to deal with constructs such as side effects, exception handling, and iterators. Equality over terms is defined in the shared language; this provides the link between the two parts of a specification.

1. **Simple algebraic specifications**

Most of the constructs in the Larch Shared Language are designed to assist in structuring specifications, for both reading and writing. The *trait* is our basic module of specification. Consider the following specification for tables that store values in indexed places:

```plaintext
TableSpec: trait
  introduces
  new: → Table
  add: Table, Index, Val→ Table
  #: #: Index, Table→ Bool
  eval: Table, Index→ Val
  isEmpty: Table→ Bool
  size: Table→ Card
  constrains new, add, ∈, eval, isEmpty, size so that
  for all [ind, ind1; Index, val: Val, t: Table]
    eval(add(t, ind, val), ind1) = if ind = ind1 then val else eval(t, ind1)
    ind ∈ new = false
    ind ∈ add(t, ind1, val) = (ind = ind1) | (ind ∈ t)
    size(new) = 0
    size(add(t, ind, val)) = if ind ∈ t then size(t) else size(t) + 1
    isEmpty(t) = (size(t) = 0)
```

This example is similar to a conventional algebraic specification in the style of [8]. The part of the specification following *introduces* declares a set of *operators* (function identifiers), each with its *signature* (the sorts of its domain and range). These signatures are used to sort-check *terms* (expressions) in much the same way as function calls are type-checked in programming languages. The remainder of the specification constrains the operators by writing equations that relate sort-correct terms containing them.

There are two things (aside from syntactic amenities) that distinguish this specification from a specification written in our earlier algebraic specification
languages:

A name, TableSpec, is associated with the trait itself.

The axioms are preceded by a constrains list.

The name of a trait is logically unrelated to any of the names appearing within it. In particular, we do not use sort identifiers to name units of specification. A trait need not correspond to a single ‘abstract data type’, and often does not.

The constrains list contains all of the operators that the immediately following axioms are intended to constrain. It is the responsibility of a specification checker to ensure that the specification conforms to this intent. The constrained operators will generally be a proper subset of the operators appearing in the axioms. In this example the constrains list informs us that the axioms are not to put any constraints on the properties of if then else, false, 0, 1, +, |, and =, despite their occurrence in the axioms. The judicious use of constrains lists is an important step in modularizing specifications.

We associate a theory with every trait. A theory is a set of well-formed formulas (wff’s) of typed first-order predicate calculus with equations as atomic formulas.

The theory, call it Th, associated with a trait written in the Larch Shared Language is defined by

- Axioms: Each equation, universally quantified by the variable declarations of the containing constrains clause, is in Th.

- Inequation: \( \neg(\text{true} = \text{false}) \) is in Th. All other inequations in Th are derivable from this one and the meaning of =.

- First-order predicate calculus with equality: Th contains the axioms of conventional typed first-order predicate calculus with equality and is closed under its rules of inference.

The equations and inequations in Th are derivable from the presence of axioms in the trait—never from their absence. Th is deliberately small, because it is important to prove theorems before a specification is complete, and we wanted to keep the addition of new operators and equations from invalidating previously proved theorems.

2. Getting richer theories

While the relatively small theory described above is often a useful one to associate with a set of axioms, there are times when a larger theory is needed, e.g., when specifying an ‘abstract data type’. Generated by and partitioned by give different ways of specifying larger theories.

Section 1 does not include an induction schema. This is an appropriate limitation when the set of generators for a sort is incomplete. Saying that sort S is generated by a set of operators, Ops, asserts that each term of sort S is equal to a term whose outermost operator is in Ops. One might, for example, say that the natural numbers
are generated by 0 and successor and the integers generated by 0, successor, and predecessor. Generated by adds an inductive rule of inference.

The clause Table generated by [new, add] can be used to derive theorems such as

$$\forall t: \text{Table }[(t = \text{new}) | (\exists \text{ind: Index } [\text{ind} \in t])]$$

that would otherwise not be in the theory.

Section 1 allows equations to be derived only by direct equational substitution, not by the absence of inequations. This is an appropriate limitation when the set of observers for a sort is incomplete. Saying that sort S is partitioned by a set of operators, Ops, asserts that if two terms of sort S are unequal, a difference can be observed using an operator in Ops. Therefore, they must be equal if they cannot be distinguished using any of the operators in Ops. This adds new equations to the theory associated with a trait, thus reducing the number of equivalence classes in the equality relation.

The clause Table partitioned by [\epsilon, \text{eval}] can be used to derive theorems such as

$$\text{add}(\text{add}(t, \text{ind}, v), \text{ind1}, v) = \text{add}(\text{add}(t, \text{ind1}, v), \text{ind}, v)$$

that would otherwise not be in the theory.

3. Combining independent traits

Our example contains a number of totally unconstrained operators, e.g., false and +. Such traits are not very useful. A straightforward thing to do is to augment the specification with additional clauses dealing with these operators. One way to do this is by trait importation. We might add to trait TableSpec:

imports Cardinal, Boolean

The theory associated with the importing trait is the theory associated with the union of all of the introduces and constrains clauses of the trait body and the imported traits.

Importation is used both to structure specifications to make them easier to read and to introduce extra checking. Operators appearing in imported traits may not be constrained in either the importing trait or in any other imported trait. This guarantees that imported traits do not 'interfere' with one another in unexpected ways. I.e., it guarantees that the theory associated with a trait is a conservative extension of each of the theories associated with its imported traits. (An extension, Th1, of a theory, Th2, is conservative if and only if every wff of the language of Th2 which is in Th1 is also in Th2.) Each imported trait can, therefore, be fully understood independently of the context into which it is imported.

As a syntactic amenity, trait Boolean is automatically imported into all other traits.
4. Combining interacting traits

While the modularity imposed by importation is often helpful, it can sometimes be too restrictive. It is often convenient to combine several traits dealing with different aspects of the same operator. This is common when specifying something that is not easily thought of as an abstract data type. Trait inclusion involves the same union of clauses as trait importation, but allows the included operators to be further constrained. Consider, for example

**Reflexive:**

```trait
introduces #: #: T, T → Bool
constrains #: so that for all [t: T]
  t #: t = true
```

**Symmetric:**

```trait
introduces #: #: T, T → Bool
constrains #: so that for all [t1, t2: T]
  t1 #: t2 = t2 #: t1
```

**Transitive:**

```trait
introduces #: #: T, T → Bool
constrains #: so that for all [t1, t2, t3: T]
  (((t1 #: t2) & (t2 #: t3)) ⇒ (t1 #: t3)) = true
```

**Equivalence:**

```trait
includes Reflexive, Symmetric, Transitive
```

Equivalence has the same associated theory as the less structured trait

**Equivalence1:**

```trait
introduces #: #: T, T → Bool
constrains #: so that for all [t1, t2, t3: T]
  t1 #: t1 = true
  t1 #: t2 = t2 #: t1
  (((t1 #: t2) & (t2 #: t3)) ⇒ (t1 #: t3)) = true
```

Any legal trait importation may be replaced by trait inclusion without either making the trait illegal or changing the associated theory. It does involve the sacrifice of the checking that ensures that the imported traits may be understood independently of the context in which they are used. We use importation when we can incorporate a theory unchanged, inclusion when we cannot.

