Interactive Formal Verification
2: Isabelle Theories

Tjark Weber
(Slides: Lawrence C Paulson)
Computer Laboratory
University of Cambridge

A Tiny Theory

```
theory BT imports Main begin

datatype 'a bt =
    Lf
  | Br 'a  "'a bt"  "'a bt"

fun reflect :: "'a bt => 'a bt" where
  "reflect Lf = Lf"
| "reflect (Br a t1 t2) = Br a (reflect t2) (reflect t1)"

lemma reflect_reflect_ident: "reflect (reflect t) = t"
  apply (induct t)
   apply auto
  done

end
```

name of the new theory

A Tiny Theory

```
theory BT imports Main begin

datatype 'a bt =
    Lf
  | Br 'a  "'a bt"  "'a bt"

fun reflect :: "'a bt => 'a bt" where
  "reflect Lf = Lf"
| "reflect (Br a t1 t2) = Br a (reflect t2) (reflect t1)"

lemma reflect_reflect_ident: "reflect (reflect t) = t"
  apply (induct t)
   apply auto
  done

end
```

A Tiny Theory

name of the new theory

the theory it builds upon

```
theory BT imports Main begin

datatype 'a bt =
    Lf
  | Br 'a  "'a bt"  "'a bt"

fun reflect :: "'a bt => 'a bt" where
  "reflect Lf = Lf"
| "reflect (Br a t1 t2) = Br a (reflect t2) (reflect t1)"

lemma reflect_reflect_ident: "reflect (reflect t) = t"
  apply (induct t)
   apply auto
  done

end
```

A Tiny Theory

name of the new theory

the theory it builds upon

declarations of types, constants, etc

```
theory BT imports Main begin

datatype 'a bt =
    Lf
  | Br 'a  "'a bt"  "'a bt"

fun reflect :: "'a bt => 'a bt" where
  "reflect Lf = Lf"
| "reflect (Br a t1 t2) = Br a (reflect t2) (reflect t1)"

lemma reflect_reflect_ident: "reflect (reflect t) = t"
  apply (induct t)
   apply auto
  done

end
```

A Tiny Theory

name of the new theory

```
theory BT imports Main begin

datatype 'a bt =
    Lf
  | Br 'a  "'a bt"  "'a bt"

fun reflect :: "'a bt => 'a bt" where
  "reflect Lf = Lf"
| "reflect (Br a t1 t2) = Br a (reflect t2) (reflect t1)"

lemma reflect_reflect_ident: "reflect (reflect t) = t"
  apply (induct t)
   apply auto
  done

end
```

the theory it builds upon

declarations of types, constants, etc

proving a theorem

Notes on Theory Structure

Notes on Theory Structure

- A theory can *import* any existing theories.

Notes on Theory Structure

- A theory can *import* any existing theories.
- Types, constants, etc., must be *declared before use*.

Notes on Theory Structure

- A theory can *import* any existing theories.
- Types, constants, etc., must be *declared before use*.
- The various declarations and proofs may otherwise appear in any order.

Notes on Theory Structure

- A theory can *import* any existing theories.
- Types, constants, etc., must be *declared before use*.
- The various declarations and proofs may otherwise appear in any order.
- Many declarations can be confined to *local scopes*.

Notes on Theory Structure

- A theory can *import* any existing theories.
- Types, constants, etc., must be *declared before use*.
- The various declarations and proofs may otherwise appear in any order.
- Many declarations can be confined to *local scopes*.
- A finished theory can be imported by others.

Some Fancy Type Declarations

```
typedecl loc  -- "an unspecified type of locations"

type_synonym val   = nat -- "values"
type_synonym state = "loc => val"
type_synonym aexp  = "state => val"
type_synonym bexp  = "state => bool"  -- "functions on states"

datatype
  com = SKIP
      | Assign loc aexp          ("_ :== _ " 60)
      | Semi   com com           ("_; _"  [60, 60] 10)
      | Cond   bexp com com      ("IF _ THEN _ ELSE _"  60)
      | While  bexp com          ("WHILE _ DO _"  60)
```

Some Fancy Type Declarations

```
typedecl loc  -- "an unspecified type of locations"

type_synonym val   = nat -- "values"
type_synonym state = "loc => val"
type_synonym aexp  = "state => val"
type_synonym bexp  = "state => bool"  -- "functions on states"

datatype
  com = SKIP
      | Assign loc aexp          ("_ :== _ " 60)
      | Semi   com com           ("_; _"  [60, 60] 10)
      | Cond   bexp com com      ("IF _ THEN _ ELSE _"  60)
      | While  bexp com          ("WHILE _ DO _"  60)
```

new basic types

Some Fancy Type Declarations

```
typedecl loc  -- "an unspecified type of locations"

type_synonym val   = nat -- "values"
type_synonym state = "loc => val"
type_synonym aexp  = "state => val"
type_synonym bexp  = "state => bool"  -- "functions on states"

datatype
  com = SKIP
      | Assign loc aexp         ("_ :== _ " 60)
      | Semi   com com          ("_; _"  [60, 60] 10)
      | Cond   bexp com com     ("IF _ THEN _ ELSE _"  60)
      | While  bexp com         ("WHILE _ DO _"  60)
```

new basic types

concrete syntax for commands

Some Fancy Type Declarations

```
typedecl loc  -- "an unspecified type of locations"

type_synonym val   = nat -- "values"
type_synonym state = "loc => val"
type_synonym aexp  = "state => val"
type_synonym bexp  = "state => bool"  -- "functions on states"

datatype
  com = SKIP
      | Assign loc aexp          ("_ :== _ " 60)
      | Semi   com com           ("_; _"  [60, 60] 10)
      | Cond   bexp com com      ("IF _ THEN _ ELSE _"  60)
      | While  bexp com          ("WHILE _ DO _"  60)
```

new basic types

concrete syntax for commands

recursive type of commands

Notes on Type Declarations

Notes on Type Declarations

- Type synonyms merely introduce *abbreviations*.

