Lecture 9. Overview

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

Built-in Operators and Axioms

See Figure 6 on Page 17 of "A Guide to LP, The Larch Prover"

Built-in Deduction Rules

```
lp_not_is_true:  when not(p)         yield p = false
lp_not_is_false: when not(p) = false yield p
lp_and_is_true:  when p & q          yield p, q
lp_or_is_false:  when p | q = false  yield p = false,
                                           q = false
lp_iff_is_true:  when p <=> q        yield p = q
```

Also, `&`, $|$ and $<=>$ are `ac` operators and `=` is a `commutative` operator.

Term-Rewriting

- A *rewrite rule* is an ordered pair $\langle l, r\rangle$ of terms, usually written $l \rightarrow r$, s.t.
  - l is not a variable and
  - every variable that occurs in r also occurs in l.
- Rewrite rules have the same logical meaning as equations but behave differently operationally

- An *equational theory* (ET) is a set of facts axiomatized by a set of equations, i.e., everything that follows from these equations.
- A *rewriting theory* (RT) is everything that can be derived from a set of facts via rewriting rules.

- The user supplies an equation $l = r$ which gets (automatically) transformed into a rewrite rule.

Term-Rewriting - Example

Axioms:

```
nat.2: i+1 -> s(i)
nat.3: 0 + i -> i
nat.4: s(j) + i -> s(i + j)
nat.5: i < 0 -> false
nat.6: i < s(i) -> true
nat.7: s(i) < s(j) -> i < j
```

Automatic proof that $1 < 1 + 1$:

```
1 < 1                Conjecture
s(0) < s(0) + s(0)   Apply nat.2 3 times
s(0) < s(0 + s(0))   Apply nat.4
0 < 0 + s(0)         Apply nat.7
0 < s(0)             Apply nat.3
true                 Apply nat.6
```

Rewrite rules `nat.3` and `nat.7` can be applied in either order.

Rewrite Rules - Formal Definition

- A *substitution* σ is a mapping from variables to terms s.t. $\sigma(v)$ is identical to v for all but a finite number of variables
- A substitution σ *matches* a term t_1 to a term t_2 if $sigma(t_1)$ is identical to t_2.

- Rewriting system R defines a binary relation $\hookrightarrow_R$ (*rewrites or reduces directly to*) on the set of all terms
- Operationally, $t \hookrightarrow_R u$ if there is some rewrite rule $l - rightarrowr$ in R and some substitution σ that matches l to a subterm of t s.t. u is the result of replacing that subterm by $\sigma(r)$.
- Relation $\hookrightarrow_R^*$ is the reflexive transitive closure of $\hookrightarrow_R$. Thus, $t \hookrightarrow_R^* u$ iff there are terms $t_1 \ldots\ t_n$, s.t. $t = t_1 \hookrightarrow_R \ldots \hookrightarrow_R t_n = u$.
- Relation $\hookrightarrow_R^+$ is the transitive irreflective closure of $\hookrightarrow_R$.

Some Key Observations

- It is essential that R be *terminating* i.e. no infinite sequence of reductions.
- But... it is undecidable whether a set of rewrite rules is terminating.
- LP provides mechanisms to automatically orient many sets of equations into terminating rewrite systems (later on this lecture).

Some Key Observations (Cont'd)

- A term t is said to be *irreducible* if there is no term u s.t. $t \hookrightarrow u$.
- If $t \hookrightarrow^* u$ and u is irreducible, then u is *normal form* of t. A term can have many different normal forms. If there is only one, it is called the *canonical* form of the term. A terminating rewriting systems in which all terms have a canonical form is said to be *convergent*.
- For convergent systems, its rewriting theory is the same as its equational theory, but in general, this is not true!!!!! (and most systems in practice are not convergent)
- LP provides various ways to compensate for that (various inference rules).

Orienting Equations into Rewrite Rules

- LP automatically orients equations into rewrite rules.
- Command `set automatic-ordering off` causes LP to not do it.
- 3 types of ordering mechanisms for orienting equations into rewrite-rules. Command `set ordering` *method*.
  - Two registered orderings (`dsmpos` and `noeq-dsmpos`), based on LP-suggested partial orderings of operators, guarantee termination of sets of rewrite rules when no `commutative` or `ac` operators are present.
  - A `polynomial` ordering, based on user-supplied polynomial interpretation of operators, guarantees termination even when `commutative` or `ac` operators are present. But difficult to use.
  - Three brute-force ordering procedures which give users complete control over whether equations are oriented from left to right or right to left. But provide no guarantee about termination.
- Default is `noeq-dsmpos`.