5. Renaming and exclusion

The specification of Equivalence in the previous section relied heavily on the coincidental use of the operator #: and the sort identifier T in three separate traits.
In the absence of such happy coincidences, renaming can force names to coincide, keep them from coinciding, or simply replace them with more suitable names.

The phrase

Tr with [id1 for id2]

stands for the trait Tr with every occurrence of id2 (which must be either a sort or operator identifier) replaced by id1. Notice that if id2 is a sort identifier this renaming may change the signatures associated with some operators.

If TableSpec contains the generated by and partitioned by of Section 2, the specification

ArraySpec: trait
imports IntegerSpec
includes TableSpec
with [defined for # ∈ #, assign for add, read for eval,
Array for Table, Integer for Index]

stands for

ArraySpec: trait
imports IntegerSpec
introduces
new: → Array
assign: Array, Integer, Val → Array
defined: Integer, Array → Bool
read: Array, Integer → Val
isEmpty: Array → Bool
size: Table → Card

constrains new, assign, defined, read, isEmpty so that
Array generated by [new, assign]
Array partitioned by [defined, read]
for all [ind, ind1: Integer, val: Val, t: Array]

read(assign(t, ind, val), ind1) =
   if ind = ind1 then val else read(t, ind1)
defined(ind, new) = false
defined(ind1, assign(t, ind, val)) = ((ind = ind1) | defined(ind1, t))
size(new) = 0
size(add(t, ind, val)) = if ind ∈ t then size(t) else size(t) + 1
isEmpty(t) = (size(t) = 0)

A final point raised by the example of this section is the importance of distinguishing between the history of a specification (how it was constructed) and the structure presented to a reader. A reader familiar with TableSpec might prefer to read the first version of ArraySpec; others might find it distracting to have to understand the more general structure before understanding ArraySpec.
6. Assumptions

We often construct fairly general specifications that, we anticipate, will later be specialized in a variety of ways. Consider, for example,

BagSpec: trait
  introduces
    {}: -> Bag
    insert: Bag, Elem -> Bag
    delete: Bag, Elem -> Bag
    #: #: Bag, elem -> Bool
  constrains
    Bag generated by [{}, insert]
    Bag partitioned by [delete, ∈]
    for all [m: Bag, e, e1: Elem]
      ∈ {} = false
      ∈ insert(m, e1) = (e = e1) | (e ∈ m)
      delete({}), e) = {}
      delete(insert(m, e), e1) =
        if e = e1 then m else insert(delete(m, e1), e)

We might specialize this to IntBag by renaming Elem to Integer and including it in a trait in which operators dealing with Integer are specified, e.g.,

IntBag: trait
  imports IntegerSpec
  includes BagSpec with [Integer for Elem]

The interactions between BagSpec and IntegerSpec are very limited. Nothing in BagSpec makes any assumptions about the meaning of the operators (other than =) that occur in IntegerSpec, e.g., 0, +, and <. Consider, however, extending BagSpec to BagSpec1 by adding an operator rangeCount,

BagSpec1: trait
  imports BagSpec, Cardinal
  introduces
    rangeCount: Bag, Elem, Elem -> Integer
    #: #: Elem, Elem -> Bool
  constrains
    rangeCount so that for all [e1, e2, e3: Elem, m: Bag]
    rangeCount({}), e1, e2) = 0
    rangeCount(insert(m, e3), e1, e2)) =
      rangeCount(m, e1, e2) + (if (e1 < e3 & (e3 < e2) then 1 else 0)

Here, BagSpec1 makes no assumptions about the properties of the < operator. Suppose, however, that this is not what we intend. We might have definite ideas about the properties that < must have in any specialization, e.g., that it should
define a total ordering. We specify such a restriction with an assumption:

BagSpec2: trait
  assumes Ordered with [Elem for T]
  imports BagSpec, Cardinal
  introduces
    rangeCount: Bag, Elem, Elem -> Integer
  constrains rangeCount so that for all [e1, e2, e3: Elem, m: Bag]
    rangeCount({}, e1, e2) = 0
    rangeCount(insert(m, e3), e1, e2) =
    rangeCount(m, e1, e2) + (if (e1 < e2 & e3 < e2) then 1 else 0)

In constructing the theory associated with BagSpec2, the assumption would be treated as if

Ordered with [Elem for T]

had been included. This could be used to derive various properties of BagSpec2, e.g., that rangeCount is monotonic in its last argument. (Ordered is defined in the Larch Handbook.)

Whenever BagSpec2 is imported or included in another trait, however, the assumption will have to be discharged. In

IntBag1: trait
  includes BagSpec2 with [Integer for Elem]
  imports IntegerSpec

this would amount to showing that the (renamed) theory associated with Ordered is a subset of the theory associated with IntegerSpec. Often, the assumptions of a trait are used to discharge the assumptions of traits it imports or includes.

7. Consequences

We have now looked at those parts of the Larch Shared Language that determine the theory associated with a valid trait. That subset of the language contains some checkable redundancy; e.g., assumptions are checked when a trait is included or imported, and constrains lists are checked against the axioms associated with them. We now turn to a part of the language whose only purpose is to introduce checkable redundancy, in the form of assertions about the theory associated with a trait.

There are two kinds of consequence assertions:
- That the theory associated with a trait contains another theory;
- That the theory associated with a trait 'adequately' defines a set of operators in terms of other operators.

The first kind of assertion is made using implies. Consider, for example, adding to BagSpec2,

implies for all [m: Bag, e1, e2, e3: Elem]
  (e2 < e3) => (rangeCount(m, e1, e2) ≤ rangeCount(m, e1, e3))
**Implies** can be used to indicate intended consequences of a specification, both for checking and to increase the reader's insight. The theory to be implied can be specified using the full power of the language, e.g., by using `generated by` and `partitioned by`, or by referring to traits defined elsewhere.

The second kind of assertion is made using `converts [Ops]`. **Converts** is used to say that the specification adequately defines a collection of operators, i.e., that each variable-free term containing the operators in Ops is provably equal to a variable-free term that does not contain any of the operators in Ops. A common problem with axiomatic systems is deciding whether there are 'enough' axioms. **Converts** provides a way of making a checkable statement about the adequacy of a set of axioms. Consider, for example, adding to TableSpec:

```
converts [isEmpty]
```

This says that each term containing `isEmpty`, such as `isEmpty(new)` or `isEmpty(add(new, ind, val))`, is equal to a term that does not contain `isEmpty`.

Now consider adding to TableSpec the stronger assertion:

```
converts [isEmpty, eval]
```

Terms containing subterms of the form `eval(new, ind)` are not convertible to terms that do not contain `eval`, so an error message of the form `eval(new, ind) not convertible` would be generated. This would present a problem if we did not wish to add an axiom to resolve this incompleteness. We therefore provide a mechanism to allow specifiers to indicate that the unconvertibility of certain terms is acceptable. If TableSpec were modified to include

```
exempts for all [ind: Index] eval(new, ind)
```

the checking associated with the `converts` would now require that the theory associated with TableSpec must contain either

- an equation, \( t = t_1 \), where \( t_1 \) has no occurrences of `isEmpty` or `eval`, or
- an equation \( t' = t_1 \), where \( t' \) is a subterm of \( t \), and \( t_1 \) is an instantiation of `eval(new, ind)`.

