Overview

- What is theorem proving
- Some theorem provers
- Industrial-size case studies
- An introductory example
- Design principles of Larch
- Larch examples
- Proof obligations
- Term-rewriting
- Some commands
- Extended example
- Larch vs PVS

Theorem Proving

Specify

- The system (at some suitable level of abstraction)
- A required property
- The assumptions
- Necessary background theories as formulas in a single logic

Prove that

background + *assumptions* + *system* $\models$ *requirement*

Variation

- Prove that an implementation is a refinement of a specification

Classical commutative diagram, or theory interpretation

Example

Want to specify a phone book with the following properties:

- It should store phone numbers in a city
- Possible to retrieve a number given a name
- Possible to add and delete entries from a phone book.

Will use functions `FindPhone`, `AddPhone` and `DeletePhone`.

Also, will use putative theorems ("formal challenges") to show that our spec are reasonable, e.g. "if I add a name `nm` with phone number `pn` to a phone book and look up the name `nm`, I should get back the phone number `pn`.

Phone Book in Larch

- 2-tier specification
- Combines axiomatic and algebraic specifications
  - Interface specification (great for checking correctness of code vs spec). Has features of prog. lang. (exceptions, etc.)
  - Theorem-proving only works for traits (algebraic specs)

Interface:

```
   phone_book is data type based on B from PhoneBook
  emptybook=proc() returns (b: phone_book)
    ensures b'= emp \AND new(b)
  FindPhone=proc(b: phone_book, nm: name)
                              returns(pn: phone_nums)
   requires isin(b, nm)
   ensures ph'= find(b, nm)
 AddPhone=proc (b: phone_book, nm: name, pn: phone_nums)
   requires ~isin(b, nm)
   modifies (b)
   ensures b'= add(b,nm,pn)
 DeletePhone=proc(b: phone_book, nm:name)
   requires isin(b, nm)
   modifies(b)
   ensures b' = rem(b,nm)
end phone_book
```

Phone Book (Cont'd)

```
PhoneBook: trait
 introduces
   emp:  -> B
   add:  B,N,P -> B
   rem:  B,N -> B
   find: B,N -> P
   isin: B,N -> Bool

 asserts
   B generated by (emp,add)
   B partitioned by (find, isin)
   for all (b:B, n,n1:N, p:P)
     rem(add(b,n,p),n1) == if n=n1 then b else
          add(rem(b,n1),n,p)
     find(add(b,n,p),n1) == if n=n1 then p else
          find(b,n1)
     isin(emp, n) == false
     isin(add(b,n,p),n1) == (n=n1) \OR isin(b,n1)

 implies
   converts(rem,find,isin)
        exempting (rem(emp),find(emp))
```


Phone Book in PVS - Error!

```
   phone1: THEORY
  BEGIN
    N: TYPE              % names
    P: TYPE              % phone numbers
    B: TYPE = [N->P]     % phone books

    n0: P
    emptybook: B
    emptyax: AXIOM FORALL (nm: N):
        emptybook(nm) = n0
    FindPhone: [B, N -> P]
    Findax: AXIOM FORALL (bk: B), (nm: N):
        FindPhone(bk, nm) = bk(nm)
    AddPhone: [B, N, P -> B]
    Addax: AXIOM FORALL (bk: B), (nm: N), (pn: P):
      AddPhone(bk, nm, pn) = bk WITH [(nm) := pn]
   DelPhone: [B, N -> B]
   Delax: AXIOM FORALL (bk: B), (nm: N):
       DelPhone(bk, nm) = bk WITH [(nm) := n0]

  % and now our "check"
  FindAdd: CONJECTURE FORALL (bk: B), (nm: N), (pn: P):
     FindPhone(AddPhone(bk, nm, pn), nm) = pn
  DelAdd: CONJECTURE FORALL (bk: B), (nm: N), (pn: P):
     DelPhone(AddPhone(bk, nm, pn), nm) = bk
END phone1
```