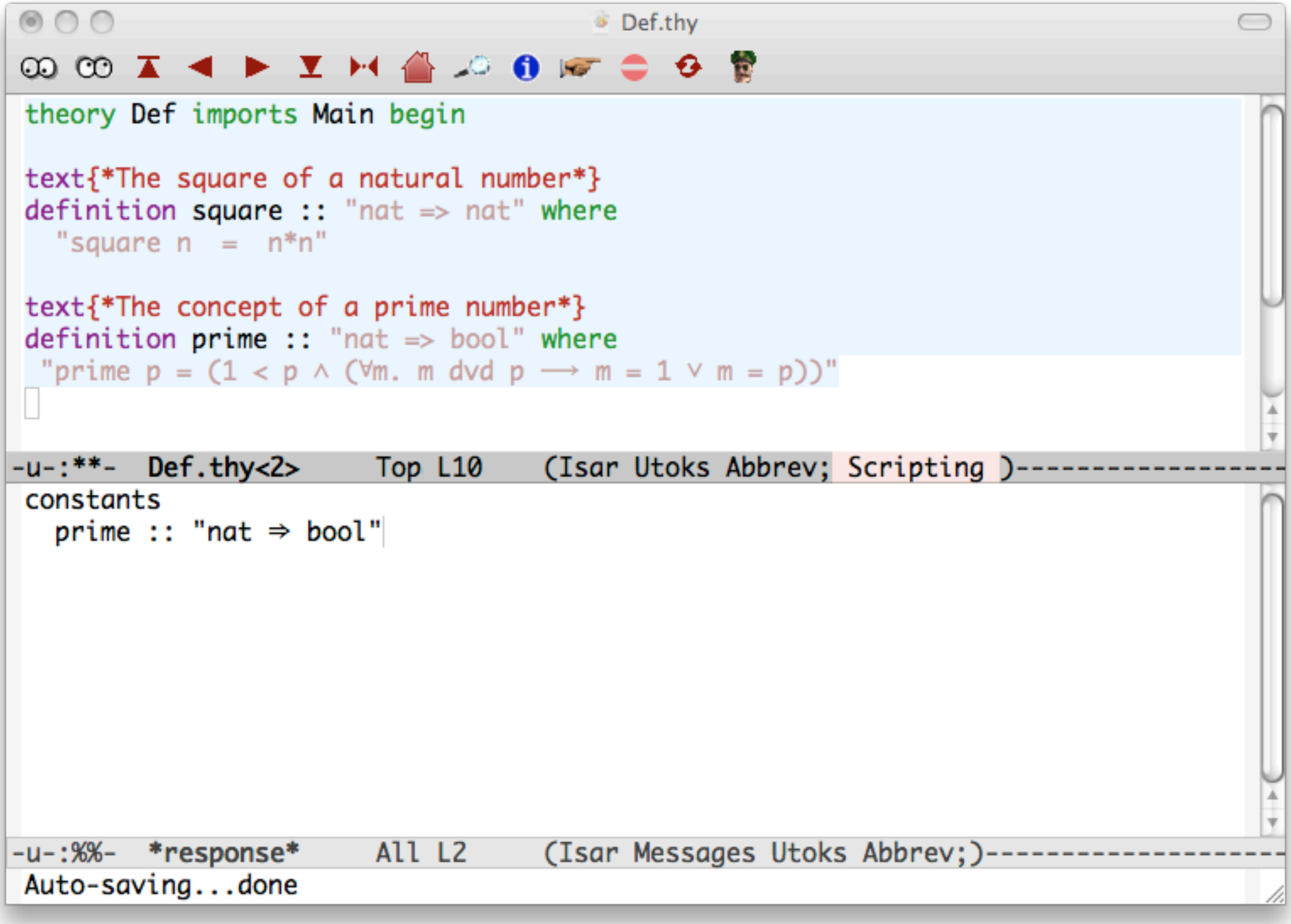
Notes on Type Declarations

- Type synonyms merely introduce *abbreviations*.
- Recursive data types are less general than in functional programming languages.
  - No recursion into the domain of a function.
  - Mutually recursive definitions can be tricky.

Notes on Type Declarations

- Type synonyms merely introduce *abbreviations*.
- Recursive data types are less general than in functional programming languages.
  - No recursion into the domain of a function.
  - Mutually recursive definitions can be tricky.
- Recursive types are equipped with proof methods for *induction* and *case analysis*.

Basic Constant Definitions



```
theory Def imports Main begin

text{*The square of a natural number*}
definition square :: "nat => nat" where
  "square n  =  n*n"

text{*The concept of a prime number*}
definition prime :: "nat => bool" where
 "prime p = (1 < p ∧ (∀m. m dvd p ⟶ m = 1 ∨ m = p))"

-u-:**-  Def.thy<2>     Top L10     (Isar Utoks Abbrev; Scripting )-----------------
constants
  prime :: "nat ⇒ bool"

-u-:%%-  *response*     All L2      (Isar Messages Utoks Abbrev;)------------------
Auto-saving...done
```

Notes on Constant Definitions

Notes on Constant Definitions

- Basic definitions are *not* recursive.

Notes on Constant Definitions

- Basic definitions are *not* recursive.
- Every variable on the right-hand side must also appear on the left.

Notes on Constant Definitions

- Basic definitions are *not* recursive.
- Every variable on the right-hand side must also appear on the left.
- In proofs, definitions are *not* expanded by default!
  - Defining the constant *C* to denote *t* yields the theorem *C*`_def`, asserting *C*=*t*.
  - Abbreviations can be declared through a separate mechanism.

Lists in Isabelle

Lists in Isabelle

- We illustrate data types and functions using a reduced Isabelle theory that lacks lists.

Lists in Isabelle

- We illustrate data types and functions using a reduced Isabelle theory that lacks lists.
- The standard Isabelle environment has a *comprehensive list library*:
  - Functions # (cons), `@` (append), `map`, `filter`, `nth`, `take`, `drop`, `takeWhile`, `dropWhile`, ...
  - Cases: (`case` xs `of` [] ⇒ [] | x#xs ⇒ ...)
  - Over 600 theorems!

List Induction Principle

List Induction Principle

To show $\varphi(xs)$, it suffices to show the *base case* and *inductive step*:

- $\varphi(\mathrm{Nil})$
- $\varphi(xs) \Rightarrow \varphi(\mathrm{Cons}(x,xs))$

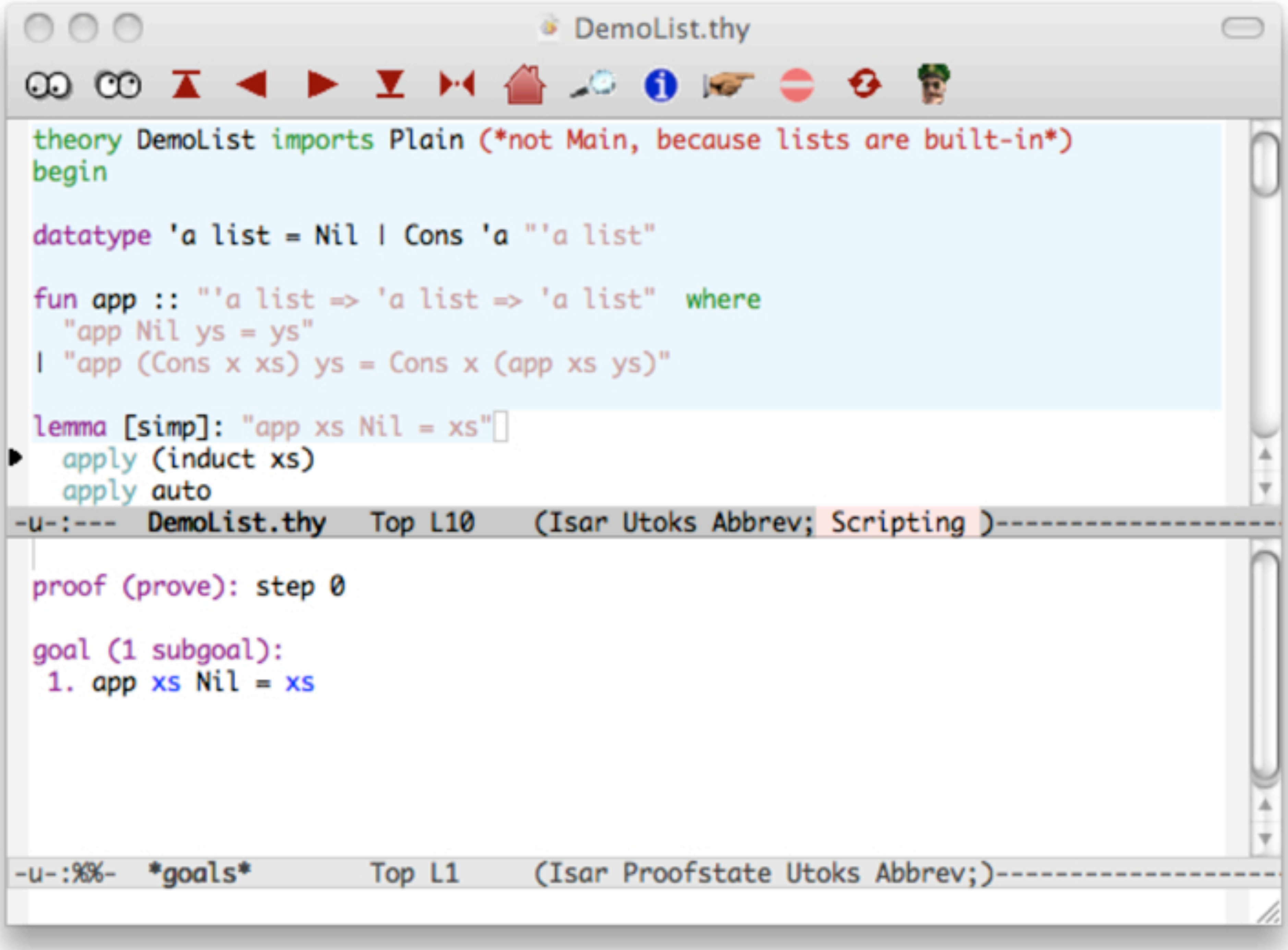
List Induction Principle

To show φ(*xs*), it suffices to show the *base case* and *inductive step*:

- φ(Nil)
- φ(*xs*) ⇒ φ(Cons(*x*,*xs*))

The principle of case analysis is similar, expressing that any list has one of the forms Nil or Cons(*x*,*xs*) (for some *x* and *xs*).