This checking ensures that each term containing operators in the `converts` list is either defined by the axioms (in terms of operators not in the list) or explicitly exempted.

8. **IfThenElse and equality**

In our examples we made use of some apparently unconstrained operators: `if then else` and `=`, with a variety of signatures. In fact, the appearance of these operators leads to the implicit incorporation of the traits IfThenElse and Equality.
Whenever a term of the form if b then $t_1$ else $t_2$ occurs in a trait we replace the mixfix symbol if then else by the prefix symbol ifThenElse. If $t_1$ and $t_2$ are of the same sort, T1, we also import the trait

\textbf{IfThenElse with [T1 for T]}

into the enclosing trait. Whenever a term of the form $t_1 = t_2$ occurs in a trait, if $t_1$ and $t_2$ are of the same sort, T1, we append the trait

\textbf{Equality with [T1 for T]}

to the consequences of the enclosing trait. These traits are defined in the Larch Reference Manual.

Notice that ifThenElse and $=$ are simply two examples of operator overloading. In the Larch Shared Language, every operator is made up of an identifier or operator symbol and a signature. If the signature is deducible from context, it need not be written. This is why signatures appear only in the \textit{introduces} clauses of the examples in this paper.

9. Some further examples

The following series of examples is adapted from the Larch Handbook. We include them here to illustrate some ways in which the facilities introduced above can be used, and to introduce some syntactic sugar. In reading these specifications, keep in mind that they are not themselves ends, but rather means to write interface specifications.

Our first example is an abstraction of those data structures that “contain” elements, e.g., Set, Bag, Queue, Stack. We have found it useful both as a starting point for specifications of various kinds of containers, and as an assumption for generic operators. The crucial part of the trait is the \textit{generated by}. It indicates that any term of sort C is equal to some term in which new and insert are the only operators with range C—even if this trait is included in one that introduces additional operators that return values of sort C. This means that any theorems proved by induction over new and insert will remain valid.

\textbf{Container: trait} \hspace{1em} % C's contain E's
\textbf{introduces}
new: $\rightarrow$C
insert: C, E$\rightarrow$C
\textbf{constrains} C so that C \textit{generated by} [new, insert]

The next example incorporate Container as an assumption. Notice that it constrains new and insert as well as the operator it introduces, isEmpty. The \textit{converts} indicates that this trait contains enough axioms to adequately specify isEmpty. Because of the \textit{generated by}, this can be proved by induction over terms of sort C, using new
as the basis and $\text{insert}(c, e)$ in the induction step. The explicit axioms in this trait are written without an ‘$=$’. An axiom of the form $\text{term}$ is syntactic sugar for the equation $\text{term} = \text{true}$.

**IsEmpty: trait**
- **assumes** Container
- **introduces** isEmpty: $C \rightarrow \text{Bool}$
- **constrains** isEmpty, new $\text{insert}$ so that for all $[c: C, e: E]$
  - isEmpty(new)
  - $\neg(\text{isEmpty}(\text{insert}(c, e)))$
- **implies** converts [isEmpty]

The next two examples assume Container. The **exempts** indicate that should these traits be included into a trait that claims the convertibility of next or rest, that trait need not convert the terms next(new) or rest(new).

**Next: trait**
- **assume** Container
- **introduces** next: $C \rightarrow E$
- **constrains** next, insert so that for all $[e: E]$
  - next(insert(new, $e$)) = $e$
- **exempts** next(new)

**Rest: trait**
- **assumes** Container
- **introduces** rest: $C \rightarrow C$
- **constrains** rest, insert so that for all $[e: E]$
  - rest(insert(new, $e$)) = new
- **exempts** rest(new)

The next example specifies properties common to various data structures such as stacks, queues, priority queues, sequences, and vectors. It augments Container by combining it with IsEmpty, Next, and Rest. The **partitioned by** indicates that next, rest, and isEmpty are sufficient to define equality over terms of sort $C$. Since we have little information about next and rest, the **partitioned by** does not yet add much to the associated theory.

**Enumerable: trait**
- **imports** IsEmpty, Next, Rest
- **includes** Container
- **constrains** $C$ so that $C$ **partitioned by** [next, rest, isEmpty]

The next example specializes Enumerable by further constraining next, rest, and insert. Sufficient axioms are given to convert next and rest. The axioms that convert isEmpty are inherited from the trait Enumerable, which inherited them from the
trait IsEmpty.

PriorityQueue: trait
  assumes TotalOrder with [E for T]  
  includes Enumerable  
  constrains next, rest, insert so that for all [q: C, e: E]  
  next(insert(q, e)) =  
    if isEmpty(q) then e  
    else if next(q) ≤ e then next(q) else e  
  rest(insert(q, e)) =  
    if isEmpty(q) then new  
    else if next(q) <~ e then insert(rest(q), e) else q  
  implies converts [next, rest, isEmpty]

In a trait, such as PriorityQueue, that defines an ‘abstract data type’ there will generally be a distinguished sort (C in this case) corresponding to the ‘type of interest’ of [7] or ‘data sort’ of [3]. In such traits, it is usually possible to categorize the operators whose range is the distinguished sort into ‘generators’, those operators which the sort is generated by, and ‘extensions’, which can be converted into generators. Operators whose domain includes the distinguished sort and whose range is some other sort are called ‘observers’. Observers are usually convertible, and the sort is usually partitioned by one or more subsets of the observers and extensions.

The next example illustrates a specialization of Container that does not satisfy Enumerable. It augments Container by combining it with IsEmpty and Cardinal, and introducing two new operators. Notice that we include Container, because we intend to constrain operators inherited from it, but import IsEmpty and Cardinal, because we do not intend to constrain any operator inherited from them. Constrains MSet is a shorthand for a constrains clause listing all the operators whose signature includes MSet. The partitioned by indicates that count alone is sufficient to distinguish unequal terms of sort MSet. Converts [isEmpty, count, delete] is a stronger assertion than the combination of an explicit converts [count, delete] with the inherited converts [isEmpty].

MultiSet: trait
  assumes Equality with [Elem for T]  
  imports Cardinal, IsEmpty with [MSet for C]  
  includes Container with [MSet for C, {} for new]  
  introduces count: Elem, MSet → Bool  
    delete: Elem, MSet → MSet  
    size: MSet → Card  
  constrains MSet so that  
    MSet partitioned by [count]  
    for all [c: MSet, e1, e2: E]  
    count({}, e1) = 0
\[
\text{count}(\text{insert}(c, e1), e2) = \text{count}(c, e2) + (\text{if } (e1 = e2) \text{ then } 1 \text{ else } 0)
\]
\[
\text{size}(\{\}) = 0
\]
\[
\text{size}(\text{insert}(c, e)) = \text{size}(c) + 1
\]
\[
\text{delete}(\{\}, e1) = \{\}
\]
\[
\text{delete}(\text{insert}(c, e1, e2)) = \\
\text{if } e1 = e2 \text{ then } c \text{ else } \text{insert}(\text{delete}(c, e2), e1)
\]

\textbf{implies converts} \{\text{isEmpty}, \text{count}, \text{delete}\}

The next example specifies a generic operator. It uses Enumerable as an assumption to delimit the applicability of this operator to containers for which it is possible to enumerate the contained elements. (To understand why we assume Enumerable rather than Container, imagine defining extOp for a MultiSet.) The \textbf{exempts} indicates that we do not intend to fully define the meaning of applying extOp to containers of unequal size. Notice that elemOp is totally unconstrained in this trait. This prevents us from having many interesting implications to state at this stage.

