CSCI.6962/4962 Software Verification— Fundamental Proof Methods in Computer Science (Arkoudas and Musser)—Chapter 3.9-3.11

Instructor: Carlos Varela
Rensselaer Polytechnic Institute
Spring 2018

Proving Equalities

Goal: to become familiar with *equational proofs* as a type of inference.

- *numeric equations*
- *equality chaining*
- *terms and sentences as trees*
- *logic behind equality chaining*
- *sample proofs*
- *mathematical induction*
- list equations
- polymorphic datatypes
- ground terms
- top-down proof development

List equations

Consider lists of natural numbers:

```
datatype N-List := nil | (:: N N-List)
```

This defines a datatype `N-List` with two constructors: `nil` and `::` (cons). Here are a few ground terms of this sort:

```
nil
```

```
(one :: nil)
```

```
(zero :: nil)
```

```
(zero :: S zero :: S S zero :: S S S zero :: nil)
```

```
(one :: zero :: nil)
```

```
(one :: zero :: S one :: nil)
```

We interpret

- `nil` as the empty list (that contains no elements), and
- $(x :: L)$ as the list whose first element is x and whose remaining elements, if any, are those of list L.

Concatenating lists

`(L1 join L2)` is a list that begins with the elements of list `L1`, in the same order as in `L1`, followed by the elements of `L2`, in the same order as in `L2`.

```
declare join: [N-List N-List] -> N-List [++]
```

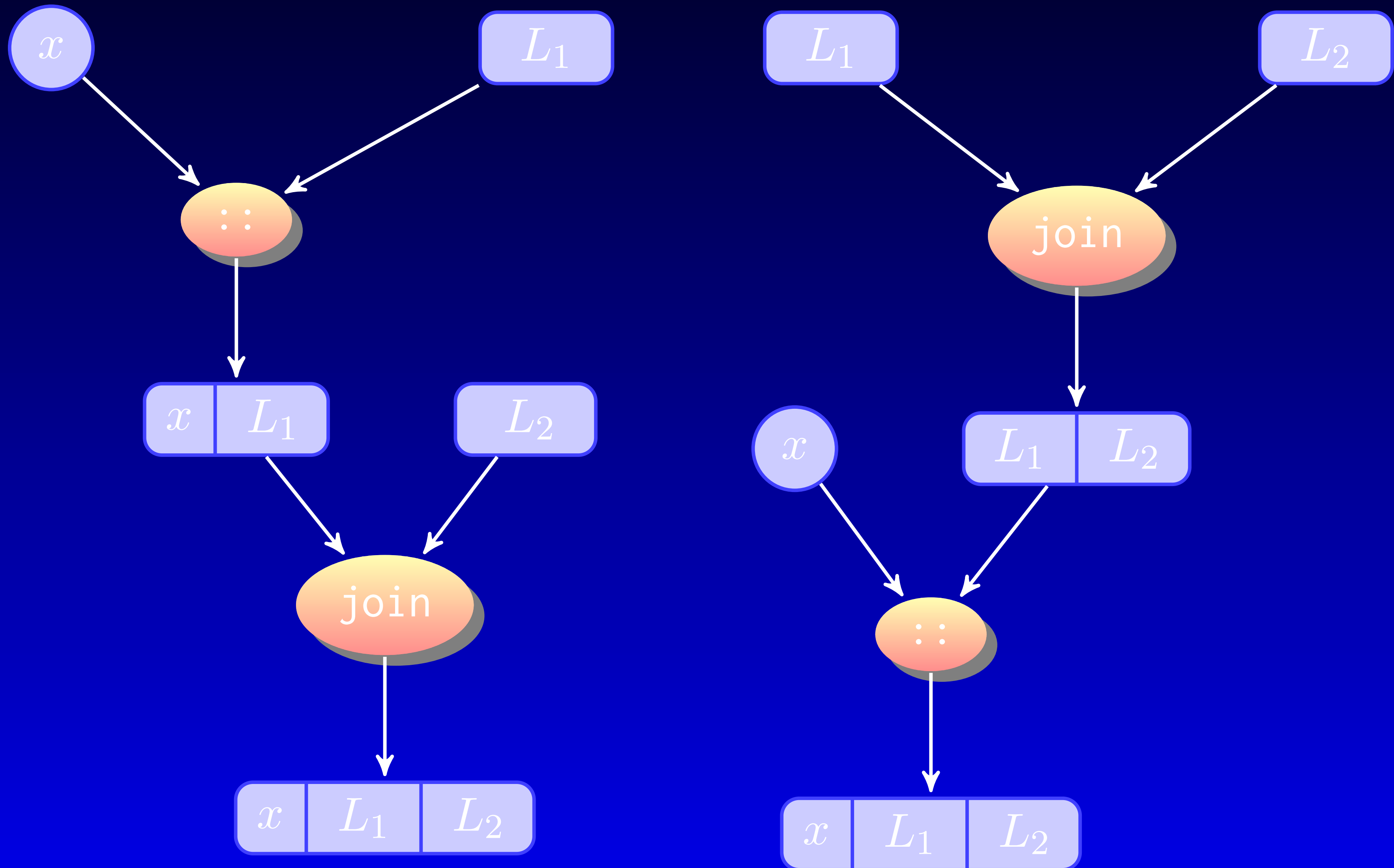
We now introduce two axioms that define concatenation:

```
assert left-empty := (forall L . nil ++ L = L)
```

```
assert left-nonempty := (forall x L1 L2 . x::L1 ++ L2 = x::(L1 ++ L2))
```