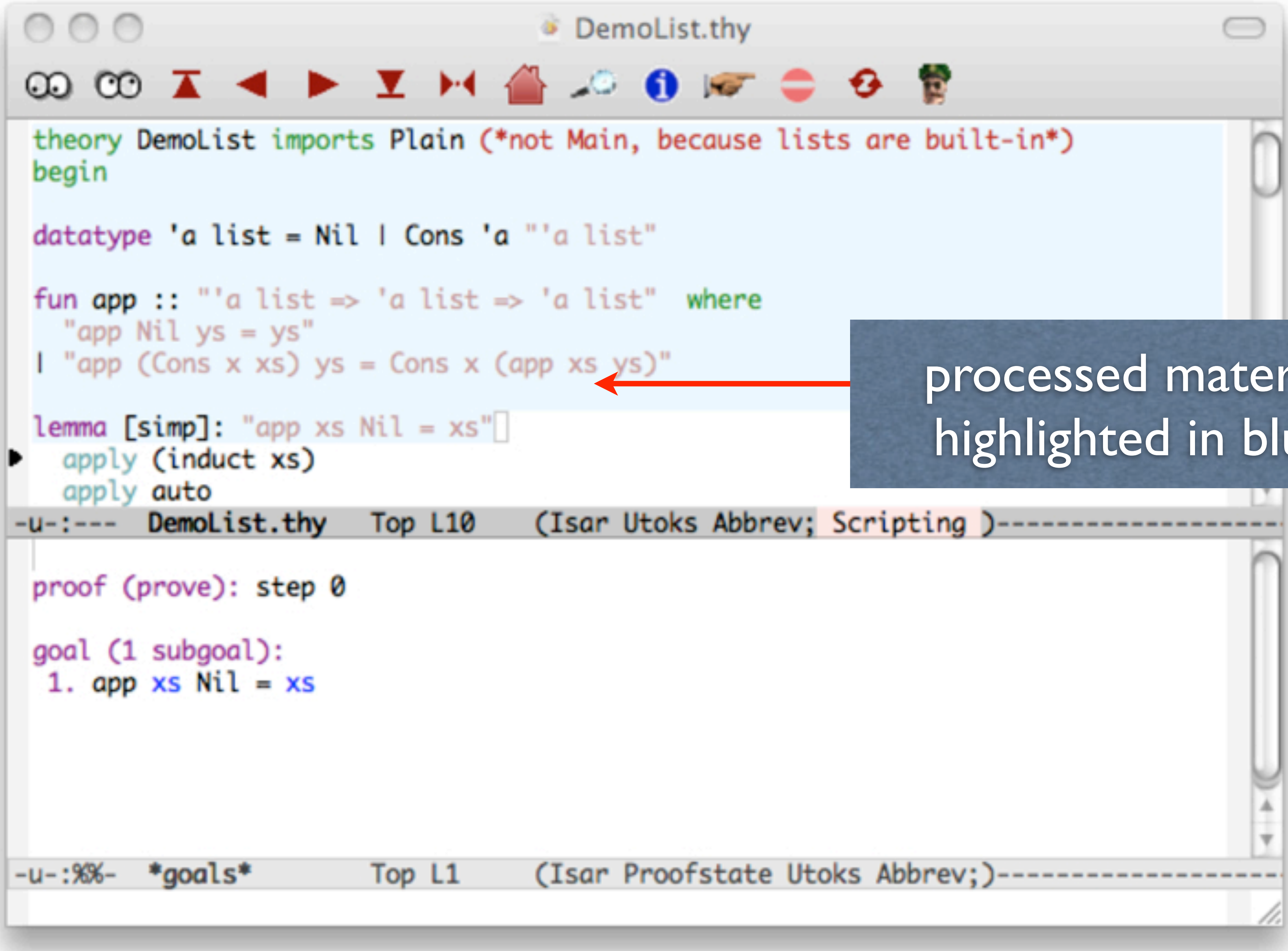
Proof General



```
theory DemoList imports Plain (*not Main, because lists are built-in*)
begin

datatype 'a list = Nil | Cons 'a "'a list"

fun app :: "'a list => 'a list => 'a list"  where
  "app Nil ys = ys"
| "app (Cons x xs) ys = Cons x (app xs ys)"

lemma [simp]: "app xs Nil = xs"
  apply (induct xs)
  apply auto
-u-:---  DemoList.thy    Top L10    (Isar Utoks Abbrev; Scripting )------------------
```

```
proof (prove): step 0

goal (1 subgoal):
 1. app xs Nil = xs

-u-:%%-  *goals*         Top L1     (Isar Proofstate Utoks Abbrev;)------------------
```

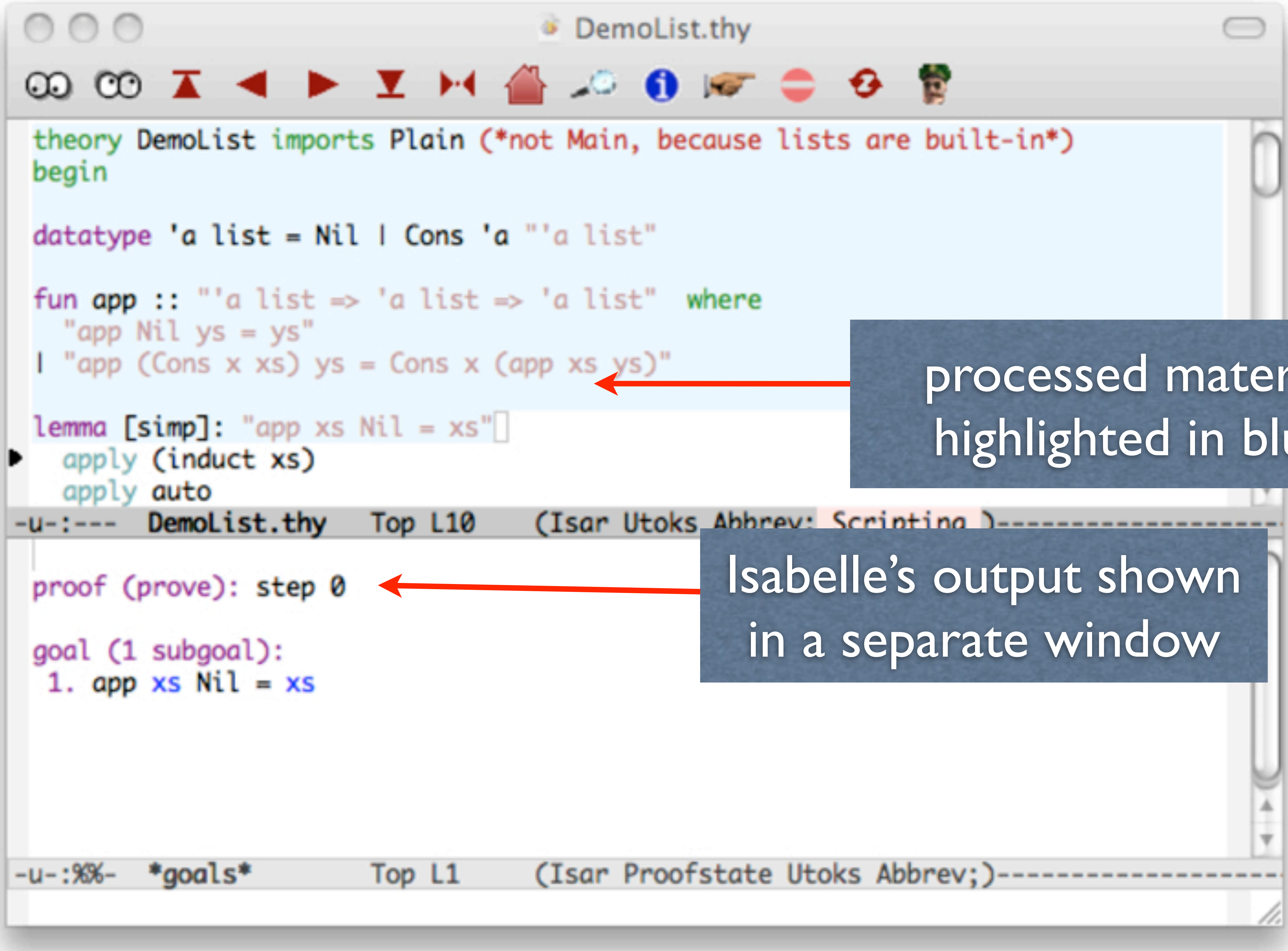
Proof General



```
theory DemoList imports Plain (*not Main, because lists are built-in*)
begin

datatype 'a list = Nil | Cons 'a "'a list"

fun app :: "'a list => 'a list => 'a list"  where
  "app Nil ys = ys"
| "app (Cons x xs) ys = Cons x (app xs ys)"

lemma [simp]: "app xs Nil = xs"
  apply (induct xs)
  apply auto
-u-:---  DemoList.thy    Top L10    (Isar Utoks Abbrev; Scripting )------------------

proof (prove): step 0

goal (1 subgoal):
 1. app xs Nil = xs



-u-:%%-  *goals*         Top L1     (Isar Proofstate Utoks Abbrev;)------------------
```