Registered Orderings

- These use information in a *registry* to orient equations - height and status.
- *Height* relates pairs of operators. If an operator f has greater height than another operator g, LP orients equations by so that an occurrence of f is replaced by one or more occurrences of g. For example,
  $g(g(x)) = f(x)$ becomes
  $f(x) \rightarrow g(g(x))$

- *Status* information assigns relative weights to the arguments of operators with arity > 1. If operator h has `left-to-right`(`right-to-left`) status, more weight is assigned to h's leftmost(rightmost) arguments.

Registered Orderings

For example, if h has left-to right status,
  $h(f(x), x) = h(x, f(x))$ becomes
  $h(f(x), x) \rightarrow h(x, f(x))$.
If h has right-to-left status, then it becomes
  $h(x, f(x)) \rightarrow h(f(x), x)$
- If an operator has `multiset` status, its arguments are given equal weight. So, for our example, if h is multiset, the equation cannot be oriented.
- LP automatically assigns multiset status to `ac` and `commutative` operators.

Specification and Meaning of Registered Orderings

Command	Effect on Ordering
`register height f > g`	rewrite f to g
`register height f = g`	give them equal height
`register height f >= g`	rule out $g > f$
`register bottom f`	rewrite any non-bottom operator to f
`register top f`	rewrite f to any non-top operator
`register status right-to-left f`	assign more weight to right arguments of f
`register status left-to-right f`	assign more weight to left arguments of f
`register status multiset f`	assign equal weight to all arguments of f

- Can combine height information into a single command:
  `register height => > (&, |) > true = false`
- LP rejects commands that are not consistent additions to the registry, e.g., $f > g$ and $g > f$.

What If This is Not Enough?

- LP generates minimal sets of extensions to registry, *suggestions*, that would permit equations to be oriented.
- These will not violate user-entered rewrite rules.
- `noeq-dsmpos` ordering does not generate suggestions assigning equal heights to two operators.
- `dsmpos` does.
- `noeq-dsmpos` is faster but less powerful. - Usually, suggestions are added automatically, but can be overridden by `set automatic-registry off`. Then LP asks user to choose a suggestion.

Suggestions (Cont'd)

Example:

Want to orient $f(a, b) = f(b, a)$ with an empty registry.

LP presents the following suggestions:

```
    Direction  Suggestions
    ---------  -----------
 1.   ->         a > b    f(L)
 2.   ->         b > a    f(R)
 3.   <-         b > a    f(L)
 4.   <-         a > b    f(R)
```

but if user entered $f(a, b) \to f(b, a)$, only the first two suggestions would have been presented.

`unregister` command allows to delete the entire registry, or to remove operators from the bottom or top, but not remove height or status information in the registry.

Polynomial Orderings

- Requires considerable user input (i.e., do not use it in Assignment 3)
- Used to experiment with termination proofs of small sets of rewrite rules.
- `polynomial` ordering is based on user-supplied interpretations of operators by sequences of polynomials. A variable is interpreted by a sequence of identity polynomials, and a compound term - by the interpretation of its root operator applied to the interpretations of its arguments.
- One term is less than another if its interpretation is lexicographically less than that of the second term. (One polynomial is less than another if its value is less than that of the other for all sufficiently large values of its variables.)

Polynomial Orderings (Cont'd)

- Command `set ordering polynomial` *length* sets ordering to the one based on sequences containing *length* polynomials. If not specified, *length* is assumed to be 1.
- Command `register polynomial f p` assigns the sequence of p's as the polynomial interpretation of f. LP understands operator precedence.
- Example.

```
set ordering polynomial
register polynomial 0  2
register polynomial s  x + 2
register polynomial +  x * y
register polynomial <  x * y
```

- LP will orient $s(i) + j = s(i + j)$ from left to right, since polynomial interpretation $(i+2)*j$ dominates the interpretation $i+j+2$ for $j > 1$.
- `noeq-dsmpos` ordering produces the same set of rewrite rules but does not guarantee termination since $+$ is `ac`.

Brute-force Orderings

- `manual` ordering causes LP to ask the user how to orient each equation. User is allowed to choose either orientation, provided it results in a valid rewrite rule.
- `left-to-right` causes LP to orient equations into rewrite rules from left to right provided that results are valid rewrite rules.
- `either-way` behaves like `left-to-right` except that it orients an equation into a rewrite rule from right to left if that is possible and left to right if not.