Setting the precedence of `::` to bind tighter than ++ (to make sure, e.g., that `(x::L1 ++ L2)` is understood as the join of `x::L1` and `L2` rather than the result of consing `x` onto the join of `L1` and `L2`).

Second concatenation axiom (`left-nonempty`)



Checking axioms are correct

Checking the ++ axioms with a few ground term inputs:

```
> (!chain [(one::zero::nil ++ one::zero::S one::nil)
          = (one::(zero::nil ++ one::zero::S one::nil)) [left-nonempty]
          = (one::zero::(nil ++ one::zero::S one::nil)) [left-nonempty]
          = (one::zero::one::zero::S one::nil)          [left-empty]])

Theorem: (= (join (:: one
                      (:: zero nil))
                  (:: one
                      (:: zero
                          (:: (S one)
                              nil))))
            (:: one
                (:: zero
                    (:: one
                        (:: zero
                            (:: (S one)
                                nil))))))
```

Proving simple properties of list concatenation

```
define right-empty := (forall L . L ++ nil = L)
```

We can use methods to prove this property for the empty list (`nil`) and for a composed list (`x::L`)):

```
define (join-nil-base) :=
  conclude (nil ++ nil = nil)
    (!chain [(nil ++ nil) = nil [left-empty]])

define (join-nil-step IH x L) :=
  conclude (x::L ++ nil = x::L)
    (!chain [(x::L ++ nil)
           = (x::(L ++ nil))      [left-nonempty]
           = (x :: L)             [IH]])
```

Proving simple properties of list concatenation

Using these methods, we can now prove `right-empty` for lists of arbitrary (finite) size:

```
> define p0 := (!join-nil-base)

Theorem: (= (join nil nil)
            nil)

Sentence p0 defined.

> define p1 :=
    pick-any x
      (!join-nil-step p0 x nil)

Theorem: (forall ?x:N
           (= (join (:: ?x:N nil)
                    nil)
              (:: ?x:N nil)))

Sentence p1 defined.
```

Proving simple properties of list concatenation

For a list of size 2:

```

> define p2 :=
    pick-any x:N y:N
      (!join-nil-step p1 x (y::nil))

Theorem: (forall ?x:N
           (forall ?y:N
             (= (join (:: ?x:N
                          (:: ?y:N nil))
                      nil)
                (:: ?x:N
                    (:: ?y:N nil)))))

Sentence p2 defined.
```

and so on.