processed material highlighted in blue

Proof General



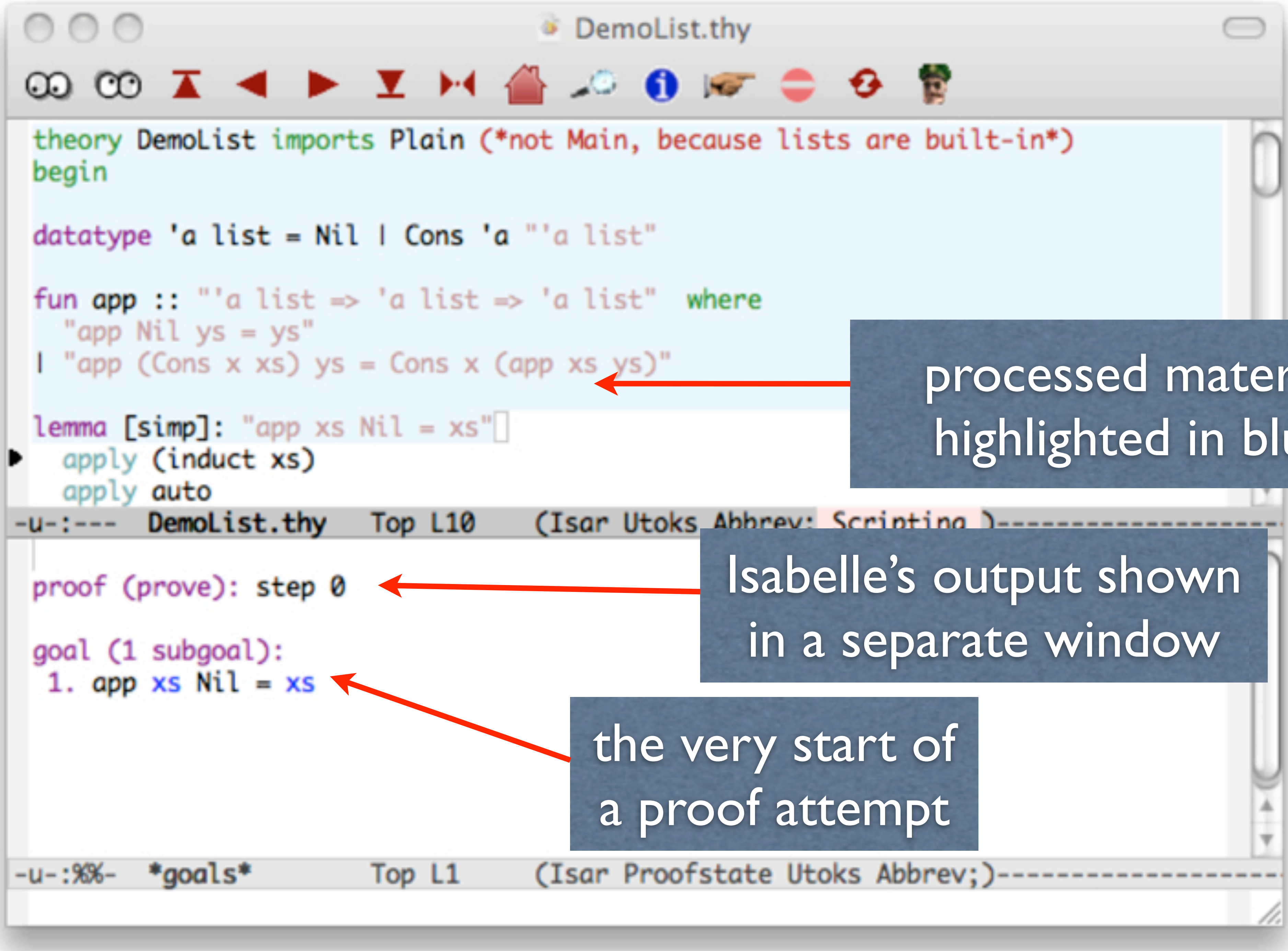
```
theory DemoList imports Plain (*not Main, because lists are built-in*)
begin

datatype 'a list = Nil | Cons 'a "'a list"

fun app :: "'a list => 'a list => 'a list"  where
  "app Nil ys = ys"
| "app (Cons x xs) ys = Cons x (app xs ys)"

lemma [simp]: "app xs Nil = xs"
  apply (induct xs)
  apply auto
-u-:---  DemoList.thy    Top L10    (Isar Utoks Abbrev; Scripting )------------------
```

```
proof (prove): step 0

goal (1 subgoal):
 1. app xs Nil = xs
-u-:%%-  *goals*    Top L1    (Isar Proofstate Utoks Abbrev;)------------------
```

processed material highlighted in blue

Isabelle's output shown in a separate window

Proof General



```
theory DemoList imports Plain (*not Main, because lists are built-in*)
begin

datatype 'a list = Nil | Cons 'a "'a list"

fun app :: "'a list => 'a list => 'a list"  where
  "app Nil ys = ys"
| "app (Cons x xs) ys = Cons x (app xs ys)"

lemma [simp]: "app xs Nil = xs"
  apply (induct xs)
  apply auto
-u-:---  DemoList.thy    Top L10    (Isar Utoks Abbrev; Scripting )-------------------
```

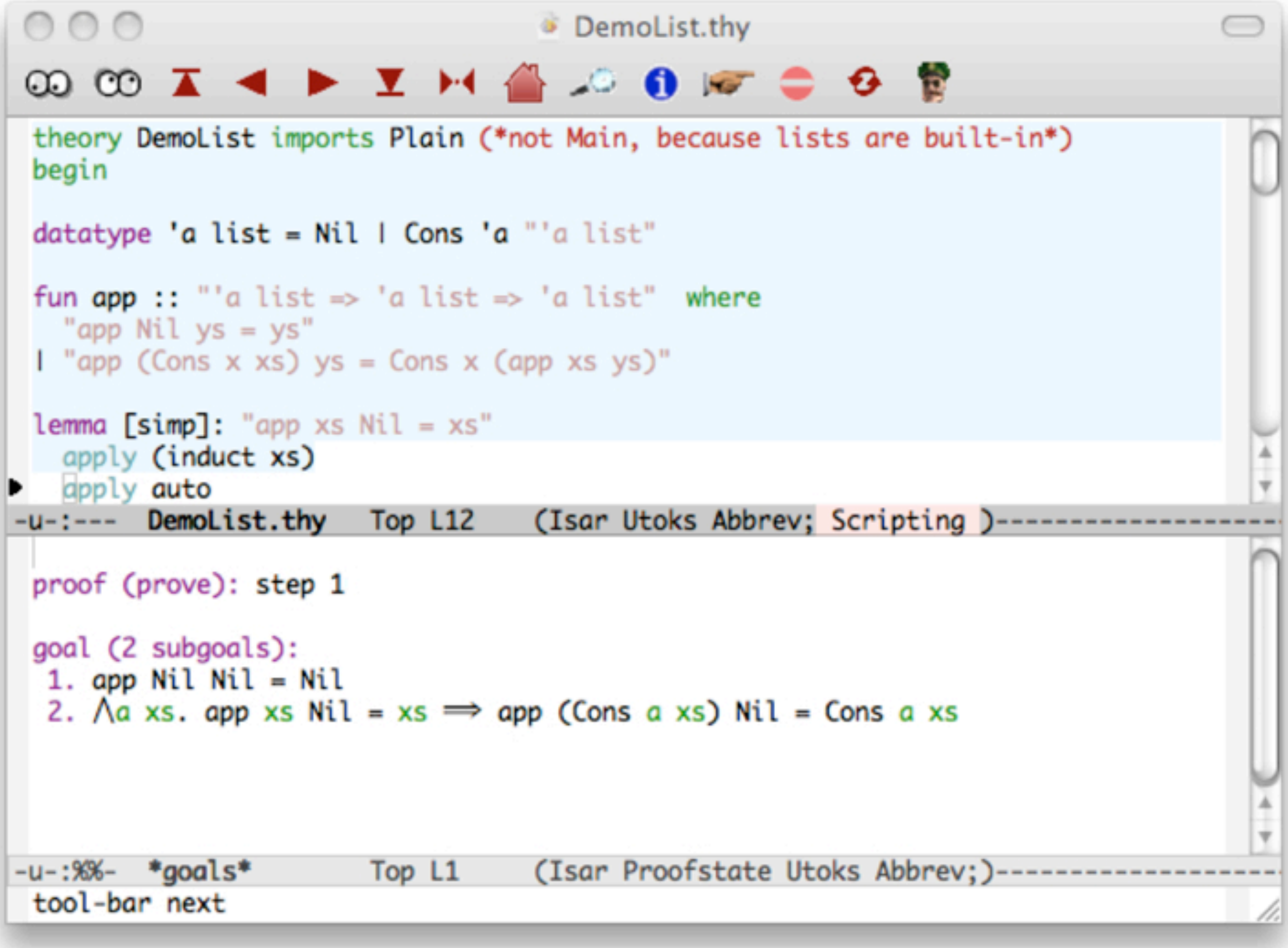
```
proof (prove): step 0

goal (1 subgoal):
 1. app xs Nil = xs
-u-:%%-  *goals*         Top L1     (Isar Proofstate Utoks Abbrev;)------------------
```

