Overview of Larch/LP

- Design principles for Larch/LP
- Larch example
- Proof obligations
- Term Rewriting
- Some (related) commands
- (time permitting) Extended example

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.

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.

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);
```


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

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 $S \sqsubset^{+} T$.

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.

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)
  ..
```


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. 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).

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 $t_1 \mid ... \mid t_n$. 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 $t_1 \Rightarrow t_2$. 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`.

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)
  ..
```

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 $t_1 ... \ t_n$, s.t. $t = t_1 \hookrightarrow_R ... \hookrightarrow_R t_n = u$.
- 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 $f(a, b) \rightarrow f(b, a)$, 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 $R \cup \{l \rightarrow r\}$

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 $l_1 \rightarrow r_1$ and $l_2 \rightarrow r_2$ 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.

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

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