Lecture 7. Overview

- Introduction to Larch two-tiered approach
- Syntax and semantics of LSL
- A look at several examples
- Checking correctness of LSL specifications
- Details about the tool `LSL`.
- Introduction to `LP`.

Next time - proving properties with LP.
For references, consult newsgroup.

What is Larch

Two-tiered definitional approach to specification.
 - One language is designed for a specific programming language (*Larch interface language*)
 - The other is independent of any programming language (*Larch Shared Language* LSL).

- Larch interface languages exist for CLU, C (LCL), etc.
- Specify information needed to use the interface and to write programs that implement it
- Deal with what can be observed about the behavior of components written in a particular programming language
- Incorporate programming language specific notations or features such as side effects, exception handling, iterators, concurrency.

Interface Languages Examples

Example of LCL:

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

Example in specification for CLU:

```
addWindow = proc (v: View, w: Window, c: Coord)
  signals (duplicate)
  modifies v
  ensures v' = addW(v, w, c)
    except when w in v signals duplicate
                       ensures v' = v^
```

Here we need to know the meaning of interface language constructs (proc, signals, modifies) and the meaning of operators appearing in expressions (addW, in).

Idea of Larch

- specify mathematical abstractions (abstract data types) in LSL tier
- specify programming pragmatics in the interface tier
- keep difficult parts in the LSL tier since
 - LSL abstractions are likely to be reusable
 - LSL has a simple underlying semantics, so difficult to make mistakes
 - Easier to make and check claims about semantic properties of LSL.

We are just concerned with specification and verification in this course. So, we will not cover interface languages. Tools like `LCLint` use `LCL` to find errors in C programs.

Example - Table trait

```
Table: trait
  includes Positive (Card for P)
  introduces
    new: -> Tab
    add: Tab, Ind, Val -> Tab
    __\in __: Ind, Tab -> Bool
    lookup: Tab, Ind -> Val
    isEmpty: Tab -> Bool
    size: Tab -> Card
  asserts
    \forall i, i': Ind, val: Val, t: Tab
       lookup(add(t, i, val), i') ==
                if i = i' then val else lookup(t, i');
       ~ (i \in new);
       i \in add(t, i', val) == (i = i') \/ (i \in t);
       size(new) == 0;
       size(add(t, i, val)) ==
                if i \in t then size(t) else size(t) + 1;
       isEmpty(t) == (size(t) = 0);
```


A Look at Some LSL Constructs

- `introduces` declares a list of operators (f-n identifiers) and their signatures (types, or *sorts*, of their domain and range)
- Equations are of the form *LHS* == *RHS*. Can be abbreviated.
So, ~(i \in new) is ~(i \in new) == true.
- Characters "__" in an operator indicate that the operator will be used in mixfix expression.
- `asserts` declares axioms of the datatype.
- `includes` allows adding theories associated with other datatypes to the trait.
  - `Positive` defines +, 0, 1, etc.
  - The datatype defined in `Positive` was called `P`. `Card for P` renames occurrences of `P` by `Card`.
- The theory associated with a trait is that of the *union* of all of the `introduces` and `asserts` clauses of trait body and included traits.

Example of Operators with Renaming

```
Reflexive: trait
   introduces __*__: T, T -> Bool
   asserts \forall t : T
       t * t
Symmetric : trait
   introduces __*__: T,T -> Bool
   asserts \forall t, t' : T
       t * t' == t' * t
Transitive: trait
   introduces __*__: T, T -> Bool
   asserts \forall t, t', t'' : T
       (t * t' /\ t' * t'') => t * t''
Equivalence1: trait
   includes Reflexive, Symmetric, Transitive
Equivalence: trait
    includes (Reflexive, Symmetric, Transitive)
            (@ for *)
```


Some More LSL Constructs

- `generated by`: which operators serve as *constructors*, i.e., given a data structure, what minimum set of operators can be used to construct it?
Example: in a stack with operators `Pop`, `Empty` and `Push`, `Empty` and `Push` are the constructors. All natural numbers can be generated by `0` and `succ`.

- `partitioned by` clause asserts that all distinct values of a sort can be distinguished by a given list of operators. Terms that are not distinguishable using any of the partitioning operators of their sort are equal.
Example: sets are partitioned by $\in$, because sets that contain the same elements are equal.