processed material highlighted in blue

Isabelle's output shown in a separate window

the very start of a proof attempt

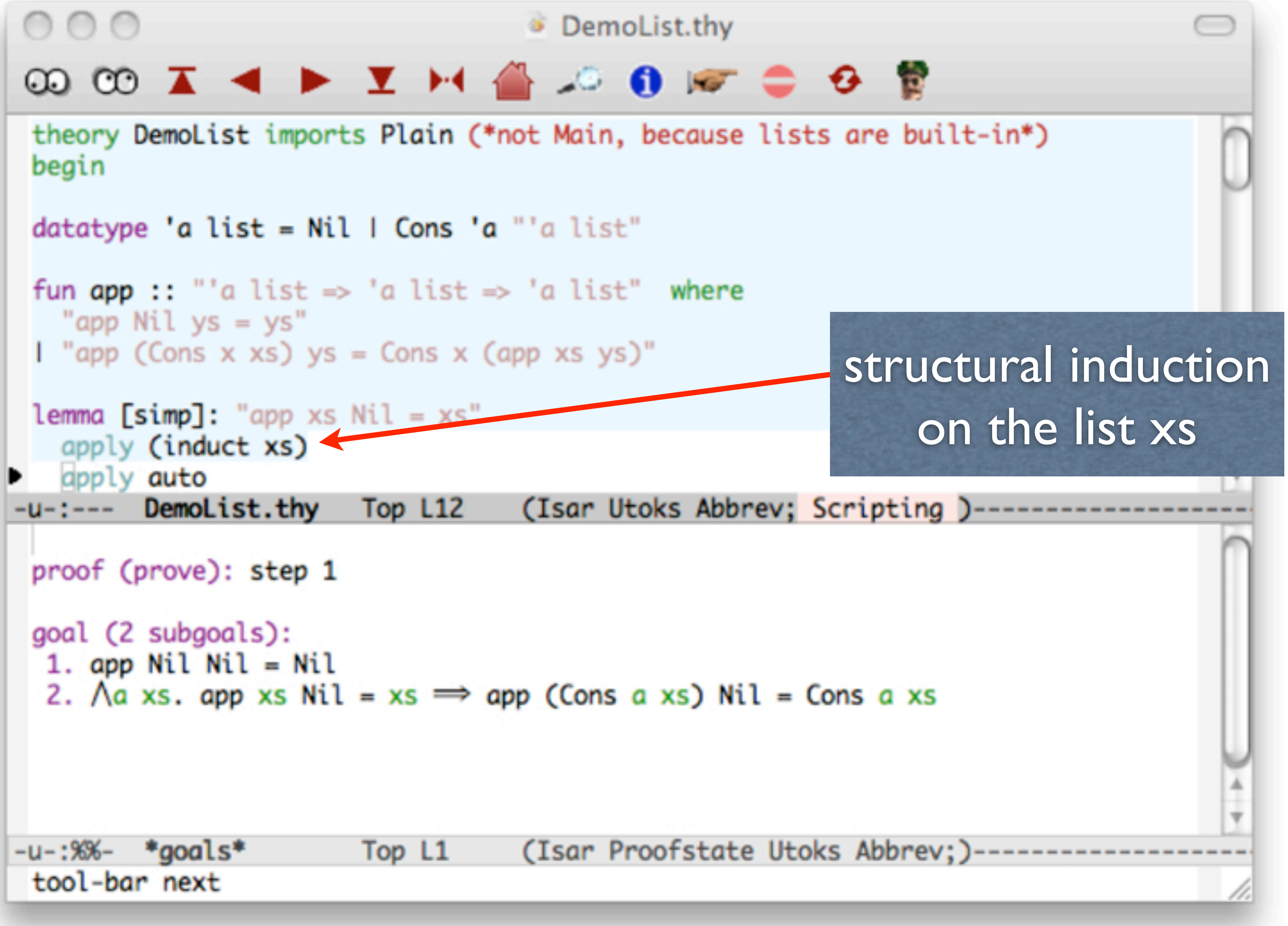
Proof by Induction



```
theory DemoList imports Plain (*not Main, because lists are built-in*)
begin

datatype 'a list = Nil | Cons 'a "'a list"

fun app :: "'a list => 'a list => 'a list"  where
  "app Nil ys = ys"
| "app (Cons x xs) ys = Cons x (app xs ys)"

lemma [simp]: "app xs Nil = xs"
  apply (induct xs)
  apply auto
-u-:---  DemoList.thy    Top L12    (Isar Utoks Abbrev; Scripting )------------------
```

```
proof (prove): step 1

goal (2 subgoals):
 1. app Nil Nil = Nil
 2. ⋀a xs. app xs Nil = xs ⟹ app (Cons a xs) Nil = Cons a xs
-u-:%%-  *goals*         Top L1     (Isar Proofstate Utoks Abbrev;)------------------
tool-bar next
```

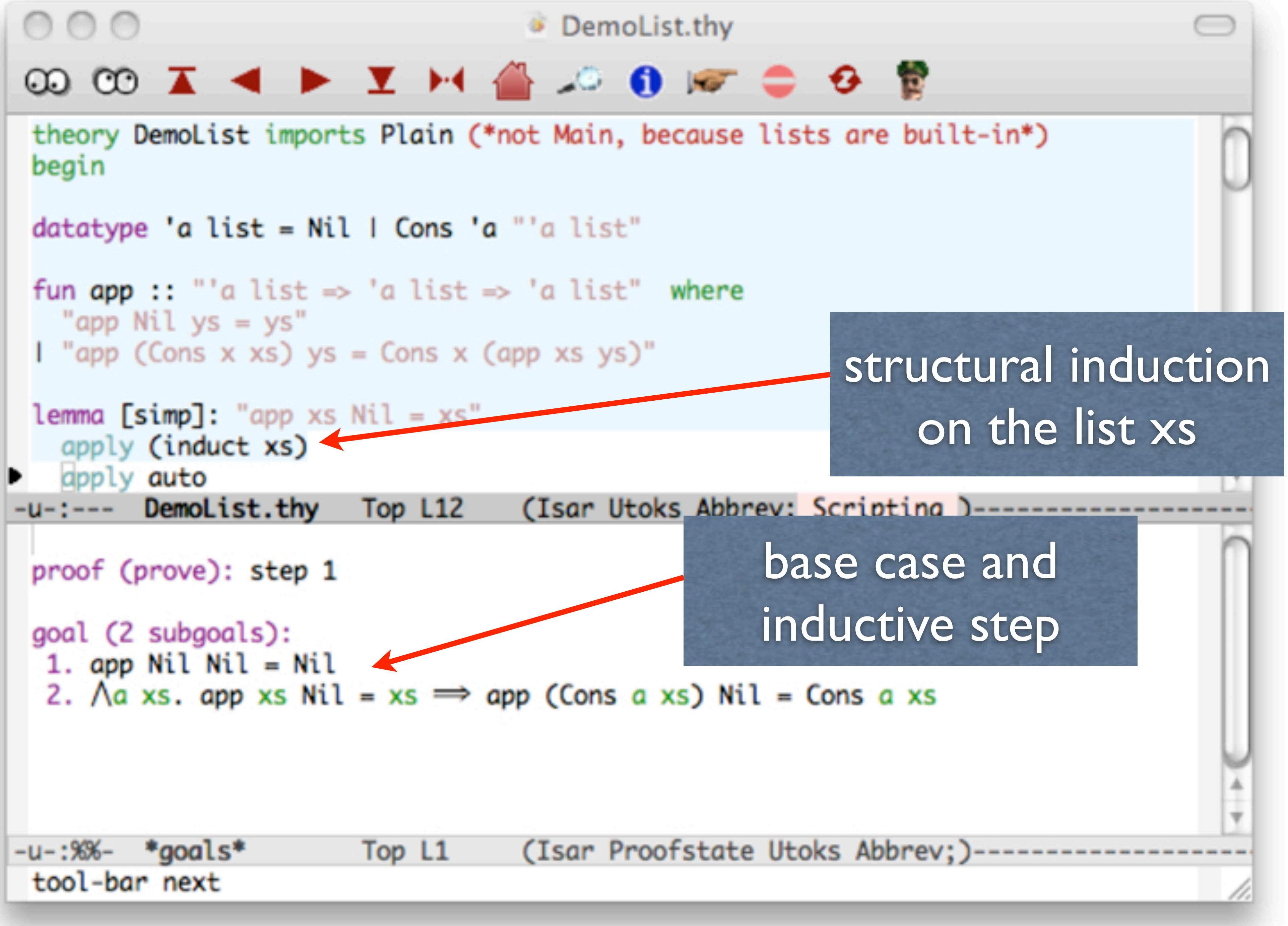
Proof by Induction



```
theory DemoList imports Plain (*not Main, because lists are built-in*)
begin

datatype 'a list = Nil | Cons 'a "'a list"

fun app :: "'a list => 'a list => 'a list"  where
  "app Nil ys = ys"
| "app (Cons x xs) ys = Cons x (app xs ys)"

lemma [simp]: "app xs Nil = xs"
  apply (induct xs)
  apply auto
-u-:---  DemoList.thy    Top L12    (Isar Utoks Abbrev; Scripting )------------------

proof (prove): step 1

goal (2 subgoals):
 1. app Nil Nil = Nil
 2. ⋀a xs. app xs Nil = xs ⟹ app (Cons a xs) Nil = Cons a xs

-u-:%%-  *goals*    Top L1    (Isar Proofstate Utoks Abbrev;)------------------
tool-bar next
```

