Lecture 8. Overview

What happened last time?
- Introduction to Larch two-tiered approach
- Syntax and semantics of Larch Shared Language
- Algebraic specifications of abstract data types
- Trait hierarchies.
- When are LSL traits correct?

What will happen today?
- Introduction to LP.
- Rewrite rules.
- Inference mechanisms used by LP.
- A look at some examples.
- Input language syntax, running the tool, etc.

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)

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
```


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)
  ..
```

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.

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
```


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))
  ..
```


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. If an equation or rewrite rule normalizes to an identity, it is discarded. 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* (happens automatically or can be applied manually to immune equations).

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)
```

Forward Inference (Cont'd)

- Computation of *critical pairs* and Knuth-Bendix *completion procedure*. This basically amounts to computing the most general unifier between two equations (next lecture). Completion - closure of critical pairs.
- Explicit *instantiation* of variables. For example, have `a < (b + c) -> true` and `(b + c) < d -> true`. If we instantiate deduction rule `when x<y = true, y<z = true yield x<z = true` with a for x, b+c for y, and d for z, get conclusion `a < d -> true`.

Backward Inference in LP

Produces lemmas whose proof will suffice to establish a conjecture. 6 methods of backward inference in LP (actually even more - see manual):

- *Normalization* rewrites conjectures. If it normalizes to an identity, it is a theorem. Otherwise it becomes a subgoal to be proved.
- *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.

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 | ... | 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 $t_1 \Rightarrow t_2$. Proves the goal $t_2\prime$ using the hypothesis $t_1\prime \rightarrow$ *true*.

Users can determine which methods are applied automatically and in what order. Command

`set proof-method &, =>, normalization`

Default is `normalization`.


```
declare sorts Elem, Set
declare variables e, e': Elem
declare variables x, y, z: Set
declare operators
  empty:                -> Set
  singleton: Elem       -> Set
  __\union __: Set, Set -> Set
  __\in __: Elem, Set   -> Bool
  insert: Elem, Set     -> Set
  ..
set name set
assert ac \union
assert sort Set generated by empty, singleton, \union
assert
  (e \in singleton(e')) = (e = e')
  ;(e \in (x \union y)) = ((e \in x) \/ (e \in y))
  ;insert(e, x) = ((singleton(e) \union x))
  ;~(e \in empty)
  ..
set name extent
assert sort Set partitioned by \in
display extent
set name thm
prove x = (x \union x)
  instantiate s by x, s1 by (x \union x) in extent
  qed
set proof-methods =>, normalization
prove e \in x => insert(e, x) = x by induction
  resume by cases ec \in xc, ec \in xc1
    critical-pairs thmCaseHyp with thmInductHyp
    critical-pairs thmCaseHyp with thmInductHyp
  qed
```

LP session

```
Welcome to LP (the Larch Prover), Release 3.1a (95/04/27).
Copyright (C) 1994, S. J. Garland and J. V. Guttag
LP0.1: execute sample
LP0.1.1: declare sorts Elem, Set
LP0.1.2: declare variables e, e': Elem
LP0.1.3: declare variables x, y, z: Set
LP0.1.4: declare operators
  empty:                 -> Set
  singleton: Elem        -> Set
  __\union __: Set, Set  -> Set
  __\in __: Elem, Set    -> Bool
  insert: Elem, Set      -> Set
  ..
LP0.1.5:
LP0.1.6: set name set
The name-prefix is now 'set'.
LP0.1.7: assert ac \union
Added 1 fact named set.1 to the system.
LP0.1.8: assert sort Set generated by empty,singleton,\unior
Added 1 fact named set.2 to the system.
LP0.1.9: assert
  (e \in singleton(e')) = (e = e')
  ;(e \in (x \union y)) = ((e \in x) \/ (e \in y))
  ;insert(e, x) = ((singleton(e) \union x))
  ;~(e \in empty)
  ..
Added 4 facts named set.3, ..., set.6 to the system.
The system now contains 4 rewrite rules.  The rewriting
system is NOT guaranteed to terminate.
LP0.1.10:
```


LP Session (Cont'd)

```
LP0.1.11: set name extent
The name-prefix is now 'extent'.
LP0.1.12: assert sort Set partitioned by \in
Added 1 fact named extent.1 to the system.
LP0.1.13: display extent
Deduction rules:
extent.1: when \A e (e \in s <=> e \in s1) yield s = s1
LP0.1.14:
LP0.1.15: set name thm
The name-prefix is now 'thm'.
LP0.1.16: prove x = (x \union x)
Attempting to prove conjecture thm.1: x = x \union x
Suspending proof of conjecture thm.1
LP0.1.17:   instantiate s by x, s1 by (x \union x) in extent
Deduction rule extent.1 was instantiated to deduction rule
extent.1.1,       when \A e (e \in x <=> e \in (x \union x))
                  yield x = x \union x
Deduction rule extent.1.1 was normalized to formula
extent.1.1.1, x = x \union x
Conjecture thm.1
[] Proved by normalization.