PairwiseExtension: \textbf{trait}

\textbf{assumes} Enumerable

\textbf{introduces}

\begin{itemize}
  \item elemOp: \(E, E \rightarrow E\)
  \item extOp: \(C, C \rightarrow C\)
\end{itemize}

\textbf{constrains} extOp so that all \([c1, c2: C, e1, e2: E]\)

\begin{itemize}
  \item extOp(new, new) = new
  \item extOp(insert(c1, e1), insert(c2, e2)) = \\
       \text{insert}(\text{extOp}(c1, c2), \text{elemOp}(e1, e2))
\end{itemize}

\textbf{implies converts} \{\text{extOp}\}

\textbf{exempts for all} \([c: C, e: E]\)

\begin{itemize}
  \item extOp(new, \text{insert}(c, e))
  \item extOp(\text{insert}(c, e), \text{new})
\end{itemize}

Now we specialize PairwiseExtension by binding elemOp to + over Cardinals:

PairwisePlus: \textbf{trait}

\textbf{assumes} Enumerable

\textbf{imports} Cardinal

\textbf{includes} PairwiseExtension with

\begin{itemize}
  \item \[# + # \text{ for elemOp}, # + # \text{ for extOp, Card for E}\]
\end{itemize}

\textbf{implies} Commutative with \[# + # \text{ for } \bigcirc, \text{C for T}\]

The validity of the implication that + (of sort \(C\)) is commutative stems from the replacement of elemOp by + (of sort Card), whose constraints (in trait Cardinal) imply its commutativity.
10. Using shared language specifications

While this paper is about the Larch Shared Language, it is important to keep in mind that specifications written in this language are intended to be used in specifications written in a Larch interface language. Interface languages are programming-language-dependent. Everything from the modularization mechanisms to the choice of reserved words is influenced by the programming language. At present, there is only one moderately well-developed Larch interface language, the Larch/CLU language. An extremely short specification written in a version of that language appears below. It uses the trait MultiSet presented in Section 9, and defines an abstract data type that exports a type, ten_bag, and four procedures. In the programming language CLU, it would be implemented by a cluster.

The semantics of Larch/CLU incorporates semantic constructs from CLU. For example, the meaning of signal in Larch/CLU derives from the meaning of signal in CLU—which is different from the meaning of SIGNAL in PL/1 or MESA. Correspondingly, Larch/CLU uses CLU-like syntax for constructs in common, e.g., procedure headers. Other interface languages would use concepts and terminology based on their programming languages. We do not define either the syntax of the semantics of Larch/CLU here. Consequently, the reader should not worry about details of the example. A detailed description of the semantics of an early version of Larch/CLU is given in [16].

```
ten_bag mutable type exports singleton, add, remove, choose

based on sort MSet from MultiSet with [int for Elem]

singleton = proc(e: int) returns(b: ten_bag)
    modifies nothing
    ensures new(b) & b = insert({}, e)

add = proc(b: ten_bag, e: int) signals (too_big)
    modifies at most [b]
    ensures normally bpost = insert(bpre, e) except too_big when size(bpre) = 10

remove = proc(b: ten_bag, e: int)
    modifies at most [b]
    ensures bpost = delete(bpre, e)

choose = proc(b: ten_bag) returns(e: int)
    requires ¬isNullOrEmpty(b)
    modifies nothing
    ensures count(b, e) > 0

end ten_bag
```
The names in this example tie it to two other kinds of formal text: traits in the Shared Language, and programs in CLU. Operators (e.g., insert), sort names (e.g., MSet), and trait names (e.g., MultiSet) provide the link to a theory defined by a collection of traits. Names of procedures (e.g., add), formal parameters (e.g., e), types (e.g., int), and signals (e.g., too_big) provide the link to implementations of the specification. The primary job of an interface language is to bring these two together. For example, the based on clause connects type names and sort names. The requires and ensures clauses contain operators, formal parameters, and signal names. These are used together to constrain the relationship between the values of the actuals on entry to a procedure and their values on exit from the procedure.

Each procedure's specification can be studied in isolation—in contrast to traits, where the core of the specification involves the interactions among operators. Of course, to understand or reason about the type, it is still necessary to consider the specifications of all its procedures. CLU's type-checking ensures the soundness of a data type induction principle for this type. This would enable us to prove that the size of any ten_bag value generated by a non-erroneous program is less than or equal to 10.

Induction over the procedures of a data type is distinct from induction over the generating operators of a sort, and is used to prove a different kind of theorem. Each value of type ten_bag can be represented by a term of sort ten_bag, but not every term represents a value that can be obtained using the procedures of the type. For example, we can prove by induction over {} and insert that for every term of sort ten_bag there is a term representing a larger ten_bag.

This specification is satisfied by a CLU cluster implementing one type, ten_bag, with four procedures, singleton, add, remove, and choose. It says nothing about 'implementation' of sorts (such as MSet) or operators (such as {} and insert). These auxiliary constructs are defined solely for the purpose of writing interface specifications; they do not exist in programs.

Execution errors, on the other hand, are properties of programs; they do not exist in traits. Requires clauses and signals provide means to specify two different ways of dealing with erroneous conditions. For example, add must raise a signal if adding another element would make its argument too big, whereas the implementor of choose is allowed to assume that it will not be called with an empty ten_bag.

Choose is probably the most interesting procedure in this example. Its specification says that it must return some value in the ten_bag it is passed, but does not say which value. Moreover, it does not even require that different invocations of choose with the same value produce the same result. Choose is an example of non-determinism, and therefore cannot be specified by equating its result to a term.

Non-determinism in an interface should not be confused with incomplete specification in a trait. We often intentionally introduce operators in traits without giving enough axioms to fully define them. Sometimes further properties will be given in other traits; sometimes the weaker theory allows greater flexibility in the implementation of an interface. However, we always insist on the substitution
property for operators; only in ‘applicative’ languages is this property likely to be
guaranteed for programming language ‘functions’.

A final thing to notice about the specification of choose is that it has a non-trivial
requires clause. The preconditions implied by this clause cannot be checked by the
program calling choose. The size operator is available for reasoning about pro-
cedures, but the interface does not supply any corresponding procedure. Ensuring
that execution of a program using this type is error-free would require some sort of (formal or informal) proof about the program.

11. Discussion

We felt that it was important to carry the design of the Larch Shared Language
through to the smallest details. This ensured that we did not overlook things that
would turn out to be less trivial than they appeared. It allowed us to complete and
check a fair number of examples. Finally, it was a necessary preliminary to the
development of the support tools that we envision for Larch. The language embodies
a large number of decisions, some of them more fundamental than others.

Among the less fundamental decisions were those dealing with syntax. We tried
to make the surface syntax of the Shared Language comprehensible to readers of
specifications, even at the expense of requiring quite a lot of punctuation (e.g., many
lengthy reserved words). However, there is still room for experimentation and
improvement here. It might make sense to adopt a more terse basic notation, and
provide a variety of reading aids (e.g., prettyprinters, cross-reference tools) in a
full-blown system.