Interacting with Ordering Procedures

When `automatic-ordering` and `automatic-registry` are `off`, LP prompts users to confirm any extensions to the registry or select an action for an equation LP is unable to orient.

```
The following sets of suggestions will allow the
equation to be ordered:
   Direction     Suggestions
   ---------     -----------
 1.  ->            a > b
 2.  <-            b > a
What do you want to do with the equation?
```

User can type `?` to see a menu:

```
Enter one of the following or type <ret> to exit.
 accept[1..2]  kill           postpone
 divide        left-to-right  right-to-left
 interrupt     ordering       suggestions
```

Meaning of options:

- `accept` confirms selected extension. If this option is missing from menu - no extension will orient the equation.

Interacting with Ordering Procedures (Cont'd)

- `divide` - LP adds two new equations that imply the original. small Useful for cases like $x/x = y/y$. LP asks the user to supply a name for a new operator, e.g., e, and will then declare the operator and assert two equations, $x/x = e$ and $y/y = e$
- `interrupt` - interrupts ordering process and returns LP to command level.
- `kill` - deletes this equation from the system.
- `left-to-right` - orients the equation without extending the registry. Removes any guarantee of termination. Same as `right-to-left`.
- `ordering` - displays the current registry.
- `postpone` - defers the attempt to orient this equation.
- `suggestions` - redisplays the LP-generated suggestions.

Problems when RT $\neq$ EQ

Example:

```
group1: (x * y) * z -> x * (y * z)
group2: i(x) * x -> e
group.3: e * x -> x
```

Have two terminal forms of term `(i(y) * y) * z`:

- `i(y) * (y * z)` (via `group.1`)
- `e * z` (via `group.2`), reduces to `z` (via `group.3`)

Equivalent under equational theory of group axioms but rewrite system cannot figure this out!

Other bad behaviors:

- LP may fail to reduce u and v to the same normal form even though $u \hookrightarrow v$
- Behavior of LP may be non-monotonic, i.e., reduce u and v using rewriting system R but not using $R \cup \{l \to r\}$

Critical-Pair Command

- method of extending the rewriting theory
- syntax: `critical-pairs group.1 with group.2`
- This computes
  `i(y) * (y * z) == e * z`

then reduced by `group.3` and oriented to give
  `group.4:  i(y) * (y * z) -> z`

How is this done?
- via unification (like in Prolog or ML!)
- `x * y` and `i(w) * w` can be unified by
  $\sigma = \{i(w)$ for x, w for $y\}$ or
  $\sigma\prime = \{i(e)$ for x, e for y, e for $w\}$
- For ordinary unification, there is always the *most general unifier*
- For some equational theories, there is no mgu. For commutative and `ac` theories, there are finite sets of *minimal unifiers*, i.e., unifiers that are not substitution instances of other unifiers.

Critical-Pairs (Cont'd)

- Let $l_1 \to r_1$ and $l_2 \to r_2$ be rewrite rules s.t. l_2 can be unified with a nonvariable subterm t_1 of l_1. l_1 and l_2 *overlap* at t_1.
- Let σ be mgu (or one of minimal unifiers) of l_2 and t_1
- *Critical-pair equation* associated with this overlap is
  $\sigma(l_1[t_1 \leftarrow r_2] == \sigma(r_1)$

i.e., result of reducing $\sigma(l_1)$ by each of the two rewrite rules.

Example:
- `critical-pairs * with *` deduces
  `e * z == i(y) * (y * z)` which reduces to
  `z == i(y) * (y * z)`
- Another application of the same command gives
  `z == i(i(z)) * e`
- Third gives `e * i(e) == i(i(e)) * (e * i(e))` which reduces to `i(e) == e`

Completion

- `complete` command causes computation of fixpoint of `critical-pairs * with *`.
- If computation finishes with an empty set of equations and a terminating set of rewrite rules, then we have a decision procedure (using reduction to normal form) for the equational theory.
- Using LP, it is advised to not complete the rewriting system because
  - it may not exist
  - may be too expensive to maintain or use
  - may lead to hard to read canonical forms
- Use `complete` to check for inconsistencies (and interrupt after a few iterations)
- `critical-pairs` and `complete` stop when they produce a consequence that results in proving the current conjecture. So, useful for finishing up proofs.