LP0.1.18:   qed
All conjectures have been proved.
LP0.1.19:
LP0.1.20: set proof-methods =>, normalization
The proof-methods are now '=>-method, normalization'.
```


LP Session (Cont'd)

```
LP0.1.21: prove e \in x => insert(e, x) = x by induction
Attempting to prove thm.2: e \in x => insert(e, x) = x
Creating subgoals for proof by structural induction on 'x'
Basis subgoals:
  Subgoal 1: e \in empty => insert(e, empty) = empty
  Subgoal 2: e \in singleton(e1) =>
               insert(e, singleton(e1)) = singleton(e1)
Induction constants: xc, xc1
Induction hypotheses:
  thmInductHyp.1: e \in xc => insert(e, xc) = xc
  thmInductHyp.2: e \in xc1 => insert(e, xc1) = xc1
Induction subgoal:
  Subgoal 3: e \in (xc \union xc1) =>
              insert(e, xc \union xc1) = xc \union xc1
Attempting to prove level 2 subgoal 1 (basis step) for
proof by induction on x
Creating subgoals for proof of =>
New constant: ec
Hypothesis:
  thmImpliesHyp.1: ec \in empty
Subgoal:
  insert(ec, empty) = empty
Attempting to prove level 3 subgoal for proof of =>
Added hypothesis thmImpliesHyp.1 to the system.
Formula thmImpliesHyp.1, false, is inconsistent.
Level 3 subgoal for proof of =>
[] Proved by inconsistent hypothesis.
Level 2 subgoal 1 (basis step) for proof by induction on x
[] Proved =>.
```


LP Session (Cont'd)

```
Attempting to prove level 2 subgoal 2 (basis step) for
proof by induction on x
Creating subgoals for proof of =>
New constants: ec, e1c
Hypothesis:
  thmImpliesHyp.1: ec \in singleton(e1c)
Subgoal:
  insert(ec, singleton(e1c)) = singleton(e1c)

Attempting to prove level 3 subgoal for proof of =>
Added hypothesis thmImpliesHyp.1 to the system.
Level 3 subgoal for proof of =>
[] Proved by normalization.
Level 2 subgoal 2 (basis step) for proof by induction on x
[] Proved =>.

Attempting to prove level 2 subgoal 3 (induction step) for
proof by induction on x
Added hypotheses thmInductHyp.1, thmInductHyp.2 to the syste
Creating subgoals for proof of =>
New constant: ec
Hypothesis:
  thmImpliesHyp.1: ec \in (xc \union xc1)
Subgoal:
  insert(ec, xc \union xc1) = xc \union xc1

Attempting to prove level 3 subgoal for proof of =>
Added hypothesis thmImpliesHyp.1 to the system.
Suspending proof of level 3 subgoal for proof of =>
```

LP Session (Cont'd)

```
LP0.1.22:   resume by cases ec \in xc, ec \in xc1
Creating subgoals for proof by cases
Case justification subgoal:
  ec \in xc \/ ec \in xc1
Case hypotheses:
  thmCaseHyp.1.1: ec \in xc
  thmCaseHyp.1.2: ec \in xc1
Same subgoal for all cases:
  singleton(ec) \union xc1 \union xc = xc \union xc1

Attempting to prove level 4 subgoal to justify proof by case
Level 4 subgoal to justify proof by cases
[] Proved by normalization.
Attempting to prove level 4 subgoal for case 1 (out of 2)
Added hypothesis thmCaseHyp.1.1 to the system.
Deleted formula thmImpliesHyp.1, which reduced to ‘true’.
Suspending proof of level 4 subgoal for case 1 (out of 2)

LP0.1.23:     critical-pairs thmCaseHyp with thmInductHyp
The following equations are critical pairs between rewrite
rules thmCaseHyp.1.1 and thmInductHyp.1.
  thm.3: singleton(ec) \union xc = xc
Critical pair computation abandoned because a theorem
has been proved.
Level 4 subgoal for case 1 (out of 2)
[] Proved by normalization.