Proof using mathematical induction

The pattern of these proofs suggests the following proof in full generality by induction:

```
> by-induction right-empty {
    nil => (!join-nil-base)
  | (h::t) => let {IH := (t ++ nil = t)}
                (!join-nil-step IH h t)
  }

Theorem: (forall ?L:N-List
           (= (join ?L:N-List nil)
              ?L:N-List))
```

Alternative proof

We can also write the proofs of the `nil` and `::` cases inline:

```
> by-induction right-empty {
    nil => (!chain [(nil ++ nil) = nil [left-empty]])
  | (h::t) =>
      let {IH := (t ++ nil = t)}
        conclude (h::t ++ nil = h::t)
          (!chain [(h::t ++ nil)
                 = (h :: (t ++ nil))   [left-nonempty]
                 = (h::t)              [IH]])
  }

Theorem: (forall ?L:N-List
           (= (join ?L:N-List nil)
              ?L:N-List))
```

Mathematical induction for lists of natural numbers

To prove $\forall\, L \,.\, P(L)$ where L ranges over lists of natural numbers, it suffices to prove:

1. *Basis case*: $P(\texttt{nil})$.
2. *Induction step*: $\forall\, L \,.\, P(L) \Rightarrow \forall\, x \,.\, P(x::L)$.

In the induction step, the antecedent assumption $P(L)$ is called the *induction hypothesis*, and x ranges over natural numbers.

`by-induction` **proof form**

`by-induction` adapts to `datatype` definitions according to the given constructors. Thus, from

```
datatype N-List := nil | (:: N N-List),
```

`by-induction` deduces that it must expect clauses, one corresponding to `nil` and one corresponding to `::`:

- The `nil` case is called a *basis case* because `nil` is an irreflexive constructor
- The `::` case, since `::` does take an argument of the datatype being defined, corresponds to an *induction step* (or "inductive step"), and `by-induction` temporarily assumes an appropriate inductive hypothesis for the duration of the proof given in that clause.

Polymorphic datatypes

None of the proofs or axioms so far depended on the sort of the list elements.

We can thus use the polymorphic definition of lists:

```
datatype (List S) := nil | (:: S (List S))

declare join: (S) [(List S) (List S)] -> (List S) [++]
```

We can now give a polymorphic definition of `join` as follows:

```
assert left-empty    := (forall L . nil ++ L = L)
assert left-nonempty := (forall x L1 L2 . x::L1 ++ L2 =  x::(L1 ++ L2))
```

Polymorphic datatypes

Since there are no ground terms, the proof works for lists of any sort `'S`:

```
define right-empty := (forall L . L ++ nil = L)

> by-induction right-empty {
    nil => (!chain [(nil ++ nil) = nil    [left-empty]])
  | (L as (h::t)) =>
      let {IH := (t ++ nil = t)}
        conclude (h::t ++ nil = h::t)
          (!chain [(h::t ++ nil)
                 = (h :: (t ++ nil))      [left-nonempty]
                 = L                      [IH]])
  }

Theorem: (forall ?L:(List 'S)
           (= (join ?L:(List 'S)
                    nil:(List 'S))
              ?L:(List 'S)))
```

Mathematical induction for lists over sort S

To prove $\forall\, L \,.\, P(L)$ where L ranges over lists of sort S, it suffices to prove:

1. *Basis case*: $P(\texttt{nil})$.
2. *Induction step*: $\forall\, L \,.\, P(L) \Rightarrow \forall\, x \,.\, P(x::L)$.

As before, the antecedent assumption $P(L)$ in the induction step is called the *induction hypothesis*; the variable x is of sort S.