Instantiation

- Command `instantiate` *variable* `by` `term ...` `in` *names* does simultaneous substitution of specified terms for variables in named equations.
- Typical use with deduction rules. Example: have a rule
  `when (forall e) e ∈ x == e ∈ y yield x == y`

and a rewrite rule `e ∈ (x ∪ y) -> (e ∈ x) ∨ (e ∈ y)`. Instantiate `y` by `x ∪ x` and get a conclusion `x == x ∪ y`.
- Instantiation can be used as alternative to computing critical-pair equations. Example:
  `instantiate x by i(y) in group.1`

generates equation
  `(i(y) * y) * z == i(y) * (y * z)`

which reduces to
  `group.1.1, i(y) * (y * z) -> z`
- This is same as rule `group.4` by `critical-pairs`.

Activity and Immunity

- LP provides features for not using facts for normalization and deduction.
- To deactivate use command `make passive` *names*. Used for rules known to be inapplicable or expensive to apply.
- `display` command indicates passive facts by letter `P` after their name.
- Can activate again using `make active` *names*.
- Can "immunize" equations, rewrite rules, and deduction rules from automatic normalization or deduction. Commands `make immune` *names* and `make nonimmune` *names*. `display` indicates immune facts by letter `I`.
- Can set global settings, `set activity` (default is `on`) and `set immunity` (default is `off`).

Activity and Immunity (Cont'd)

- Intermediate degree of immunity.
- Commands `set immunity ancestor` and `make ancestor-immune` *names* prevent facts from being reduced by rules that are ancestors of the fact. Rule $a.1$ is ancestor of rule $a.1.2$
- Provides way to preserve instantiations of rewrite rules.
- `display` indicates ancestor-immune facts by letter `i`.

- Can do rule application by hand, using commands `normalize` *factNames* `with ruleNames` and `rewrite` *factNames* `with ruleNames`
- Commands `normalize conjecture with` *ruleNames* and `rewrite conjecture with` *ruleNames* apply named rules to the current conjecture.
- For deduction rules, command is `apply` *ruleNames* `to` *factNames*.

Managing Proofs

- "Prove as you would program. Design your proofs. Modularize them. Think about their computational complexity."
- Always set scripting and logging on at the start of an LP session. If too late, use command `history all`.
- Be careful to not let variables disappear too quickly in a proof. Once they are gone, you cannot do a proof by induction. Start with induction before =>, cases or `if`.
- Splitting a conjecture into separate conjuncts (using the & proof method) early in a proof often leads to repeating work on multiple conjuncts.
- To keep lemmas and theorems from disappearing (because they normalize to identities), make them immune.

When a proof gets stuck

- Be skeptical. Maybe your conjecture is not a theorem after all.
- In conjecture is a conditional, conjunction or implication, try the corresponding proof method.
- Try computing critical pairs between hypotheses and other rewrite rules.
- Use proof by cases on the test in an `if` in a rewrite rule.
- Display hypotheses to see if any are missing or are not ordered the way you expected.
- Look for a useful lemma to prove. But if not fruitful, use command `cancel` to remove this conjecture.
- LP automatically normalizes facts. Use command `show normal-form` *E* to see what happened to your fact *E*. Set trace level up to 6 to see which rewrite rules are applied in the normalization.

Getting Lost in the Proof

- `display`, `resume` and `history` can help find a place in the subgoal tree.
- `display *hyp` - to find your place in nested case analyses
- `display proof-status` displays the entire proof stack
- `display conjectures` *names* - the named conjectures
- `resume` shows just the current conjecture (normalized if the `proof-methods` include normalization)
- `history` *number* displays indented history, including LP-generated box and diamond lines.

Making Proofs Go Faster

Use `statistics` command to find out what is consuming a lot of time.

- If rewriting is costly,
  - immunize facts that you know are irreducible
  - deactivate rewrite rules needed only occasionally
  - make definitions passive and apply them manually
  - avoid big terms, especially with `ac` operators - If ordering is costly, put ordering constraints in the registry.
- If unification or critical pairing is costly, try to use smaller rule lists as arguments to `critical-pair` commands. Avoid computing critical pairs between rules that contain subterms such as $t_1\&t_2\&...\&t_n$ with multiple occurrences of the same `ac` operator.

Example - Simple Windowing System

- These are preliminary versions of traits that would be expanded as specifications (including interface parts) are developed.

```
Coordinate: trait
  introduces
    origin: -> Coord
    __-__: Coord, Coord -> Coord
  asserts \forall cd: Coord
    cd - cd == origin
Region(R): trait
  assumes Coordinate
  introduces
    __\in __: Coord, R -> Bool
    % cd \in r is true if point cd is in region r
    % Nothing assumed about shape or contiguity of regions
Displayable(T): trait
  assumes Coordinate
  includes Region(T)
  introduces
    __[__]: T, Coord -> Color
    % t[cd] represents appearance of object t at point cd
```

Proof obligations are easily discharged.