Attempting to prove level 4 subgoal for case 2 (out of 2)
Added hypothesis thmCaseHyp.1.2 to the system.
Deleted formula thmImpliesHyp.1, which reduced to ‘true’.
```


LP Session (Cont'd)

```
LP0.1.24:     critical-pairs thmCaseHyp with thmInductHyp
The following equations are critical pairs between rewrite
rules  thmCaseHyp.1.2 and thmInductHyp.2.
  thm.3: singleton(ec) \union xc1 = xc1
Critical pair computation abandoned because a theorem has
been proved.
Level 4 subgoal for case 2 (out of 2):
  singleton(ec) \union xc1 \union xc = xc \union xc1
[] Proved by normalization.
Level 3 subgoal for proof of =>:
  insert(ec, xc \union xc1) = xc \union xc1
[] Proved by cases ec \in xc, ec \in xc1.
Level 2 subgoal 3 (induction step) for proof by
induction on x:
  e \in (xc \union xc1) =>
            insert(e, xc \union xc1) = xc \union xc1
[] Proved =>.
Conjecture thm.2: e \in x => insert(e, x) = x
[] Proved by structural induction on ‘x’.

LP0.1.25:   qed
All conjectures have been proved.
```


Annotated LP Script

Results from `set script` *filename*. Then *filename*.`lpscr` contains the script.

```
set script sample
%% execute sample
declare sorts Elem, Set

...   BLAH BLAH (same as original) ...

prove x = (x \union x)
  instantiate s by x, s1 by (x \union x) in extensionality
  [] conjecture
qed

set proof-methods =>, normalization
prove e \in x => insert(e, x) = x by induction
    <> basis subgoal
      <> => subgoal
      [] => subgoal
    [] basis subgoal
    <> basis subgoal
      <> => subgoal
      [] => subgoal
    [] basis subgoal
    <> induction subgoal
      <> => subgoal
```


Annotated LP Script (Cont'd)

```
      resume by cases ec \in xc, ec \in xc1
        <> case justification
        [] case justification
        <> case ec \in xc
        critical-pairs thmCaseHyp with thmInductHyp
        [] case ec \in xc
        <> case ec \in xc1
        critical-pairs thmCaseHyp with thmInductHyp
        [] case ec \in xc1
      [] => subgoal
    [] induction subgoal
  [] conjecture
qed
```

Checking LSL Traits Using LP

Proof obligations in LSL traits require to check:
- Theory containment, that is, that claims follow from axioms
  - Proving an equation
  - Proving a "partitioned by"
  - Proving a "generated by"
  - Proving a "converts"
- Consistency

Proving an Equation - TotalOrder Example

```
TotalOrder(E): trait
  introduces
    __< __: E, E -> Bool
    __> __: E, E -> Bool
  asserts forall x, y, z: E
    ~(x < x);
    ((x < y) /\ (y < z)) => (x < z);
    (x < y) \/ (x = y) \/ (y < x);
    x > y == y < x
  implies
    TotalOrder (E, > for <, < for >)
    \forall x, y: E
        ~((x < y) /\ (y < x))
```


Axioms in TotalOrder_Axioms.lp

```
%%% Axioms for trait TotalOrder
%% Operator declarations
declare sorts
  E
  ..
declare operators
  __ < __: E, E -> Bool
  , __ > __: E, E -> Bool
  ..
set automatic-ordering off
%% Assertions
declare variables
  x: E
  y: E
  z: E
  ..
% main trait: TotalOrder
set name TotalOrder
assert
  (~ (x < x))
  ;((x < y /\ y < z) => x < z)
  ;(x < y \/ x = y \/ y < x)
  ;(x > y) = (y < x)
  ..
set automatic-ordering on
```


Theorems in TotalOrder_Checks.lp

```
set script TotalOrder
set log TotalOrder
%%% Proof Obligations for trait TotalOrder
execute TotalOrder_Axioms
%% Implications
declare variables
  x: E
  y: E
  z: E
  ..
% main trait: TotalOrder
set name TotalOrderTheorem
prove
  (~ (x < y /\ y < x))
  ..
set name TotalOrderTheorem
prove
  (~ (x > x))
  ..
prove
  ((x > y /\ y > z) => x > z)
  ..
prove
  (x > y \/ x = y \/ y > x)
  ..
prove
  (x < y) = (y > x)
  ..
```

Running LP (Cont'd)

```
LP0.1.24: prove (x > y \/ x = y \/ y > x)
Attempting to prove level 2 lemma TotalOrderTheorem.4:
       x = y \/ x > y \/ y > x
Level 2 lemma TotalOrderTheorem.4
[] Proved by normalization.
Deleted formula TotalOrderTheorem.4, which reduced to ‘true

LP0.1.25: prove (x < y) = (y > x)
Attempting to prove level 2 lemma TotalOrderTheorem.5:
      x < y = y > x
Level 2 lemma TotalOrderTheorem.5
[] Proved by normalization.