The rest of this section touches on more fundamental decisions. These decisions
may be wrong, but it would probably not be easy to change any of them without
significantly affecting the character of the language.

A key assumption underlying our design was that specifications should be con-
structed and reasoned about incrementally. This led us to a design that ensures that
adding things to a trait never removes formulas from its associated theory. The
desire to maintain this monotonicity property led us to construe the equations of a
trait as denoting a first order theory. Had we chosen to take the theory associated
with either the initial or final interpretation of a set of equations (as in [6] and
[15]), the monotonicity property would have been lost.

While we felt that many traits would correspond to complete abstract data types,
we felt that many would not. This led us to introduce generated by and partitioned
by as independent constructs. Generated by is used to close the set of constructors
of a sort, and partitioned by to close the set of observers. Separating these constructs
affords the specifier considerable flexibility.

Great flexibility is also afforded by the freedom to substitute, in a with list, for
any operator or sort identifier in a trait. In effect, all such identifiers in a trait are
formals. In an earlier version of the Larch Shared Language we had explicit lambda
abstraction. We discovered, however, that our initial assumptions about which names to make parameters were often incorrect. In particular, we discovered that often we wished to substitute for a name that we had failed to make a parameter. On the other hand, we frequently used the same identifier for the actual as the formal, because in specific instances we did not need to use all the potential parameters.

Another important aspect of names in the Larch Shared Language is that operator names are qualified by a signature rather than by either a single sort or a trait. This is in contrast to many programming languages, e.g., CLU. This decision was forced upon us by our desire to make heavy use of overloading in specifications.

Reading specifications is an important activity, and what one sees when reading a specification is a syntactic object, i.e., a trait, rather than the theory. For this reason, we chose to use syntactic transformations to define the mechanisms for combining Larch Shared Language specifications. However, for each of our combining operations on traits, there is a corresponding operation on theories such that the theory associated with any combination of traits is the same as the combination of their associated theories. In an earlier version of the Larch Shared Language [9], we had one mechanism that violated this property, without.

We devoted a great deal of attention to mechanisms for introducing checkable redundancy into specifications. Assumes, imports, and includes differ only in the checking associated with each. Constrains lists and the consequences section have no effect on the theory associated with a trait. They exist only to supply checkable redundancy. We chose to make the introduction of redundancy relatively fine-grained. Thus, for example, we have constrains list of operators rather than lists of 'protected' sorts.

The introduction of mechanisms to facilitate checking was not without some cost. The Larch Shared Language would be considerably smaller without them. Furthermore, experience indicates that it takes people roughly as long to learn those parts of the language involved with checking as it does to learn the part required to generate theories.

In contrast to our emphasis on syntactic mechanisms for building traits, we included a number of semantic constraints on the legality of traits, which were chosen to detect classes of errors that we expected to be common. A powerful theorem prover will be the heart of any implementation of the Larch Shared Language. Most of the properties to be checked are undecidable. Thus the best that any checker can do is to answer ‘definitely OK’, ‘definitely bad’, or ‘too hard’. We think that for most of the checks, the third answer will not occur too frequently. Although we do not yet have much experience to support this belief, we are encouraged by recent progress in the area of rewrite rule systems generally, and the Reve system specifically [5, 13].

In many respects, the Larch Shared Language is distinguished as much by what it does not include as by what it does.

The Shared Language provides no mechanism for 'hiding' operators. The hiding mechanisms of other specification languages allow one to introduce auxiliary
operators that do not have to be implemented. These operators are not completely hidden, since they must be read to understand the specification, and they are likely to appear in reasoning based on the specification. However, the operators appearing in a Shared Language specification are all auxiliary. Thus the introduction of a hiding mechanism would have no effect.

There is no mechanism other than sort checking for restricting the domain of operators. Terms such as \texttt{eval(new, i)} are considered to be well-formed. Furthermore, no special 'error' elements are introduced to represent the value of such terms. As discussed in the previous section, preconditions and errors are handled at the interface language level.

Similarly, non-determinism is left to the interface language. It is frequently useful to write incomplete specifications that admit distinct equivalence relations on terms (and non-isomorphic models). That is to say there are distinct terms that are neither provably equal or provably unequal. However, it is always the case that for every term \( t \), \( t = t \). The whole mathematical basis of algebra and the Larch Shared Language depends on the ability to freely substitute 'equals for equals'. This property would be destroyed by the introduction of 'non-deterministic functions'.

Since our approach to specification frequently leads us to construct traits in which many things are left unconstrained, we do not include 'completeness' among the properties that are required of a well-formed trait. Instead, we provide mechanisms (\textit{converts} and \textit{exempts}) that allow the specifier to define the completeness properties he would like checked. The choice will often depend on the intended interaction between a trait and the interface specifications that use it.

We have chosen not to use 'higher-order' entities in the Larch Shared Language. Traits are simple textual objects. Their associated theories are first-order theories. We have completely sidestepped the subtle semantic problems associated with parameterized theories, theory parameters, and the like [4].

12. Status and plans

We are still in the early phases of the Larch project. A primitive checker for the Shared Language has been implemented. In addition to parsing specifications, this program checks various context sensitive constraints and provides mechanisms for 'expanding' assumptions, importations, and inclusions. This checker is an interim tool. We designed our specification language in tandem with an editing and viewing tool. Many language design decisions were influenced by the presumption that specifications would be produced and read interactively using this tool. A first design [17] is complete, and a preliminary implementation is currently in use.

We are in the process of implementing term-rewriting software that we hope will provide much of the theorem-proving capability needed for analyzing specifications. The definition of the Larch Shared Language calls for a number of checks for which there can be no effective procedure. We have what we believe are useful procedures,
based on sufficient or necessary (but not both) conditions, for some of these checks, e.g., consistency and 'completeness' [12]. We are working on procedures for the others, e.g., checking constrains clauses. This is a difficult task. Diagnostics present a particularly vexing problem: How should relatively complicated theorem-proving procedures report problems to users who are not familiar with either their internal structure or the theory underlying them?

It is too soon to draw any strong conclusions about the utility of Larch in software development. We have written a significant number of Larch Shared Language specifications. On the whole, we were pleased with the specifications, and with the ease of constructing them. While writing them, we uncovered several design errors by inspection; we are encouraged that many of these errors would have been uncovered by the checks called for in the language definition. However, until we have implemented tools that will allow us to gain some experience with automated semantic checking, it is impossible to know how helpful these checks will be.

Larch Shared Language specifications are not an end in themselves. Our experience with Larch interface specifications is slight. We are in the process of documenting Larch/CLU, and plan to begin writing substantial specifications in that language. That experience should give us a much firmer basis for evaluating the Larch Shared Language and the Larch style of specification.

Larch Shared Language Reference Manual

0. Structure of manual

This part of the paper is a self-contained reference manual for the Larch Shared Language. In it we give the syntax and static semantics of the Larch Shared Language. We also define how theories are associated with traits.

Section 1 presents a grammar for the kernel subset of the Larch Shared Language. Section 2 defines the context sensitive checking and the theory associated with each specification written in the kernel subset.

Section 3 extends the kernel subset by introducing mechanisms for specifying intended consequences of a specification written in the kernel subset.