`Generated by` and `Partitioned by`, Cont'd

So, our `Table` trait was generated by ... partitioned by...

Adding an axiom
`Tab generated by new, add`
can be used to prove theorems by induction over `new` and `add`, e.g.
$\forall t : \texttt{Tab}(\texttt{isEmpty}(t) \vee \exists i : \texttt{Ind}(i \in t))$

Adding an axiom
`Tab partitioned by \in, lookup`
can be used to derive theorems that do not follow from equations alone, e.g.
$\forall t : \texttt{Tab}, i, i\prime : \texttt{Ind}, v : \texttt{Val}(\texttt{add}(\texttt{add}(t, i, v), i\prime, v) = \texttt{add}(\texttt{add}(t, i\prime, v), i, v))$

Another Example

Renaming may also change the signatures associated with some of the operators.

```
SparseArray: trait
  includes Integer, Table(Arr for Tab, defined for \in,
          assign for add, __[__] for lookup, Int for Ind)
```

This is the same as

```
Table: trait
  includes Integer, Positive (Card for P)
  introduces
    new: -> Arr
    assign: Arr, Ind, Val -> Arr
    defined: Int, Arr -> Bool
    __[__]: Arr, Int -> Val
    isEmpty: Arr -> Bool
    size: Arr -> Card
  asserts \forall i, i': Int, val: Val, t: Arr
    assign(t, i, val)[i'] == if i = i' then val else t[i'];
    ~defined(i, new);
    defined(i, assign(t, i', val)) ==
          (i = i') \/ defined(i, t);
    size(new) == 0;
    size(assign(t, i, val)) ==
          if defined(i, t) then size(t) else size(t) + 1;
    isEmpty(t) == (size(t) = 0);
```


Checks on LSL Specifications

- Consistency: traits whose theory contains `true == false` are illegal

- Theory containment (using `implies`). Claims like

```
implies \forall a: Arr, i : int
  defined(i,a) => ~ isEmpty(a)
```

Enables specifiers to include information they believe to be redundant to draw attention to smth or as a check of their understanding. Uses `partitioned by` and `generated by` clauses and references to other traits.

- Completeness: using `converts` operator. Example:

```
implies converts isEmpty
```

If the interpretations of all the other operators are fixed, there is a unique interpretation of `isEmpty` satisfying the axioms.

Completeness, Cont'd

What happens if you do not want to specify behavior of some operation completely, e.g., `lookup`?
- Can say `lookup(new, i) == errorVal`. But why?
- What can you say to make division complete?
- Instead, use `exempting` clause which specifies terms that need not be defined:

```
implies converts isEmpty, lookup
  exempting \forall i: Ind lookup(new, i)
```

- This means that if interpretations of the other operators and of all terms matching `lookup(new, i)` are fixed, there are unique interpretations of `isEmpty` and `lookup` that satisfy the trait's axioms. This is provable from the spec.

Some more specifications - Container Classes

`Container` class has common properties of data structures that contain elements.

```
Container(E, C): trait
  introduces
    new: -> C
    insert: E, C -> C
  asserts C generated by new, insert
```

`LinearContainer` includes `Container`, constrains `new` and `insert` and introduces operators. Can be specialized to defined stacks, queues, etc.

```
LinearContainer(E, C): trait
  includes Container
  introduces
    isEmpty: C -> Bool
    next: C -> E
    rest: C -> C
  asserts
    C partitioned by next, rest, isEmpty;
    \forall c: C, e: E
      isEmpty(new);
      ~isEmpty(insert(e, c));
      next(insert(e, new)) == e;
      rest(insert(e, new)) == new;
    implies converts isEmpty;
```


Container Classes, Cont'd

`PriorityQueue` specializes `LinearContainer`.

```
PriorityQueue(E, Q): trait
  assumes TotalOrder(E)
  includes LinearContainer(Q for C)
  introduces __\in __: E, Q: -> Bool
  asserts \forall e, e': E, q: Q
    next(insert(e, q)) ==
      if q = new then e
      else if next(q) < e then next(q) else e;
    rest(insert(e, q)) ==
      if q = new then new
      else if next(q) < e then insert(e, rest(q)) else q;
    ~(e \in new);
    e \in insert(e'. q) == e = e' \/ e \in q;
  implies
    \forall q: Q, e: E
      e \in q => ~(e < next(q));
    converts next, rest, isEmpty, \in
            exempting next(new), rest(new)
```