Attempting to prove conjecture TotalOrderTheorem.1
Deleted formula TotalOrderTheorem.5, which reduced to ‘true
Suspending proof of conjecture TotalOrderTheorem.1
End of input from file ‘chechik/TotalOrder_Checks.lp’.

LP2: critical-pairs TotalOrder with TotalOrder
The following equations are critical pairs between rewrite
rules TotalOrder.1 and TotalOrder.2.
  TotalOrderTheorem.6: ~(z < y /\ y < z)
The system now contains 1 formula and 4 rewrite rules.  The
rewriting  system is guaranteed to terminate.
Critical pair computation abandoned because a theorem has
been proved.
Conjecture TotalOrderTheorem.1: ~(x < y /\ y < x)
[] Proved by normalization.
The system now contains 5 rewrite rules.  The rewriting
system is guaranteed to terminate.
```


Running LP. File `TotalOrder.lplog`

```
LP (the Larch Prover), Release 3.1a (95/04/27) logging to
‘/homes/u1/chechik/TotalOrder.lplog’ on 12 March 1997
LP0.1.5: execute TotalOrder_Axioms
LP0.1.5.3: declare sorts E
LP0.1.5.5: declare operators
  __ < __: E, E -> Bool
  , __ > __: E, E -> Bool
  ..
LP0.1.5.7: set automatic-ordering off
LP0.1.5.10: declare variables
  x: E
  y: E
  z: E
  ..
LP0.1.5.15: set name TotalOrder
The name-prefix is now ‘TotalOrder’.
LP0.1.5.17: assert
  (~ (x < x))
  ;((x < y /\ y < z) => x < z)
  ;(x < y \/ x = y \/ y < x)
  ;(x > y) = (y < x)
  ..
Added 4 facts named TotalOrder.1, ..., TotalOrder.4 to
the system. The system now contains 4 formulas.
LP0.1.5.19: set automatic-ordering on
Automatic-ordering is now ‘on’.
The system now contains 4 rewrite rules.  The rewriting
system is  guaranteed to terminate.
All equations have been oriented into rewrite rules.  The
rewriting system is guaranteed to terminate.
```


How Hard Is It to Prove Equations?

Varying amounts of assistance. To check that `LinearContainer` implies `isEmpty(c) =>` $\sim$`(e` $\in$ `c)`, use a single LP command:

```
  resume by induction on c
```

When proving that `PriorityQueue` implies `e` $\in$ `q =>` $\sim$`(e` $<$ `next(q))`, need a lot more guidance.

Some theorems of `PriorityQueue`:

```
set name conversionChecks
prove (next(_x1_:Q)) = (next'(_x1_:Q))
qed
prove (rest(_x1_:Q)) = (rest'(_x1_:Q))
qed
prove (isEmpty(_x1_:Q)) = (isEmpty'(_x1_:Q))
qed
prove (_x1_:E \in _x2_) = (_x1_:E \in' _x2_)
qed
prove (_x1_:E \in _x2_) = (_x1_:E \in' _x2_)
qed
set name PriorityQueueTheorem
prove (e \in q => ~ (e < next(q)))
qed
```


Running LP (Cont'd)

```
LP0.1.8: declare variables
  x: E
  y: E
  z: E
  ..
LP0.1.13: set name TotalOrderTheorem
The name-prefix is now ‘TotalOrderTheorem’.
LP0.1.15: prove (~ (x < y /\ y < x))
Attempting to prove conjecture TotalOrderTheorem.1:
           ~(x < y /\ y < x)
Suspending proof of conjecture TotalOrderTheorem.1

LP0.1.20: set name TotalOrderTheorem
The name-prefix is now ‘TotalOrderTheorem’.
LP0.1.22: prove (~ (x > x))
Attempting to prove level 2 lemma TotalOrderTheorem.2:
           ~(x > x)
Level 2 lemma TotalOrderTheorem.2
[] Proved by normalization.