Phone Book using Sets (PVS)

```
phone_3 : THEORY
BEGIN
  N: TYPE                       % names
  P: TYPE                       % phone numbers
  B: TYPE = [N -> setof[P]] % phone books
  nm, x: VAR N
  pn: VAR P
  bk: VAR B

  emptybook(nm): setof[P] = emptyset[P]
  FindPhone(bk, nm): setof[P] = bk(nm)
  AddPhone(bk, nm, pn): B = bk WITH [(nm) :=
      add(pn, bk(nm))]
  DelPhone(bk,nm): B = bk WITH [(nm) :=
      emptyset[P]]
  DelPhoneNum(bk,nm,pn): B = bk WITH [(nm) :=
      remove(pn, bk(nm))]

  FindAdd: CONJECTURE member(pn,
      FindPhone(AddPhone(bk, nm, pn), nm))
  DelAdd: CONJECTURE DelPhoneNum(AddPhone(bk,
      nm, pn), nm, pn) = DelPhoneNum(bk, nm, pn)
END phone_3
```


Techniques and Issues

How rich a logic?

- Easier to automate restricted logics (e.g., unquantified)
- But can make for awkward specifications

Interactive guidance vs. automation

- Really, need both; the issue is *balance*

Automation: decision procedures.

- Propositional calculus (NP complete at least)
- Equality over uninterpreted function symbols
- Linear arithmetic over rationals and integers
- Functional updates (arrays, stores), i.e.

f **with** $[(x := y](z) \stackrel{\text{def}}{=}$ if $z = x$ **then** y **else** $f(z)$

Techniques and Issues

Automation: Rewriting
- Provides decision procedures if rules are finite terminating
- More often used heuristically
- Hard to tell when theorem is wrong rather than we failed to find a proof

Automation: Heuristics
- First order: resolution (like in Prolog), etc.
- Induction can help in proofs over infinite domains
- Difficult to interpret failure

Pros and Cons of Theorem Proving

- Specialized methods can be very effective on a limited domain (e.g., boolean equivalence checking in CAD tools)
- General methods can state and prove any true property given enough time, skill and patience
- Often require too much time, skill and patience
- Many theorem provers are poorly matched to the requirements of formal methods
  1) lack adequate automation
  2) do not support civilized specification language, nor theories needed for computer science
  3) do not fail gracefully
     - false conjecture or inadequate heuristic?
     - do not help pinpoint error

State of the Art Theorem Provers

User-guided automatic deduction tools

- Systems are guided by a sequence of lemmas and definitions but each theorem is proved automatically using builtin heuristics for induction, lemma-driven rewriting and simplification

ACL2 [Kaufmann and Moore 1995], Eves [Cragen et al 1988], LP [Garland and Guttag 1988], Nqthm [Boyer and Moore 1979], Reve [Lescanne 1983], RRL [Kapur and Musser 1987]

- Nqthm was used to check a proof of Godel's first incompleteness theorem

Proof checkers

- Used to formalize and verify hard problems in mathematics and in program verification
- Coq [Cornes et al. 1995], HOL [Gordon 1987], LEGO [Luo and Pollack 1992], LCF [Gordon et al. 1979], Nuprl [Constable et al. 1986]

State of the Art Theorem Provers

Combination provers

- Analytica [Clarke and Zhao 1993] - combines theorem proving with symbolic algebra system Mathematica. Proved some hard number-theoretic problems
- PVS and STep combine decision procedures and model checking with interactive proof
- PVS was used to verify a number of hardware designs and reactive, real-time and fault-tolerant algorithms

Theorem Proving - Industrial Uses

SRT division algorithm (1995, Clarke, German and Zhao)

- Used automatic theorem-proving techniques based on symbolic algebraic manipulation to prove the correctness of an SRT division algorithm line the one in the Pentium
- Verification method runs automatically and count have detected the error in the Pentium, caused by a faulty quotient digit selection table
- SRI's PVS was also used on this same example (1996)