Constructing Data Types

Abstract data type's operators are categorized as generators, observers and extensions (sometimes in more than one way).
- Generators produce all the values of the sort
- Extensions are remaining operators whose range is the sort
- Observers are the operators whose domain is the sort and whose range is some other sort
- Abstract data type specification usually converts the observers and the extensions
- The sort is usually partitioned by at least one subset of the observers and extensions.

When do we stop generating equations?
- Write an equation defining the result of applying each observer or extension to each generator.

PriorityQueue as an Abstract DataType

`Q` is the distinguished sort, `new` and `insert` form a generator set, `rest` is an extension, `next`, `isEmpty` and `\in` are the observers, and `next`, `rest` and `isEmpty` form a partitioning set.

We have defined 4 out of 8 necessary equations in `PriorityQueue`, inherited two more from `LinearContainer`. The remaining two, `next(new)` and `rest(new)`, are explicitly exempted.

Example: `PriorityQueue`

`NatPriorityQueue`
Check assumption of `TotalOrder(Nat)` by `PriorityQueue`
Use the assertions of all traits except `TotalOrder`

`PriorityQueue`
Check implications
Use the assertions of `PriorityQueue` and the theories of `LinearContainer` and `TotalOrder`

`NaturalNumber`
Check ...
Use...

`LinearContainer`
Check implications
Use local assertions and theories of `Container`

`TotalOrder`
Check implications
Use local assertions

`Container`
Check implications and local assertions

Another Example

Given a binary operation $*$, `PairwiseExtention` defines a new binary operator on containers, @, by applying $*$ to each element.

```
PairwiseExtention(E, C): trait
  assumes LinearContainer
  introduces
    __*__: E,E -> E
    __@__: C,C -> C
  asserts \forall e, e' : E, c, c' : C
    new @ new == new;
    insert(e, c) @ insert(e', c') ==
            insert(e * e', c @ c');
  implies converts @
    exempting forall e: E, c: C
      new @ insert(e, c),
      insert(e, c) @ new;
```

Now we specialize `PairwiseExtention` by binding $*$ to an operator, $+$, whose definition is to be taken from the trait `Integer`.

```
PairwiseSum(C) : trait
  assumes LinearContainer (Card for E)
  includes Integer
    PairwiseExtention(Card for E, + for *, + for @)1<
  implies (Associative, Commutative)
    (+.C for \circ, C for T)
```


Operator Overloading vs Built-In Operators

- Some of the operators, like `if then else`, $=$, $\neq$, and Boolean operators are built in.
- Can always overload the operator by declaring it in the `introduces` clause. Larch deduces signatures from the context. For example:

```
OrderedString (E, Str): trait
  assumes TotalOrder(E)
  introduces
    empty: -> Str
    insert: E, Str -> Str
    __<__: Str, Str -> Bool
  asserts
    Str generated by empty, insert
    \forall e, e' : E, s, s' : Str
      empty < insert(e, s);
      ~(s < empty);
      insert(e, s) < insert(e', s') ==
         e < e' \/ (e = e' /\ s < s');
```

But can disambiguate directly.
Example: `a.S = b` means that `a` is of sort `S`.
This also defines signatures of `=` and `b`.

Composing LSL Specifications

- `includes` and `assumes`. Do the same thing but differ in the checking they entail
- `PriorityQueue` includes `LinearContainer`. Its `assumes` clause indicates that its theory also contains that of `TotalOrder`.
- use of `assumes` entails checking that the assumptions must be discharged whenever `PriorityQue` is incorporated into another trait. So, if we check the trait

```
Nat PriorityQueue:  trait
  includes PriorityQueue(Nat, NatQ), NaturalNumber
```

we need to check that assertions in the traits `PriorityQueue`, `LinearContainer` and `NaturalNumber` together imply that of `TotalOrder(Nat)`.