More list concatenation properties

So let's next consider whether our list `join` function is associative:

```
define join-associative :=
  (forall L0 L1 L2 . (L0 ++  L1) ++ L2 = L0 ++ (L1 ++ L2))
```

Proof of associativity of list concatenation

```
by-induction join-associative {
  nil =>
    pick-any L1 L2
      (!chain [((nil ++ L1) ++ L2)
           --> (L1 ++ L2)              [left-empty]
           <-- (nil ++ (L1 ++ L2))   [left-empty]])
| (L as (h::t)) =>
    let {IH := (forall L1 L2 . (t ++ L1) ++ L2 = t ++ (L1 ++ L2))}
      conclude (forall L1 L2 . (L ++ L1) ++ L2 = L ++ (L1 ++ L2))
        pick-any L1 L2
          (!chain
            [((h::t ++ L1) ++ L2)
         --> ((h::(t ++ L1)) ++ L2)  [left-nonempty]
         --> (h::((t ++ L1) ++ L2))  [left-nonempty]
         --> (h::(t ++ (L1 ++ L2)))  [IH]
         <-- (h::t  ++ (L1 ++ L2))   [left-nonempty]])
}
```

Evaluation of ground terms

- How do we know that the two axioms we have given for `Plus` represent binary addition function on the natural numbers?

```
assert right-zero := (forall n . n + zero = n)

assert right-nonzero := (forall n m . n + S m = S (n + m))
```

- We can "test" the axioms on various "inputs," e.g., we can prove:

  `(S S S zero Plus S S zero = S S S S S zero).`

- Can we generalize the proof generation to any two natural numbers (represented by canonical terms)?

Evaluation of ground terms

```
define (derive-plus a b) :=
  match b {
    zero => (!chain [(a + b) = a [right-zero]])
  | (S k) => match (!derive-plus a k) {
               ((a Plus k) = sum) =>
                 (!chain [(a + S k)
                        = (S (a + k))    [right-nonzero]
                        = (S sum)        [(a + k = sum)]])
             }
  }
```

The `derive-plus` method works for `Plus` by matching `b`.

- If it is `zero`, then the axiom `right-zero` is used to derive the identity `(a + zero = a)`.
- Otherwise, if `b` is of the form `(S k)` for some canonical term `k`, then we recursively invoke the method on `a` and `k`.

Evaluation of ground terms

We can now test as many inputs as we like:

```
> (!derive-plus (S zero) (S zero))

Theorem: (= (Plus (S zero)
                  (S zero))
            (S (S zero)))

> (!derive-plus (S S S zero) (S S zero))

Theorem: (= (Plus (S (S (S zero)))
                  (S (S zero)))
            (S (S (S (S (S zero))))))
```

Athena's `eval` built-in procedure

`eval` takes any term t, containing arbitrary function symbols, and attempts to reduce it to a canonical form by using the various rewrite rules that have been asserted as axioms.

```
> (eval (S zero + S zero))

Term: (S (S zero))

> let {three := (S S S zero)}
    (eval (three + three))

Term: (S (S (S (S (S (S zero))))))
```

The built-in precedence of `eval` is very low, so these can be written:

```
(eval S zero + S zero)
let {three := (S S S zero)}
    (eval three + three)
```

Athena's `eval` built-in procedure

A few more examples:

```
> (eval zero::nil ++ S zero::nil)

Term: (:: zero
          (:: (S zero)
              nil:(List N)))

> let {1|0|nil := (S zero :: zero :: nil);
       3*3|nil := ((S S S zero * S S S zero) :: nil)}
   (eval 1|0|nil ++ 3*3|nil)

Term: (:: (S zero)
          (:: zero
              (                     (S (S (S zero))))))))))
                  nil:(List N))))
```

Top-down proof development

`reverse` takes a list and returns a list with the elements in opposite order.

```
declare reverse: (S) [(List S)] -> (List S) [120 [(alist->clist int->nat)]]

assert* reverse-def :=
  [(reverse nil = nil)
   (reverse h::t = (reverse t) ++ h::nil)]

define [reverse-empty reverse-nonempty] := reverse-def
```

- The precedence of `reverse` is set higher (120) than that of `++`
- `[(alist->clist int->nat)]` allows `reverse` to accept as inputs Athena lists (in square-bracket notation), which are then converted to `::`-built terms, while converting integer numerals to natural numbers along the way, using `int->nat`.