structural induction
on the list xs

Proof by Induction

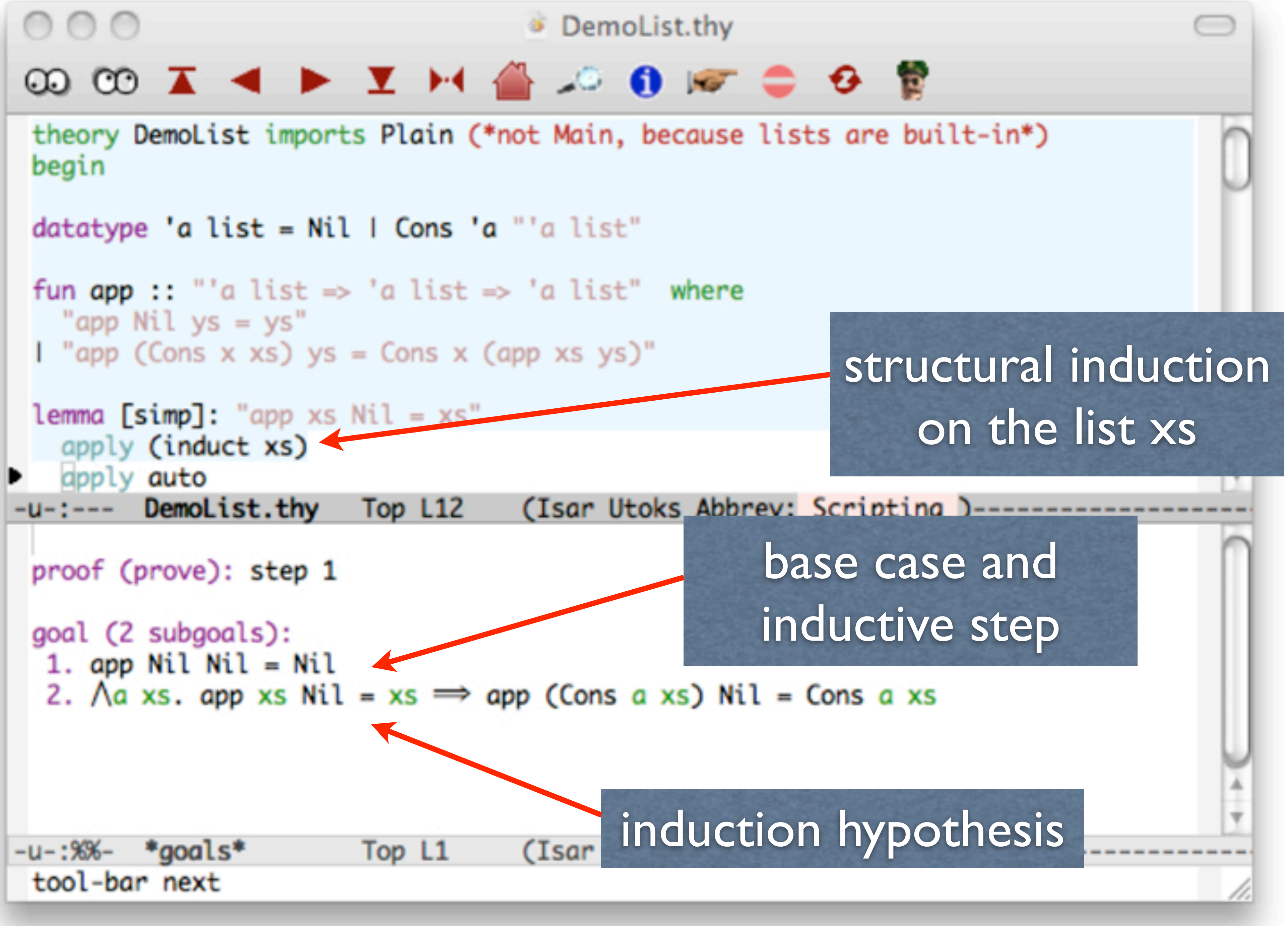


```
theory DemoList imports Plain (*not Main, because lists are built-in*)
begin

datatype 'a list = Nil | Cons 'a "'a list"

fun app :: "'a list => 'a list => 'a list"  where
  "app Nil ys = ys"
| "app (Cons x xs) ys = Cons x (app xs ys)"

lemma [simp]: "app xs Nil = xs"
  apply (induct xs)
  apply auto
-u-:---  DemoList.thy    Top L12     (Isar Utoks Abbrev; Scripting )------------------

proof (prove): step 1

goal (2 subgoals):
 1. app Nil Nil = Nil
 2. ⋀a xs. app xs Nil = xs ⟹ app (Cons a xs) Nil = Cons a xs

-u-:%%-  *goals*         Top L1      (Isar Proofstate Utoks Abbrev;)------------------
tool-bar next
```

structural induction on the list xs

base case and inductive step

Proof by Induction



```
theory DemoList imports Plain (*not Main, because lists are built-in*)
begin

datatype 'a list = Nil | Cons 'a "'a list"

fun app :: "'a list => 'a list => 'a list"  where
  "app Nil ys = ys"
| "app (Cons x xs) ys = Cons x (app xs ys)"

lemma [simp]: "app xs Nil = xs"
  apply (induct xs)
  apply auto
-u-:---  DemoList.thy    Top L12    (Isar Utoks Abbrev; Scripting )------------------
```