Enumerations, Tuples and Unions

- Provide compact readable representations for common kinds of theories.
- `Temp enumeration of cold, warm, hot` means an enumerated type with successor relation, s.t. `succ(cold) == warm` and `succ(warm) == hot`
- `Tuple` is used to introduce fixed-length tuples. So, `C tuple of hd: E, tl: S` is a shorthand for

```
introduces
  [__,__]: E,S -> C
  __.hd: C -> E
  __.tl: C -> S
  set_hd: C, E -> C
  set_tl: C, S -> C
asserts
  C generated by [__, __];
  C partitioned by .hd, .tl;
  \forall e, e': E, s, s': S
    [e,s].hd == e;
    [e,s].tl == s;
    set_hd([e,s], e') == [e',s];
    set_tl([e,s], s') == [e,s'];
```


Unions

- `Union of` corresponds to tagged unions. For example,

```
  S union of atom: A, cell: C
```

is the same as

```
S_tag enumeration of atom, cell
introduces
  atom: A -> S
  cell: C -> S
  __.atom: S -> A
  __.cell: S -> C
  tag: S -> S_tag
asserts
  S generated by atom, cell
  S partitioned by .atom, .cell, tag
  \forall a: A, c: C
     atom(a).atom == a;
     cell(c). cell == c;
     tag(atom(a)) == atom;
     tag(cell(c)) == cell;
```

Each field name is incorporated in three distinct operators!

LSL Design Decisions

- Specifications will be constructed and checked incrementally. So, adding axioms to a trait never invalidates theorems.
- Sometimes algebraic datatypes are complete, but sometimes they are not. `generated by` and `partitioned by` allow specification of all complete list of generators and partitioning sets
- There can be various styles of keywords, allowing for pretty printing, etc. These are specified in initialization files, `lslinit.lsi`.
- There are constructs allowing for checkable redundancy. It helps prove that wrong specifications will be detectably illegal. But need a theorem prover to fully check traits (see next lecture)

LSL Design Decisions, Cont'd

- `converts` clauses allow specification of checkable claims about completeness.
- No way to specify precedence of user-defined operators!
- Reuse is done with `includes`, `assumes` and renaming.
- No constructs for specifying partial functions, i.e., cannot restrict domains of operators.
- Traits are simple textual objects and their associated theories are first-order.

Checking LSL Specifications

- Run LSL to check the syntax and static semantics of Larch specifications and to generate LP proof obligations from their claims
- These are consistency, theory containment and relative completeness
- Typing `lsl` <`Trait-name`>, checks the file `Trait-name.lsl` for syntax and static semantics
- Typing `lsl -p ...` pretty-prints the trait file (Latex fonts are not available in distribution!)
- Typing `lsl -lp` <`Trait-name`> not only checks the file but also generates several files with proof obligations:
  - `Trait-name_Axioms.lp`
  - `Trait-name_Theorems.lp`
  - `Trait-name_Checks.lp`

Proof Obligations

- There are no cycles in the trait hierarchy.

Let $\sqsubset^+$ be transitive closure of relation defined by setting $S \sqsubset T$ iff T includes or assumes S.
Let $\Rightarrow^+$ be transitive closure of the relation defined by setting $S \Rightarrow T$ iff S implies T.
LSL checks for the following conditions:
1. $\sqsubset^+$ is a strict partial order
2. There are no traits S and T such that both $S \sqsubset^+ T$ and $S \Rightarrow^+ T$.

This means that traits can be checked separately. (BTW, $\Rightarrow^+$ is not a strict partial order)

Proof Obligations, Cont'd

LSL extracts six sets of propositions from each trait T:
- The assertions of T consist of the propositions in the `asserts` clauses of T and of all traits (transitively) included in T.
- The assumptions of T consist of all assertions of all traits (transitively) assumed by T.
- The axioms of T consist of its assertions and its assumptions.
- The immediate consequences of T consist of the propositions in T's `implies` clause and the axioms of all traits that T explicitly implies.

Proof Obligations, Cont'd

- The explicit theory of T consists of its axioms, the propositions in its `implies` clause, and the explicit theories of all traits S such that $S \sqsubset^+ T$ or $T \Rightarrow^+ S$. Explicit theory is not closed under logical consequence.
- The lemmas available for checking T, when Condition 2 is satisfied, consist of the explicit theories of all traits S such that $S \sqsubset^+ T$.