Input expansion and output transformation

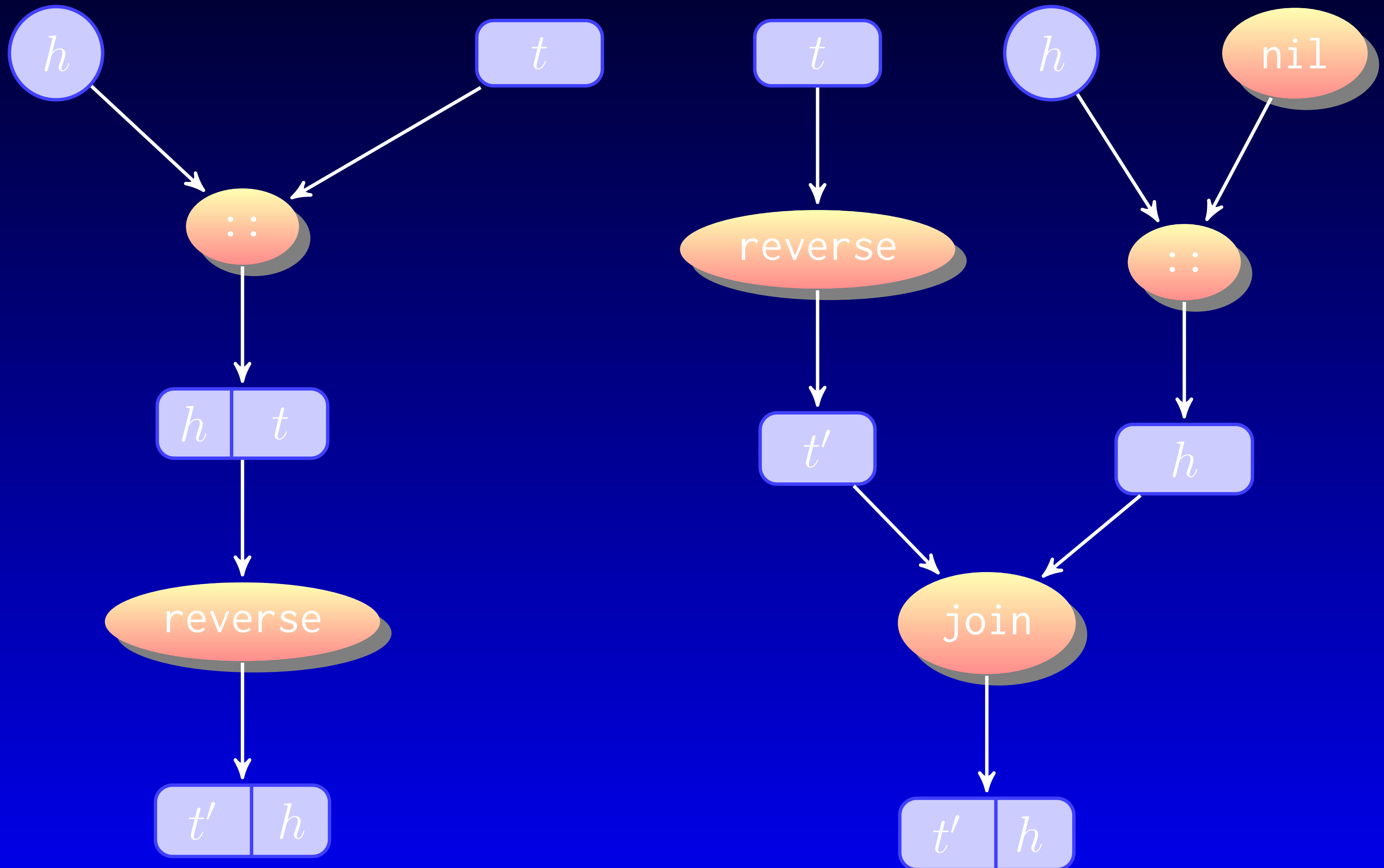
- `[(alist->clist int->nat)]` is an example of *input expansion*, so that we can apply `reverse` directly to an expression like `[1 2 3]`.
- Another mechanism that aids readability is *output transformation*, e.g., transforming the output of `eval` by converting lists built with `::` to square-bracket lists:

```
transform-output eval [(clist->alist nat->int)]

> (eval reverse [1 2 3 4 5 6])

List: [6 5 4 3 2 1]
```