Sections 4-10 define successive extensions to the language. They extend the grammar to introduce additional aspects of the language and describe any additional context sensitive checking required. They also provide a translation from the newly extended language to the previously defined subset. The result of this translation is subjected to all the applicable checking. The theory associated with any specification written in the full language is the same as the theory associated with its translation.

Section 11 describes additional checks, defined in terms of the theories associated with traits, that are associated with various language features. To be legal, a specification and each of the parts from which it is built must satisfy these checks as well as the context sensitive checks described earlier.
Finally, Section 12 presents a reference grammar for the entire language.

1. Kernel syntax

1.1. Syntactic conventions

| alternative separator
{e} e is optional
e* zero or more e's
e*, zero or more e's, separated by commas
e+ one or more e's
alpha alpha is a nonterminal symbol
alpha alpha is a terminal symbol
'( ') parentheses as terminal symbols
(e) parentheses for grouping syntactic expressions

1.2. Grammar

\[
\begin{align*}
\text{trait} & ::= \text{traitId} : \text{trait} \ \text{traitBody} \\
\text{traitBody} & ::= \text{simpleTrait} \\
\text{simpleTrait} & ::= \{ \text{opPart} \} \ \text{propPart}* \\
\text{opPart} & ::= \text{introduces} \ \text{opDcl}* \\
\text{opDcl} & ::= \text{opld} \text{ signature} \\
\text{signature} & ::= \text{domain} \rightarrow \text{range} \\
\text{domain} & ::= \text{sortId}* , \\
\text{range} & ::= \text{sortId} \\
\text{propPart} & ::= \text{asserts} \ \text{props} \\
\text{props} & ::= \text{generators}* \ \text{partitions}* \ \text{axioms}* \\
\text{generators} & ::= \text{sortId} \ \text{generated} \ \text{bylist}* , \\
\text{partitions} & ::= \text{sortId} \ \text{partitioned} \ \text{bylist}* , \\
\text{bylist} & ::= \text{by} [ \text{sortedOp}*, ] \\
\text{sortedOp} & ::= \text{opDcl} \\
\text{axioms} & ::= \text{for all} [ \text{varDcl}, ] \ \text{equation}* \\
\text{varDcl} & ::= \text{varId}* , \ : \ \text{sortId} \\
\text{equation} & ::= \text{term} = \text{term} \\
\text{term} & ::= \text{sortedOp} \ ( (\text{term}, , )) | \text{varId} \\
\text{opld} & ::= \text{alphaNumeric} + | \text{opForm} \\
\text{opForm} & ::= \{ \# \} \ \text{opSym} ( \# \ \text{opSym})* \ {\#} \\
\text{opSym} & ::= \text{specialChar} + | . \ \text{alphaNumeric} + \\
\text{traitId} & ::= \text{alphaNumeric} + \\
\text{sortId} & ::= \text{alphaNumeric} + \\
\text{varId} & ::= \text{alphaNumeric} + 
\end{align*}
\]
Comments start with % and terminate with end of line. They may appear after any token.

2. Simple traits

2.1. Context sensitive checking

simpleTrait:
The sets of varld's, sortld's and opld's appearing in a trait must be disjoint.
Each sortld appearing anywhere in a simpleTrait must appear in its opPart.
Each sortedOp appearing anywhere in a simpleTrait must appear in its opPart.

opDcl:
Each opForm must have the same number of #'s as the number of occurrences of sortld's in the signatories domain.

generators:
The range of each sortedOp must be the sortld of the generators.
At least one sortedOp in each bylist must have a domain in which the sortld of the generators does not occur.

partitions:
The domain of each sortedOp must include the sortld of the partitions.
The range of at least one sortedOp in each bylist must be different from the sortld of the partitions.

axioms:
Each varld used in a term must appear in exactly one varDcl.
No varld may occur more than once in [varDcl*,:].

equation:
The sorts of both term's must be the same, where
The sort of a term of the form sortedOp '{(term*, ')'} is the range of the sortedOp.
The sort of a term of the form varld is the sortld of the varDcl in which the varld is declared.

term:
In sortedOp '{(term*, ')'} the domain of the sortedOp must be the sequence of the sorts of the term's in term*.

2.2. Associated theory

We associate a theory with each trait. A theory is an inference-closed set of well-formed formulas (wffs) of typed first-order predicate calculus with equality. This section defines the theory associated with a simpleTrait.

We adopt the conventional meanings of the equality symbol (=), the propositional connectives (&, |, −, ⇒, . . .), and the quantifiers (∀ and ∃). Since we use the same
symbols to denote connectives as to denote the operators of traits Boolean and Equality, wffs containing unquantified terms can be ambiguous. However, since traits Boolean and Equality give the propositional connectives and = the same meanings as the corresponding predicate connectives, the ambiguity is harmless.

The theory, call it Th, associated with a simple Trait is defined by:

Axioms: Each equation, universally quantified by the varDcl's of its containing axioms, is in Th.

Inequation: \( \neg (\text{true:} \rightarrow \text{Bool} = \text{false:} \rightarrow \text{Bool}) \) is in Th.

First order predicate calculus with equality: Th contains the axioms of conventional typed first-order predicate calculus with equality and is closed under its rules of inference.

Induction: If the trait has a generators with sortId S and a bylist by \([\text{op}_1, \ldots, \text{op}_n]\), and \( P(s) \) is a wff with a free variable, \( s \), of sort S, Th contains the wff

\[
\forall [s: S] P(s)
\]

if for each \( \text{op}_i \) in \([\text{op}_1, \ldots, \text{op}_n]\)

\( Q_i \Rightarrow P(\text{op}_i(x_1, \ldots, x_k)) \) is in Th, where

- \( k \) is the arity of \( \text{op}_i \),
- the \( x_j \)'s are variables that do not appear free in \( P \), and
- \( Q_i \) is the conjunction of \( P(x_j) \), for each \( j \) such that the \( j \)th argument of \( \text{op}_i \) is of sort S.

Reduction: If the trait has a partitions with sortId S and a bylist by \([\text{op}_1, \ldots, \text{op}_n]\), Th contains the wff

\[
\forall [s_1, s_2: S](Q \Rightarrow s_1 = s_2)
\]

where \( Q \) is the conjunction, for each \( \text{op}_i \) in \([\text{op}_1, \ldots, \text{op}_n]\),

and each \( j \) such that the \( j \)th argument of \( \text{op}_i \) is of sort S of:

\[
\forall [x_j: S_1, \ldots, x_k: S_k] (\text{Subst}(\text{op}_i, j, s_1) = \text{Subst}(\text{op}_i, j, s_2)),
\]

where

- \( S_1, \ldots, S_k \) is the domain of \( \text{op}_i \), and
- \( \text{Subst}(\text{op}_i, j, s) \) is \( \text{op}(x_1, \ldots, x_k) \) with \( s \) substituted for \( x_j \).