LinearContainer Revisited

```
LinearContainer(E, C): trait
  introduces
    isEmpty: C -> Bool
    next: C -> E
    rest: C -> C
    insert: C, E -> C
    new: -> C
    __\in __: E, C -> Bool
  asserts
    C generated by new, insert
    C partitioned by next, rest, isEmpty
    \forall c: C, e, e': E
      isEmpty(new);
      ~isEmpty(insert(c, e));
      next(insert(new, e)) == e;
      rest(insert(new, e)) == new;
      ~(e \in new);
      e \in insert(c, e') == e = e' \/ e \in c;
    implies
      forall c: C, e: E
         isEmpty(c) => ~(e \in c)
    converts isEmpty, \in
```


To check a hierarchy of traits...

Need to prove that the axioms of each trait T are consistent by discharging the following proof obligations:

- T's immediate consequences must follow from its axioms. If Condition 2 is satisfied, it is sound to use the lemmas available for T when performing this check.
- T's `converts` clauses must follow from its explicit theory. (The preceding proof obligation ensures that T's explicit theory follows from its axioms.)
- The assumptions of each trait explicitly included in T must follow from T's axioms.

LP Axioms for `LinearContainer`

```
declare sorts
  C, E
  ..
declare operators
  isEmpty: C -> Bool
  , next: C -> E
  , rest: C -> C
  , insert: C, E -> C
  , new: -> C
  , __ \in __: E, C -> Bool
  ..
%% Assertions
declare variables
  c: C
  e: E
  e': E
% main trait: LinearContainer
set name LinearContainer
assert
  sort C generated by new, insert
  ;sort C partitioned by next, rest, isEmpty
  ;(isEmpty(new))
  ;(~ isEmpty(insert(c, e)))
  ;(next(insert(new, e))) = (e)
  ;(rest(insert(new, e))) = (new)
  ;(~ (e \in new))
  ;(e \in insert(c, e')) = (e = e' \/ e \in c)
  ..
```


Translating LSL traits into LP

Logical system of LP consists of:
- a signature (given by declarations) and equations
- rewrite rules, operator theories
- induction rules and deduction rules Can be presented to LP incrementally, rather than all at once

Equations should be presented as rewrite rules, which LP uses to reduce terms to normal forms.
Rewriting relation should be terminating.
LP automatically converts axioms into rewrite rules.
But rewriting theory is not as powerful as equational theory (not everything provable can be proven via rewrite rules)

LP Operator Theories

LP provides special mechanisms for dealing with associativity and commutativity. These equations cannot be oriented into terminating rewrite rules.

Two nonempty operator theories:
- associative-commutative theory
  E.g. `assert ac +`
- commutative theory

So, term rewriting is done modulo these theories - much slower than conventional term rewriting

Sometimes, LP can suggest operator ordering which is potentially non-terminating but LP authors said that it has not caused difficulties yet.

Induction Rules

- LP uses induction rules to generate subgoals to be proved for the basic and induction steps in proofs by induction.
- Can have multiple rules for induction.

```
declare sorts E, S
declare operators
  {}: -> S
  {__}: E -> S
  __ \cup __: S, S -> S
  insert: S, E -> S
  ..
set name setInduction1
assert S generated by {}, insert
set name setInduction2
assert S generated by {}, {__}, \cup
```

Now we can use the appropriate rule when attempting to prove an equation by induction:

`prove` $x \subseteq (x \cup y)$ `by induction on` x
   `using setInduction2`

In LSL, there is typically just one `generated by` for a sort, but more might be useful.

Deduction Rules

- LP uses them to deduce equations from other equations and rewrite rules
- They can be asserted as axioms or proved as theorems
- E.g. `partitioned by` is equivalent to a universal-existential axiom, which can be expressed as a deduction rule

```
declare sorts E, S
declare operator ∈:  E, S -> Bool
declare variables e:  E, x, y:  S
assert when (∀ s), e ∈ x == e ∈ y
  yield x == y
```

This defines a deduction rule, which can also be expressed as `assert S partitioned by ∈`.
Equivalent to axiom

$$(\forall x, y : S)[(\forall e : E)(e \in x \equiv e \in y) \Rightarrow x = y]$$

LP can deduce that equation $e \in x == e \in (x \cup x)$ is the same as $x == x \cup x$.