Second reverse axiom (`reverse-nonempty`)



Proving the reverse of the reverse is the identity function

```
define reverse-reverse-theorem := (forall L .  reverse reverse L = L)
```

We begin a proof by induction and note that the basis case, `L = nil`, is quite easy.

```
by-induction reverse-reverse-theorem {
  nil => conclude (reverse reverse nil = nil)
           (!chain [(reverse reverse nil)
                --> (reverse nil)          [reverse-empty]
                --> nil                    [reverse-empty]])
| (L as (h::t)) => ...
}
```

Proving the reverse of the reverse is the identity function

But when we try to prove the induction step, we run into a problem:

```
by-induction reverse-reverse-theorem {
  nil =>
    conclude (reverse reverse nil = nil)
      (!chain [(reverse reverse nil)
           --> (reverse nil)          [reverse-empty]
           --> nil                    [reverse-empty]])
| (L as (h::t)) =>
    conclude (reverse reverse L = L) {
      let {IH := (reverse reverse t  = t)}
        (!chain [(reverse reverse h::t)
             --> (reverse ((reverse t) ++ h::nil)) [reverse-nonempty]
             --> ?                                 [??]
```

No `reverse` or ++ axioms or theorems apply. Neither does the inductive hypothesis.

Top-down proof development

There are two strategies to find a proof:

- *top-down* from a goal property we want to prove to conjecturing and proving other new properties as subgoals and then using them as lemmas
- *bottom-up* from axioms to goals

top-down is far more typical in practice than *bottom-up*. Top-down is goal-driven, whereas bottom-up must anticipate needed properties and there are infinitely many theorems to state!

Proving the reverse of the reverse is the identity function

We were staring at

`(reverse ((reverse t) ++ h::nil)),`

a term that we somehow need to prove equal to `(h::t)`.

- Intuitively, we know that the last element of `((reverse t) ++ h::nil)` is `h`, and therefore `h` should be the first element of the reversed list.
- So the result is of the form `(h :: ···)`, and thus if we can fill in that ··· with `t`, we'll be done.
- But, in fact, without the `h` on the end, `((reverse t) ++ h::nil)` is just ++'s first argument, `(reverse t)`.
- And by the induction hypothesis, `(reverse reverse t)` is `t`!

Proving the reverse of the reverse is the identity function

To replace this intuitive argument with a real proof, we write the key step as a formal sentence and try to prove it:

```
define reverse-join :=
  (forall L x . reverse ((reverse L) ++ x::nil) = x::L)

by-induction reverse-join {
  nil => pick-any x
           (!chain [(reverse ((reverse nil) ++ x::nil))
                --> (reverse (nil ++ x::nil))       [reverse-empty]
                --> (reverse x::nil)                [left-empty]
                --> ((reverse nil) ++ x::nil)       [reverse-nonempty]
                --> (nil ++ x::nil)                 [reverse-empty]
                --> (x::nil)                        [left-empty]])
| (L as (h::t)) => ...
}
```

Proving the reverse of the reverse is the identity function

But the inductive case leads us to a dead end:

```
| (L as (h::t)) =>
   conclude (forall x . reverse ((reverse L) ++ x::nil) = x::L)
     let {IH := (forall x . reverse ((reverse t) ++ x::nil) = x::t)}
       pick-any x
         (!chain
           [(reverse ((reverse h::t) ++ x::nil))
        --> (reverse (((reverse t) ++ h::nil) ++ x::nil)) [reverse-nonempty]
        --> ?                                              [??]
            ...])
```

We get stuck again, without any axiom or theorem that can be applied, and neither can the induction hypothesis be applied (because of the extra `join` in the term we are trying to rewrite).