3. Consequences and exemptions

Exempts and consequences affect only the checking (see Section 11.5) and do not affect the theory. We add to the grammar the productions:

\[
\text{trait} ::= \text{traitId : trait traitBody \{consequences\} \{exempts\}}
\]

\[
\text{consequences} ::= \text{implies} \text{conseqProps \{converts\}}
\]

\[
\text{conseqProps} ::= \text{props}
\]

\[
\text{converts} ::= \text{converts \text{conversion}^*},
\]

\[
\text{conversion} ::= [\text{sortedOp}^*]
\]

\[
\text{exempts} ::= \text{exempts \text{exemptTerms}^*}
\]

\[
\text{exemptTerms} ::= \{\text{for all \{varDcl\} \}_* \text{ term}^*,
\]
3.1. Context sensitive checking

conseqProps:
If the props of the conseqProps is appended to the propPart of the containing trait, the resulting trait must satisfy the checks of Section 2.

exempts:
Each term must satisfy the checks of Section 2.1.

4. Constrains clauses

Constrains clauses affect only the checking (see Section 11.4), not the theory. We add to the grammar the productions:

\[
\begin{align*}
propPart & ::= \text{constrains props} \\
\text{constrains} & ::= \text{constrains (sortld|sortedOp*,)} \text{ so that}
\end{align*}
\]

4.1. Translation

Replace the constrains by asserts.

5. Implicit signatures and partial OpForms

In the kernel language each sortedOp is an opDcl. Here we relax this restriction to allow omitted and partial signatures and omitted #’s. We add to the grammar the production:

\[
\text{sortedOp ::= opld \{ \rightarrow range}\}
\]

5.1. Context sensitive checking

There must be a unique mapping from occurrences of sortedOp’s to opDcl’s of the traitBody such that the translation described in Section 5.2 produces a legal traitBody and for each sortedOp, opDcl pair:

The opld’s match, i.e.,
They are the same, or
They are both opForms and the one in the sortedOp is the same as the one in the opDcl with all #’s removed.
If the sortedOp includes → range, it is the same as the range of the opDcl.

5.2. Translation

The checking ensures that each occurrence of a sortedOp corresponds to a unique opDcl. The translation is simply to replace it by that opDcl.
6. Mixfix operators

In the language presented thus far, all operators are treated as either nullary or prefix. Here we relax that restriction. We replace the grammar for \textit{term} by:

\[
\begin{align*}
\text{term} &::= \text{secondary} | \text{if secondary then secondary else term} \\
\text{secondary} &::= \{ \text{opSym} \} \text{ primary} (\text{opSym primary})* \{ \text{opSym} \} \\
\text{primary} &::= \text{sortedOp} \{ (\text{term*}, \text{')}) \mid \text{varld} \mid (\text{term'}) \}
\end{align*}
\]

6.1. Translation

\textit{equation}:

It is necessary to resolve the grammatical ambiguity between the $=$ connective in \textit{equations} and the $=$ \textit{opSym}. In any \textit{equation} the first occurrence of $=$ that is not bracketed by parentheses or within an \textit{if then else} is the \textit{equation} connective; the remainder are \textit{opSyms}. Parentheses can be used to enforce any desired parsing.

\textit{term}:

Translate each \textit{term} of the form if b then t$_1$ else t$_2$ into a \textit{term} of the form ifThenElse(b, t$_1$, t$_2$).

\textit{secondary}:

Translate each \textit{secondary} containing \textit{opSym’s} into a \textit{primary} of the form opld'((term*, ')), where

- opld is derived by replacing each \textit{primary} in the \textit{secondary} by #.
- term\text{*}, is the sequence of \textit{primary’s}.

\textit{primary}:

After the previous transitions have been performed, remove the outer parentheses from \textit{primary’s} of the form '((term')).

7. Boolean terms as equations

It is convenient to use terms of sort \textit{Bool} as equations. We add to the grammar the production:

\[
\textit{equation} ::= \textit{term}
\]

7.1. Context sensitive checking

The \textit{term} must be of sort \textit{Bool}.

7.2. Translation

Replace the \textit{term} by the equation

\textit{term} = true
8. External references

We add to the kernel grammar the productions:

\[
\begin{align*}
\text{traitBody} & : = \text{externals simpleTrait} \\
\text{externals} & : = \{\text{assumes}\} \{\text{imports}\} \{\text{includes}\} \\
\text{assumes} & : = \text{assumes traitRef}^*, \\
\text{imports} & : = \text{imports traitRef}^*, \\
\text{includes} & : = \text{includes traitRef}^*, \\
\text{traitRef} & : = \text{traitId} \\
\text{conseqProps} & : = \text{traitRef}^*, \text{props}
\end{align*}
\]

8.1. Context sensitive checking

externals:

Recursive externals are not permitted; i.e., the traitId of the containing trait may not appear in an externals, nor in any partial translation of a traitRef in its externals.

8.2. Translation

The translation of a trait is derived bottom-up; i.e., before a trait with traitRef's is translated, each of its traitRef's is replaced by the translation of the trait labeled by that traitRef's traitId. Let T be a trait whose simpleTrait is S and let E consist of the translations of the traitRef's in T's externals. The translation of T consists of:

- An opPart containing S's opDcls and E's opDcls,
- A propPart* containing S's propPart's and E's propPart's,
- An exempts containing T's exemptTerms and E's exemptTerms, and
- A consequences containing the props of
  - T's conseqProps,
  - the propPart's of the translations of the traitRef's in T's conseqProps, and
  - E's consequences.

9. Modifications

We add to the grammar the production:

\[
\begin{align*}
\text{traitRef} & : = \text{traitId} \{\text{renaming}\} \\
\text{renaming} & : = \text{with} \ [(\text{sortRename} | \text{opRename})^*,] \\
\text{sortRename} & : = \text{sortId for oldSort} \\
\text{oldSort} & : = \text{sortId} \\
\text{opRename} & : = \text{opId for oldOp} \\
\text{oldOp} & : = \text{sortedOp}
\end{align*}
\]
9.1. Context sensitive checking

traitRef:

No sortedOp may occur more than once as an oldOp.
No sortld may occur more than once as an oldSort.
Each oldSort must appear in an opDcl in the translation of the trait labeled by the traitld.
There must be a unique mapping from oldOp's to opDcl's of the translation of the trait labeled by the traitld, such that for each oldOp, opDcl pair:
The opld's match (see Section 5.1),
If the oldOp includes a domain, it is the same as the domain of the opDcl.
If the oldOp includes \( \rightarrow \) range, it is the same as the range of the opDcl.

9.2. Translation

The translation of the trait labeled by the traitld of the traitRef is modified by applying first the opRename's, and then the sortRename's:

Simultaneously, for each opRename, replace the opld part of each occurrence of the opDcl to which the oldOp maps by the opld of the opRename.
Then, simultaneously, for each sortRename, replace each occurrence of its oldSort by its sortld.

10. Implicit incorporation of Boolean, IfThenElse, and Equality

Three traits, Boolean, IfThenElse, and Equality, are implicitly incorporated into various other traits to assure uniform meanings for the operators they constrain.

10.1. Translation

Append the traitRef Boolean to the imports of each trait except Boolean.

Append the traitRef IfThenElse with \([T1 \text{ for } T]\) to the imports of each trait containing a term of the form \( \text{if } b \text{ then } t_1 \text{ else } t_2 \) in which \( t_1 \) and \( t_2 \) have the same sort, \( T1 \).