Deduction Rules, Cont'd

- Also serve to improve the performance of LP and reduce the need for user interactions.

Examples are &-splitting law:

```
declare variables p, q:  Bool
when p & q == true yield p == true, q == true
```

and cancellation law for addition:

```
declare variables x, y, z:  Nat
when x + y == x + z yield y == z
```

- LP does not have too many of deduction rules and is more of a proof checker than an actual theorem prover.

Theorems for `LinearContainer`

```
%%% Theorems from trait LinearContainer
execute LinearContainer_Axioms
declare variables
  c: C
  e: E
  ..
%% Theorems
% main trait: LinearContainer
set name LinearContainer
assert
  (isEmpty(c) => ~ (e \in c))
  ..
```


Proof Obligations for `LinearContainer`

```
set script LinearContainer
set log LinearContainer
%%% Proof Obligations for trait LinearContainer
execute LinearContainer_Axioms
%% Implications
declare variables
  c: C
  e: E
  ..
% main trait: LinearContainer
set name LinearContainerTheorem
prove
  (isEmpty(c) => ~ (e \in c))
  ..
%% Conversions
freeze LinearContainer
%% converts isEmpty, \in
thaw LinearContainer
declare operators
  isEmpty': C -> Bool
  , __ \in' __: E, C -> Bool
  ..
% subtrait 0: LinearContainer
%       (isEmpty': C -> Bool for isEmpty: C -> Bool, __
%   \in' __: E, C -> Bool for __ \in __: E, C -> Bool)
```


Proof Obligations for `LinearContainer`, Cont'd

```
set name LinearContainer
assert
  sort C partitioned by next, rest, isEmpty'
  ;(isEmpty'(new))
  ;(~ isEmpty'(insert(c, e)))
  ;(~ (e \in' new))
  ;(e \in' insert(c, e')) = (e = e' \/ e \in' c)
  ..
declare variables
  _x1_: C
  _x1_: E
  _x2_: C
  ..
set name conversionChecks
prove (isEmpty(_x1_:C)) = (isEmpty'(_x1_:C))
qed
prove (_x1_:E \in _x2_) = (_x1_:E \in' _x2_)
qed
prove (_x1_:E \in _x2_) = (_x1_:E \in' _x2_)
qed
```


Forward Inference in LP

Produces consequences from a logical system. 4 methods of forward inference in LP:

- Automatic *normalization* produces new consequences when a rewrite rule is added to a system. LP keeps everything in normal form. Users can "immunize" them to protect them from automatic normalization. Can also "deactivate" rewrite rules and deduction rules to prevent them from being automatically applied.
- Automatic *application of deduction rules*.
- Computation of *critical paths* and Knuth-Bendix *completion procedure* (next lecture).
- Explicit *instantiation* of variables

Backward Inference in LP

Produces lemmas whose proof will suffice to establish a conjecture. 6 methods of backward inference in LP:

- *Proofs by induction*
- *Proofs by contradiction*. If an inconsistency follows from adding the negation of conjecture to LP's logical system, then it is a theorem
- *Proofs by conjunctions*. LP can be directed to prove t_1, ... t_n as subgoals of $t_1 ... \& t_n$. Reduces the expense of term rewriting.
- *Normalization* rewrites conjectures. If it normalizes to an identity, it is a theorem. Otherwise it becomes a subgoal to be proved.

Backward Inference in LP, Cont'd

- *Proofs by cases* can further rewrite a conjecture. When subdividing into cases $t_1 ... t_n$, then one subgoal is to prove $t_1 \mid ... \mid t_n$. The remaining are generated by substituting new constants for the variables of t_i in e to form $e_i\prime$ and tries to prove $e_i\prime$. If conjecture is a theorem, this may prove it. Otherwise, it may simplify the conjecture and make it easier to understand where the proof failed.

- *Proofs by implications* is a simplified proof by cases. For conjectures of type $t_1 \Rightarrow t_2$. More details next lecture.

food for slide eater

food for slide eater