Theorem Proving - Uses (Cont'd)

Processor Designs

- Verity verification tool is widely used within IBM in design of PowerPC and System/390
- Tool can handle entire processor designs containing millions of transistors (when applied in hierarchical manner)
- Can model the functional behavior of a hardware system as a boolean state transition function at register transfer level, gate level, or transistor level
- Use BDDs to check the equivalence of the state transition functions at different design levels

Theorem Proving - Examples (Cont'd)

Motorola 68020 (Boyer and Yu, 1991-95)

- Constructed an Nqthm specification of Motorola 68020 microprocessor
- Specification included 80 - Used specification to prove correctness of many binary machine code programs produced by commercial compilers from Lisp, C, Ada
- Verified MC68020 binary code produced by gcc compiler for 21 of 22 programs in the Berkeley string library

AAMP5 (1993-95, Srivas, Stanford and Miller, Rockwell)

- specified and verified Collins Commercial Avionics AAMP5 microprocessor
- used PVS to specify 108 of the 209 AAMP5 instructions and verified the microcode for 11 representative instructions

Theorem Proving - Examples (Cont'd)

AMD5K86 (1995, Moore and Kaufmann of Computational Logic, Inc. and Lynch of Advanced Micro Devices, Inc)

- Proved correctness of Lynch's microcode for floating point division on the AMDK86
- Started from informal proof of correctness
- Formalized proof in the ACL2 logic and checked with ACL2 mechanical theorem prover
- Erros were found in informal "proof" but the microcode was correct
- Effort took nine weeks

Theorem Proving - Examples (Cont'd)

Motorola CAP (1992-96 Brock of Computational Logic, Inc.)

- Developed an ACL2 specification of the entire Motorola Complex Arithmetic Processor (CAP)
- Most complicated microprocessor yet formalized
- Three state pipeline, six independent memories, four multiplier-accumulators, over 250 programmer-visible registers
- Instruction set allows simultaneous modification of over 100 registers in a single instruction
- Used ACL2 to verify binary microcode for several digital signal processing algorithms

For More Information on Larch/LP

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
   `http:/www.cs/ chechik/courses00/csc2108/LPdoc`

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.

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 divisionmeo plete?

- 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.

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)
```

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 `PriorityQueue` 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)`.

Example: `PriorityQueue`

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

`PriorityQueue`
Check implications
Use assertions of `PriorityQueue` and 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

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`.

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 the 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 (later this section)

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`

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)
  ..
```

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
```


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 $S \sqsubset^{+} T$.

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)

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 + 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 $l \to r$ 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 $t_1 \dots\ t_n$, s.t. $t = t_1 \hookrightarrow_R \dots \hookrightarrow_R t_n = u$.
- 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 (we will look at this later).

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).

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

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`

Checking Consistency

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

`complete`

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

But

- many equational theories cannot be com- ed at all

some cannot be completed in an accept- amount of time and space

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.

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).

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.

Forward Inference (Cont'd)

- Computation of *critical pairs* and Knuth-Bendix *completion procedure*. 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`.

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 ∪ y` 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`.

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.

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.

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.mCo mand

`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)
  ..
```


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.
```

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'
```


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.
```

How Hard Is It to Prove Equations?

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

```
resume by induction on c
```

When proving that `PriorityQueue` implies `e ∈ q => ~(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
```


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`

Contrary to "Debugging LSL Specs", nothing is checked. Instead, `partitioned by` results in a universal-existential axiom, expressed as a deduction rule:

```
declare sorts E, S
declare operator ∈:  E, S -> Bool
declare variables e:  E, x, y:  S
assert when (∀ e), 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 $x == x \cup x$.

For example,

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


Using `Generated By`

`generated by` 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
...
```


Extended Example

```
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
```

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 4)
- 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. 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.

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