Append the traitRef Equality with \([T1 \text{ for } T]\) to the traitRef* of the conseqProps of each trait (except Equality) containing a term of the form \( t_1 = t_2 \) in which \( t_1 \) and \( t_2 \) have the same sort, \( T1 \).

10.2. Built-in traits

Boolean: trait
    introduces
        true: \( \rightarrow \text{Bool} \)
false: → Bool
¬#: Bool → Bool
# & #: Bool, Bool → Bool
# | #: Bool, Bool → Bool
# ⇒ #: Bool, Bool → Bool
# = #: Bool, Bool → Bool

asserts Bool generated by [true, false] for all [b: Bool]
¬true = false
¬false = true
(true & b) = b
(false & b) = false
(true | b) = true
(false | b) = b
(true ⇒ b) = b
(false ⇒ b) = true
(true = b) = b
(false = b) = ¬b

implies converts [¬, &, |, ⇒, =]

IfThenElse: trait

introduces ifThenElse: Bool, T, T → T

asserts for all [t1, t2: T]
ifThenElse(true, t1, t2) = t1
ifThenElse(false, t1, t2) = t2

implies converts [ifThenElse]

Equality: trait

introduces #: = #: T, T → Bool

asserts T partitioned by [=]

for all [x, y, z: T]
(x = x)
(x = y) = (y = x)
((x = y) & (y = z)) ⇒ (x = z)

11. Semantic checking

In addition to the syntactic constraints specified above, we require that each trait be logically consistent, discharge the assumptions of its external traits, be a conservative extension of its imports, be properly constraining, and imply its consequences.

11.1. Consistency

A traitBody is consistent if its associated theory does not contain the equation
true: → Bool = false: → Bool
11.2. Assumptions

Let $A(T)$ be all of the \textit{assumes} of the \textit{traits} imported or included in $T$, and $R(T)$ be the result of translating $T$ after removing these \textit{assumes}. $A(T)$ is \textit{discharged} by $T$ if the theory associated with the translation of each \textit{traitRef} of $A(T)$ is a subset of the theory associated with $R(T)$.

11.3. Imports

The theory associated with a \textit{trait} must be a \textit{conservative extension} of the theory associated with the translation of each \textit{traitRef} in its \textit{imports}; i.e., if \textit{trait} T1 imports T2 and $W$ is a \textit{wff} containing only operators introduced in T2, $W$ is in the theory associated with T1 if and only if it is in the theory associated with T2.

11.4. Constraints

A \textit{propPart} is \textit{properly-constraining} if it implies properties of only the operators in its \textit{constrains}. The occurrence of a \textit{sortId} in a \textit{constrains} stands for the list of all \textit{sortedOp}'s in the containing \textit{trait}'s \textit{opPart} whose \textit{signature}'s include that \textit{sortId}.

Let $T$ be a \textit{trait} and $P$ be the \textit{propPart} \texttt{constrains \textit{sortedOp}*}, so that \texttt{props}. $P$ is properly-constraining in the \textit{trait} consisting of $T$ plus $P$ if and only if each \textit{wff} in the theory associated with $T$ plus $P$ is also in the theory associated with $T$ or else contain an \textit{op} in \textit{sortedOp*}.

Since the translation of a \textit{traitRef} converts \textit{constrains} to \textit{asserts}, this check is performed only on traits in which \textit{constrains} appear explicitly.

11.5. Consequences

A \textit{trait} \textit{implies} its \textit{consequences} if the theory associated with its \textit{conseqProps} is a subset of the theory associated with the \textit{trait} and the [\textit{sortedOp*},] in each \textit{converts} is convertible. Convertibility is defined using the theory and \textit{exempts} of a \textit{trait}.

\texttt{conseqProps}:

The theory associated with \texttt{conseqProps} must be a subset of the theory of the \textit{trait} in which the \textit{consequences} appears. The theory associated with a \texttt{conseqProps} is the theory associated with the \textit{traitBody}:

\begin{verbatim}
  includes traitRef*,
  opPart
  asserts props
\end{verbatim}

where \texttt{traitRef*}, and \texttt{props} form the \texttt{conseqProps}, and \texttt{opPart} is the \texttt{opPart} of the \textit{trait} in which the \textit{consequences} appears.

\texttt{conversion}:

Let $C$ be a \textit{conversion}. For each \textit{term}, $t$, that contains no variables of any sort appearing in a \textit{generators} in the containing \textit{trait}, the theory of the containing \textit{trait} must either
contain an equation \( t = u \), where \( u \) contains no \( \text{sortedOp} \) appearing in \( C \)'s \( \text{sortedOp*} \), or contain an equation \( t' = u \), where \( t' \) is a subterm of \( t \), and \( u \) is an instantiation of a \( \text{term} \) appearing in an \( \text{exempts} \) of the containing \( \text{trait} \).

12. Reference grammar for the Larch Shared Language

```plaintext
trait := traitld : trait traitBody {consequences} {exempts}
traitBody := externals simpleTrait
externals := {assumes} {imports} {includes}
assumes := assumes traitRef*,
imports := imports traitRef*,
includes := includes traitRef*,
traitRef := traitld {renaming}
renaming := with [(sortRename | opRename)*,]
sortRename := sortld for oldSort
oldSort := sortld
opRename := opld for oldOp
oldOp := sortedOp
sortedOp := opDcl|opld \(\rightarrow\) range
simpleTrait := {opPart} propPart*
opPart := introduces opDcl*
opDcl := opld : signature
signature := domain \(\rightarrow\) range
domain := sortld*,
range := sortld
propPart := (asserts | constrains) props
constrains := constrains (sortld | sortedOp*,) so that
props := generators* partitions* axioms*
generators := sortld generated bylist*,
partitions := sortld partitioned bylist*,
bylist := by [sortedOp*,]
axioms := for all [varDcl*,] equation*
varDcl := varld* :, sortld
equation := term \{= term\}
term := secondary if secondary then secondary else term
secondary := {opSym} primary (opSym primary)* \{opSym\}
primary := sortedOp \{'(term*,',')\}| varld \{'(term')\}
opld := alphaNumeric +| opForm
opform := \{#\} opSym (# opSym)* \{#\}
opSym := specialChar + | . alphaNumeric +
traitld := alphaNumeric +
sortld := alphaNumeric +
```
varld ::= \text{alphaNumeric} +
consequences ::= \text{implies} \text{conseqProps} \{\text{converts}\}
conseqProps ::= \text{traitRef}^{*}, \text{props}
converts ::= \text{converts} \text{conversion}^{*},
conversion ::= [\text{sortedOp}^{*},] \text{exempts} ::= \text{exempts} \text{exemptTerms}^{*}
exemptTerms ::= \{\text{for all [}\varDcl^{*},] \text{term}^{*},$

Acknowledgment

Butler Lampson and Jeannette Wing have looked at many versions of the Larch Shared Language. Their penetrating questions have been a great help to us. Dave Detlefs, Ron Kownacki, and Joe Zachary all worked on syntax and static semantics checkers for various versions of the language. In doing so, they discovered problems in both the language and the reference manual. Their efforts were a big help. Versions of this paper were read by a number of people. We received particularly useful advice from Soren Prehn, Mary Shaw, and the members of IFIP Working Groups 2.2 and 2.3. Mary-Claire van Leunen went over this version of the paper with a fine tooth comb, and found more errors than we would have hoped.


References