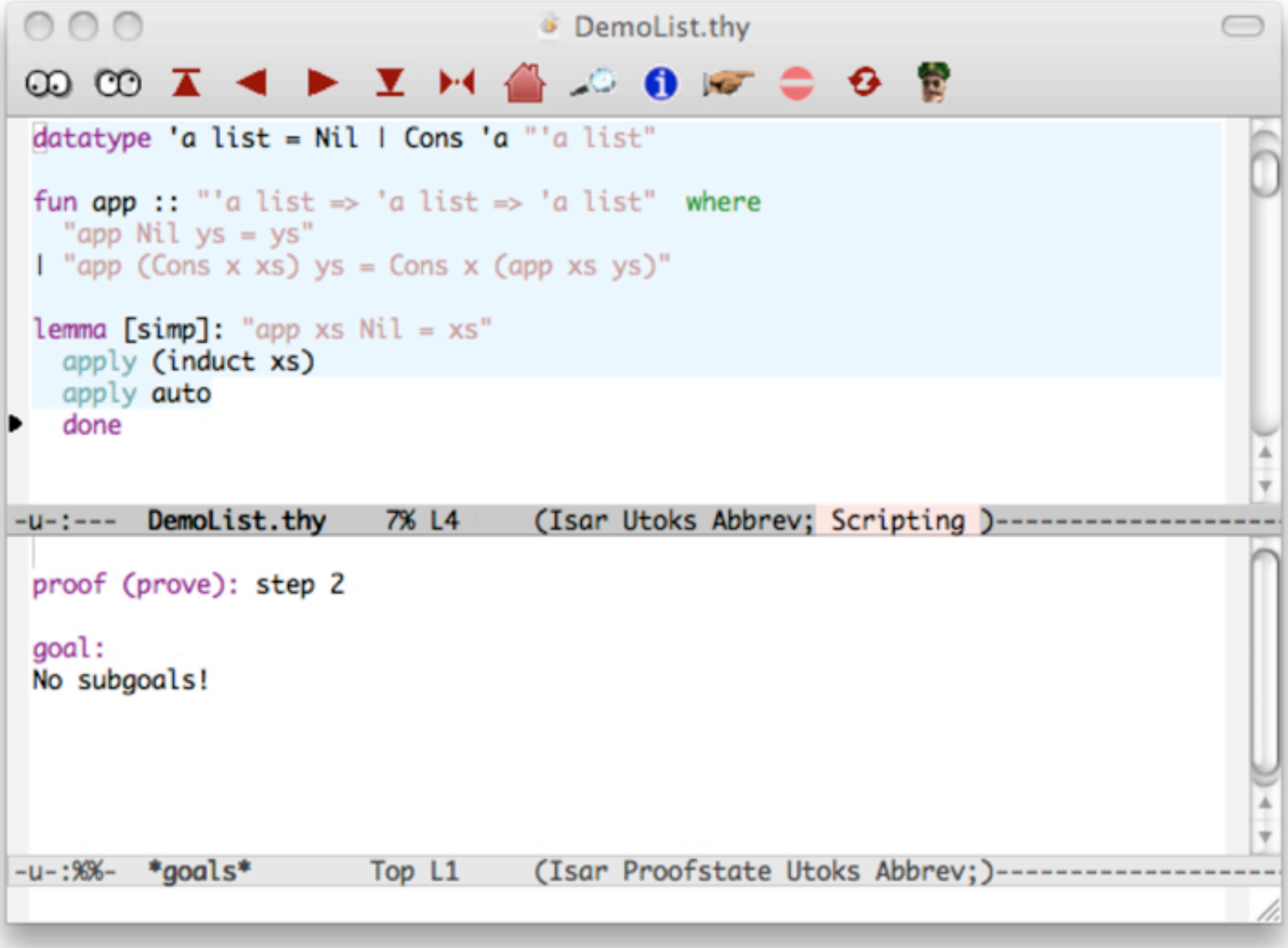
```
proof (prove): step 1

goal (2 subgoals):
 1. app Nil Nil = Nil
 2. ⋀a xs. app xs Nil = xs ⟹ app (Cons a xs) Nil = Cons a xs

-u-:%%-  *goals*    Top L1    (Isar
tool-bar next
```

structural induction on the list xs

base case and inductive step

induction hypothesis

Finishing a Proof



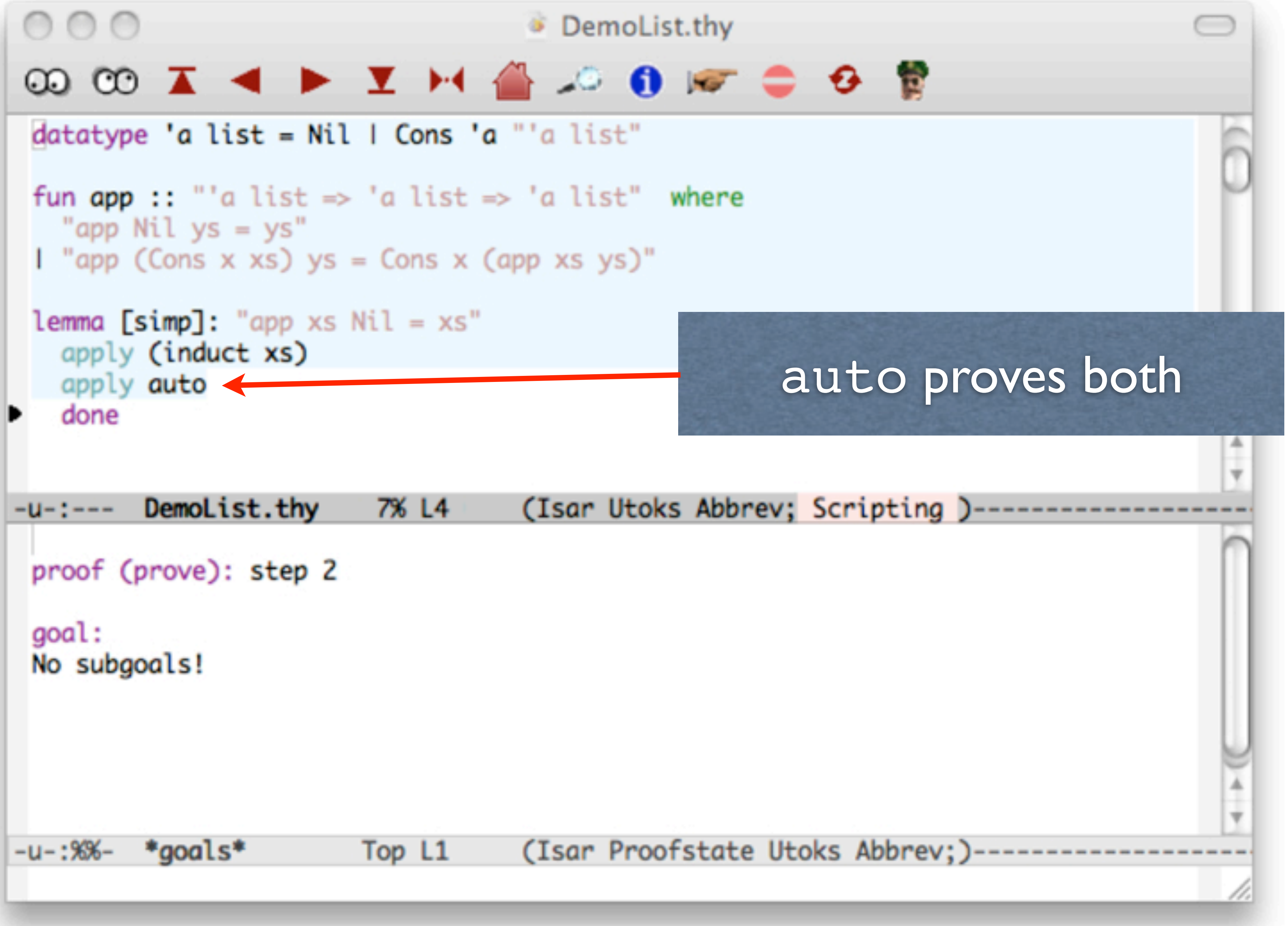
```
datatype 'a list = Nil | Cons 'a "'a list"

fun app :: "'a list => 'a list => 'a list"  where
  "app Nil ys = ys"
| "app (Cons x xs) ys = Cons x (app xs ys)"

lemma [simp]: "app xs Nil = xs"
  apply (induct xs)
  apply auto
  done
```

```
-u-:---  DemoList.thy    7% L4     (Isar Utoks Abbrev; Scripting )------------------
```

```
proof (prove): step 2

goal:
No subgoals!
```

```
-u-:%%-  *goals*       Top L1     (Isar Proofstate Utoks Abbrev;)------------------
```

Finishing a Proof



```
datatype 'a list = Nil | Cons 'a "'a list"

fun app :: "'a list => 'a list => 'a list"  where
  "app Nil ys = ys"
| "app (Cons x xs) ys = Cons x (app xs ys)"

lemma [simp]: "app xs Nil = xs"
  apply (induct xs)
  apply auto
  done
```