Proving the reverse of the reverse is the identity function

- We need to step back and try to simplify what we are trying to prove by *generalizing* it.
- A more general property gives us a *more general induction hypothesis*. Being more general, the induction hypothesis gives us more ammunition that we can use to advance the proof.
- We can make `reverse-join` more general by replacing `(reverse L)` by `L` on the left-hand side of the equation and compensating by the opposite replacement on the right-hand side (which is well-motivated by our intuitive belief in `reverse-reverse-theorem`, though we haven't yet proved it!):

```
define reverse-join-1 :=
  (forall L x . reverse (L ++ x::nil) = x::reverse L)
```

Proving the reverse of the reverse is the identity function

Now, we have no difficulty in proving this more general conjecture, first the base case:

```
by-induction reverse-join-1 {
  nil =>
    conclude (forall x . reverse (nil ++ x::nil) = x::reverse nil)
      pick-any x
        (!chain [(reverse (nil ++ x::nil))
            --> (reverse x::nil)                        [left-empty]
            --> (reverse nil ++ x::nil)                 [reverse-nonempty]
            --> (nil ++ x::nil)                         [reverse-empty]
            --> (x::nil)                                [left-empty]
            <-- (x::reverse nil)                        [reverse-empty]])
| (L as (h::t)) => ...
}
```

Proving the reverse of the reverse is the identity function

Then, the inductive case:

```
| (L as (h::t)) =>
    conclude (forall x . reverse (L ++ x::nil) = x::reverse L)
      let {ih := (forall x . reverse (t ++ x::nil) = x::reverse t)}
        pick-any x
          (!chain [(reverse (h::t ++ x::nil))
               --> (reverse h::(t ++ x::nil))        [left-nonempty]
               --> (reverse (t ++ x::nil) ++ h::nil) [reverse-nonempty]
               --> (x::(reverse t) ++ h::nil)        [ih]
               --> (x::(reverse t ++ h::nil))        [left-nonempty]
               <-- (x::(reverse h::t))              [reverse-nonempty]])
```

Proving the reverse of the reverse is the identity function

And, finally, with this lemma at hand, `reverse-reverse-theorem` can be easily derived:

```
by-induction reverse-reverse-theorem {
  nil => conclude (reverse reverse nil = nil)
           (!chain [(reverse reverse nil)
                --> (reverse nil)               [reverse-empty]
                --> nil                         [reverse-empty]])
| (list as (h::t)) =>
    conclude (reverse reverse list = list)
      let {ih := (reverse reverse t = t)}
        (!chain [(reverse reverse h::t)
             --> (reverse (reverse t ++ h::nil)) [reverse-nonempty]
             --> (h::reverse reverse t)          [reverse-join-1]
             --> (h::t)                          [ih]])
}
```

Summary of top-down approach

1. Start with the main goal and try to advance the proof by applying any axiom, previously proved theorem, or the inductive hypothesis. Do not go in circles!
2. If you run out of properties that can be applied, make one up! Form a conjecture, based on what is needed to move the current proof and on the ultimate goal.
3. If you are having trouble proving your conjecture, step back and see if you can generalize it. That may make it easier to prove, and if you succeed, you will still be able to use this more general property as a lemma in proving the original goal.
4. Although we succeeded in proving `reverse-reverse-theorem` after conjecturing and proving only one lemma, you will occasionally have to do this several times in order to complete a complex proof.