Attempting to prove conjecture TotalOrderTheorem.1
Deleted formula TotalOrderTheorem.2, which reduced to ‘true
Suspending proof of conjecture TotalOrderTheorem.1

LP0.1.23: prove ((x > y /\ y > z) => x > z)
Attempting to prove level 2 lemma TotalOrderTheorem.3:
            x > y /\ y > z => x > z
Level 2 lemma TotalOrderTheorem.3
[] Proved by normalization.
Deleted formula TotalOrderTheorem.3, which reduced to ‘true
```

How does the Proof Go?

Want to prove e ∈ q => ∼(e < next(q)). Go by induction:

```
prove by induction on q
```

LP generates two subgoals:

```
Basis subgoal:
  Subgoal 1: e \in new => ~(e < next(new))
Induction constant: qc1
Induction hypothesis:
  conversionChecksInductHyp.2: e \in qc1 => ~(e < next(qc1)
Induction subgoal:
  Subgoal 2:
   e \in insert(qc1, e1) => ~(e < next(insert(qc1, e1)))
```

It is able to prove basis subgoal by normalization. For subgoal 2, we ask to handle the case where qc1 = new:

```
resume by cases qc1 = new
```

This becomes

```
Case 1: ...
Case 2:
 (e = e1) \/ (e \in qc1) => ~(e < (if e1 < next(qc1)
                                   then e1 else next(qc1))
```


Proof (Cont'd)

So, we proceed with the next case:

```
resume by cases e1 < next(qc1)
```

and get

```
Case 1: (e = e1) \/ (e \in qc1) => ~(e < e1)
Case 2: (e = e1) \/ (e \in qc1) => ~(e < next(qc))
```

Now, for the first case, try setting e to e1:

```
resume by case e = e1
  % This case succeeds
  % Handle case e <> e1
complete
```

For the second case, do the same thing:

```
resume by case e = e1
  % This case succeeds
  % Handle case e <> e1
critical-pairs *CaseHyp with *InductHyp
qed
```


Proof Guidance

```
prove e \in q => ~(e < next(q)) by induction on q
    <> basis subgoal
    [] basis subgoal
    <> induction subgoal

    resume by case qc = new
      <> case qc = new
      [] case qc = new
      <> case ~(qc = new)
      resume by case next(qc) < e1
        <> case next(qc) < e1c
        resume by case e = e1c
          <> case ec = e1c
          complete
          [] case ec = e1c
          <> case ~(ec = e1c)
          [] case ~(ec = e1c)
        [] case next(qc) < e1c
        <> case ~(next(qc) < e1c)
        resume by case e = e1c
          <> case ec = e1c
          [] case ec = e1c
          <> case ~(ec = e1c)
          complete
          [] case ~(ec = e1c)
```


Proving a `Partitioned By`

"Debugging LSL Specs" say that this is checked. In fact, no proof obligations are generated for this. Instead, this becomes a deduction rule. For example, `LinearContainer` has `partitioned by next, rest, isEmpty`. This results in rule

```
LinearContainer.2:
      when next(q) = next(q1),
           rest(q) = rest(q1),
           isEmpty(q) <=> isEmpty(q1)
      yield q = q1
```

Proving a `generated by`

Again, there is nothing to prove. This becomes basis for induction. For `LinearContainer`, we get

```
Induction rules:
   LinearContainer.1:
       sort C generated by new, insert
```

Proving a `converts`

- Need to show that the axioms of the trait define the operators in the set relative to other operators in the trait.
- For `LinearContainer`, make two copies of `LinearContainer_Axioms`, where the second copy replaces all occurrences of `isEmpty` and `\in` by `isEmpty'` and `\in'`. Then can prove that

```
prove (isEmpty(x1) = (isEmpty'(x1)))
qed
prove (x2 \in x3) = (x2 \in' x3)
qed
```

User gives instruction to proceed by induction. - Exemptions are treated by specifically declaring that the behavior is the same for these cases. For example, for `PriorityQueue`, we have

```
assert next'(new) = next(new)
assert rest'(new) = rest(new)
prove next'(q) = next(q)
qed
prove rest'(q) = rest(q)
qed
...
```


Checking Consistency

Basically, we want to show that our system does not contain any axioms of the form `true = false`. The authors suggest running a command

```
complete
```

in order to complete the system. And if no `true = false` is generated, the system is complete.

But

- many equational theories cannot be completed at all
- some cannot be completed in an acceptable amount of time and space

Next Time

- Extended example
- General theorem proving
  - Rewrite rules
  - More on LP inteference
  - Running the tool - some useful commands
  - Managing proofs

For More Information

1. S.J. Garland, J. V. Guttag, A Guide to LP, The Larch Prover, December 1991, SRC report 82.
2. S. J. Garland, J. V. Guttag, Debugging Larch Shared Language Specifications, July 1990, SRC report 60.
3. HTML manual on LP, found at `/local/lib/LP/html/Index.html`. LP scripts in 1) and 2) were created for previous version of LP. Changes can be found at `/local/lib/LP/html/news/changes3_1.html`.
4. LP on-line help. Can say `?`, `help`, `help lp`, `help` *topic*, etc.

food for slide eater

food for slide eater

food for slide eater

food for slide eater