Example (Cont'd)

Define a window as an object composed of content and clipping regions, foreground and background colors and window identifier.

```
Window: trait
   assumes Coordinate
   includes Region, Displayable(W)
   asserts
     W tuple of cont, clip: R, fore, back: Color, id: WId
     \forall w: W, cd: Coord
       cd \in w == cd \in w.clip
       w[cd] == if cd \in w.cont then w.fore else w.back
   implies converts __[__], \in: Coord, W -> Bool
```

There are three proof obligations. Assumptions of `Coordinate` in `Region` and `Displayable` are syntactically discharged using `Window`'s assumption. The `converts` clause is discharged by LP without user assistance. Consistency - run completion procedure to search for inconsistency. Proves nothing.

Example (Cont'd)

Define a view as an object composed of windows at locations.

```
View: trait
  assumes Coordinate
  includes Window, Displayable(V)
  introduces
    emptyV: -> V
    addW: V, Coord, W -> V
    __\in __: W, V -> Bool
    inW: V, WId, Coord -> Bool
  asserts
    V generated by emptyV, addW
    forall cd, cd': Coord, v: V, w, w': W, wid: WId
      ~(cd \in emptyV)
      cd \in addW(v, cd', w) == ((cd - cd') \in w)
                                  \/ (cd \in v)
      ~(w \in emptyV)
      w \in addW(v, cd', w') == (w.id = w'.id) \/ (w \in v)
      addW(v, cd', w)[cd] == if (cd - cd') \in w then
         w[cd - cd'] else v[cd]
      % In view only if in a window, offset by origin
      ~inW(emptyV, wid, cd)
      inW(addW(v, cd, w), wid, cd') == (w.id = wid
           /\ (cd - cd') \in w) \/ inW(v, wid, cd')
```


Example(Cont'd)

```
  implies
    \forall cd, cd': Coord, v, v': V, w: W
      % Adding a new window does not affect the appearance
      % of parts of the view lying outside the window
        ~inW(addW(v, cd, w), w.id, cd') =>
           addW(v, cd, w)[cd'] = v[cd']
      % Appearance within a newly added window is
      % independent of the view to which it is added
        inW(addW(v,cd',w), w.id, cd) =>
           addW(v, cd', w)[cd] = addW(v', cd', w)[cd]
  converts inW, \in: Coord, V -> Bool, \in: W, V -> Bool,
           __[__]: V, Coord -> Color
  exempting \forall cd: Coord
           emptyV[cd]
```

Trying to prove explicit equations in `implies` clause of `View`.. LP reduces the conjecture to

```
    if (cdc' - cdc) \in wc.clip
      then if (cdc' - cdc) \in wc.cont
          then wc.fore else wc.back
      else vc[cdc']
    == vc[cd']
```

and reduces the assumed hypothesis of implication to

```
    ~((cdc - cdc') \in wc.clip)
```


Example (Cont'd)

Discover that we have written `cd - cd'` in second equation for `inW` in `View`. Change that to

```
      inW(addW(v, cd', w), wid, cd) == (w.id = wid
           /\ (cd - cd') \in w) \/ inW(v, wid, cd)
```

and everything works fine. The second conjecture reduces to

```
  if (cdc - cdc') \in wc.clip
    then if (cdc - cdc') \in wc.cont
         then wc.fore else wc.back
    else v[cdc]
  ==
  if (cdc - cdc') \in wc.clip
    then if (cdc -cdc') \in wc.cont
         then wc.fore else wc.back
    else v'[cdc]
```

We reduce this to `vc[cdc] == v'[cdc]`. `v'` is a variable, `vc` is a constant... Hmm... Turns out, we assumed that no view should contain two windows with the same `id` but our spec does not guarantee it!

Example (Cont'd)

So, try to add `numW` to `View` spec:

```
  numW: V, WId -> Nat

  numW(emptyV, wid) == 0
  numW(addW(v, cd', w), wid) ==
     numW(v, wid) + (if w.id = wid then 1 else 0)
  numW(v, wid) <= 1  % New invariant
```

But now, when we run LP completion procedure, we get an inconsistency.

Etc., until we are done.

food for slide eater

food for slide eater

food for slide eater

food for slide eater