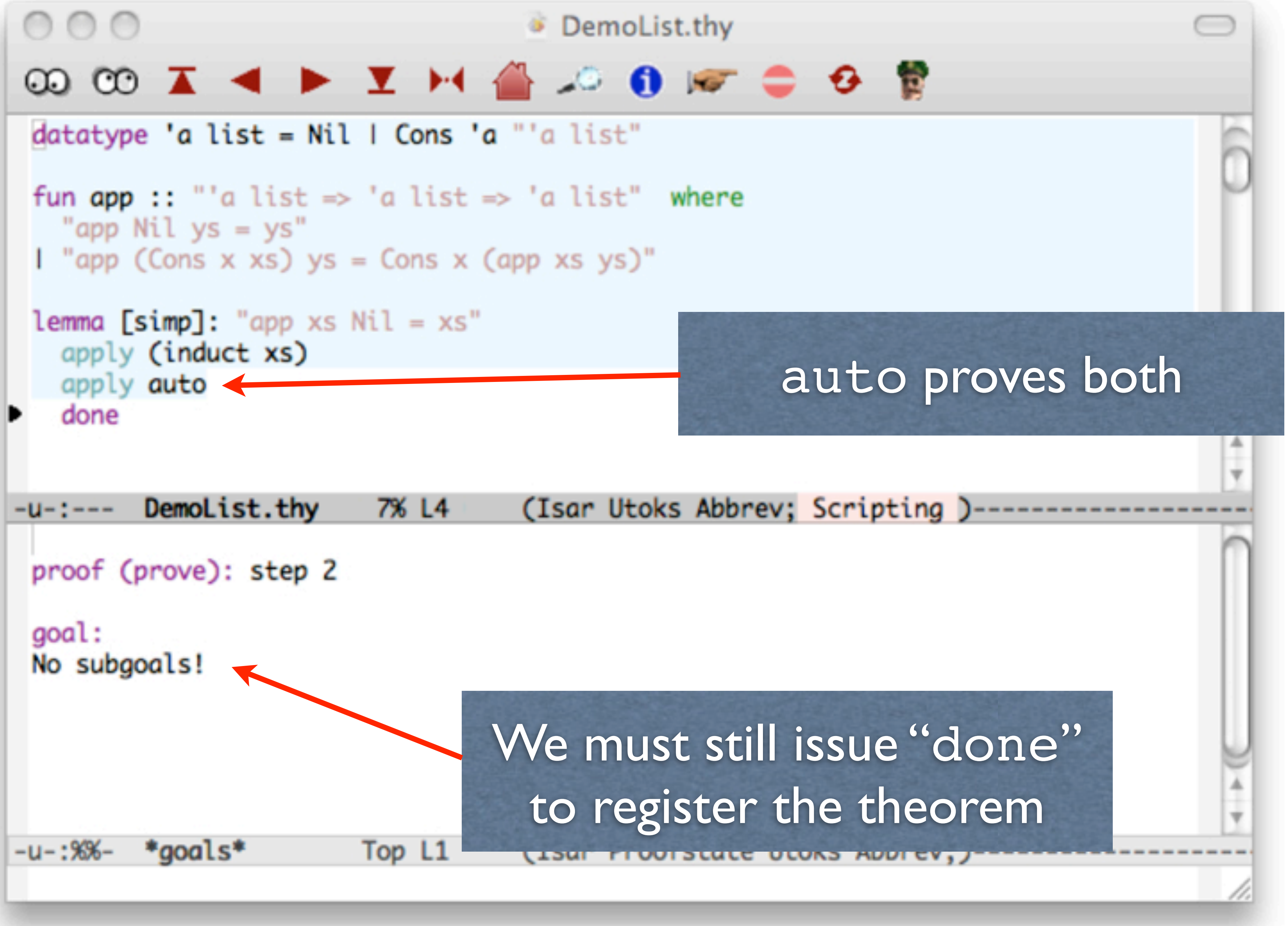
```
-u-:---  DemoList.thy     7% L4     (Isar Utoks Abbrev; Scripting )------------------
```

```
proof (prove): step 2

goal:
No subgoals!
```

```
-u-:%%-  *goals*        Top L1     (Isar Proofstate Utoks Abbrev;)------------------
```

auto proves both

Finishing a Proof



```
datatype 'a list = Nil | Cons 'a "'a list"

fun app :: "'a list => 'a list => 'a list"  where
  "app Nil ys = ys"
| "app (Cons x xs) ys = Cons x (app xs ys)"

lemma [simp]: "app xs Nil = xs"
  apply (induct xs)
  apply auto
  done
```

```
-u-:---  DemoList.thy      7% L4      (Isar Utoks Abbrev; Scripting )------------------
```

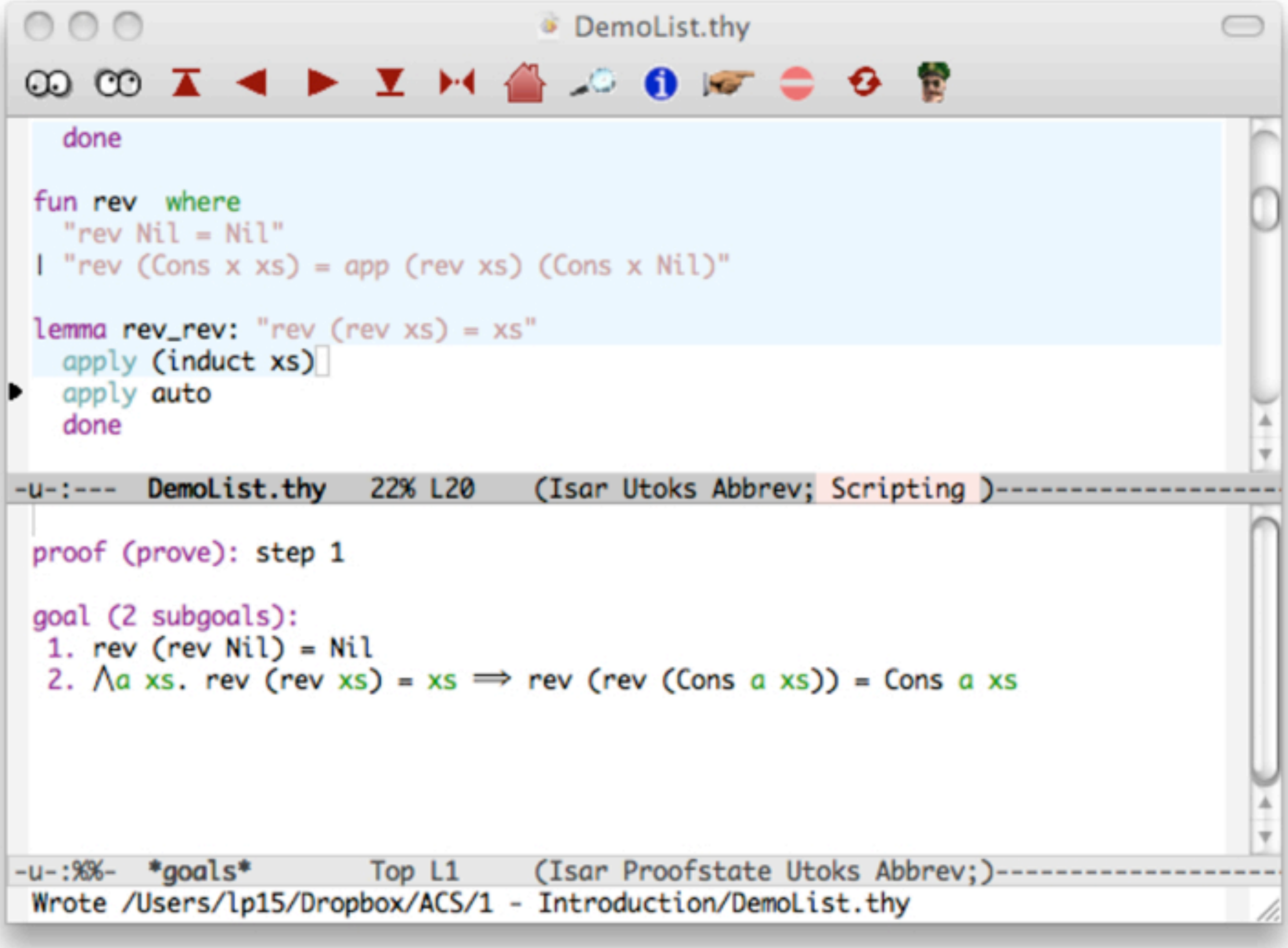
```
proof (prove): step 2

goal:
No subgoals!
```

```
-u-:%%-  *goals*         Top L1
```

auto proves both

We must still issue "done" to register the theorem

Another Proof Attempt



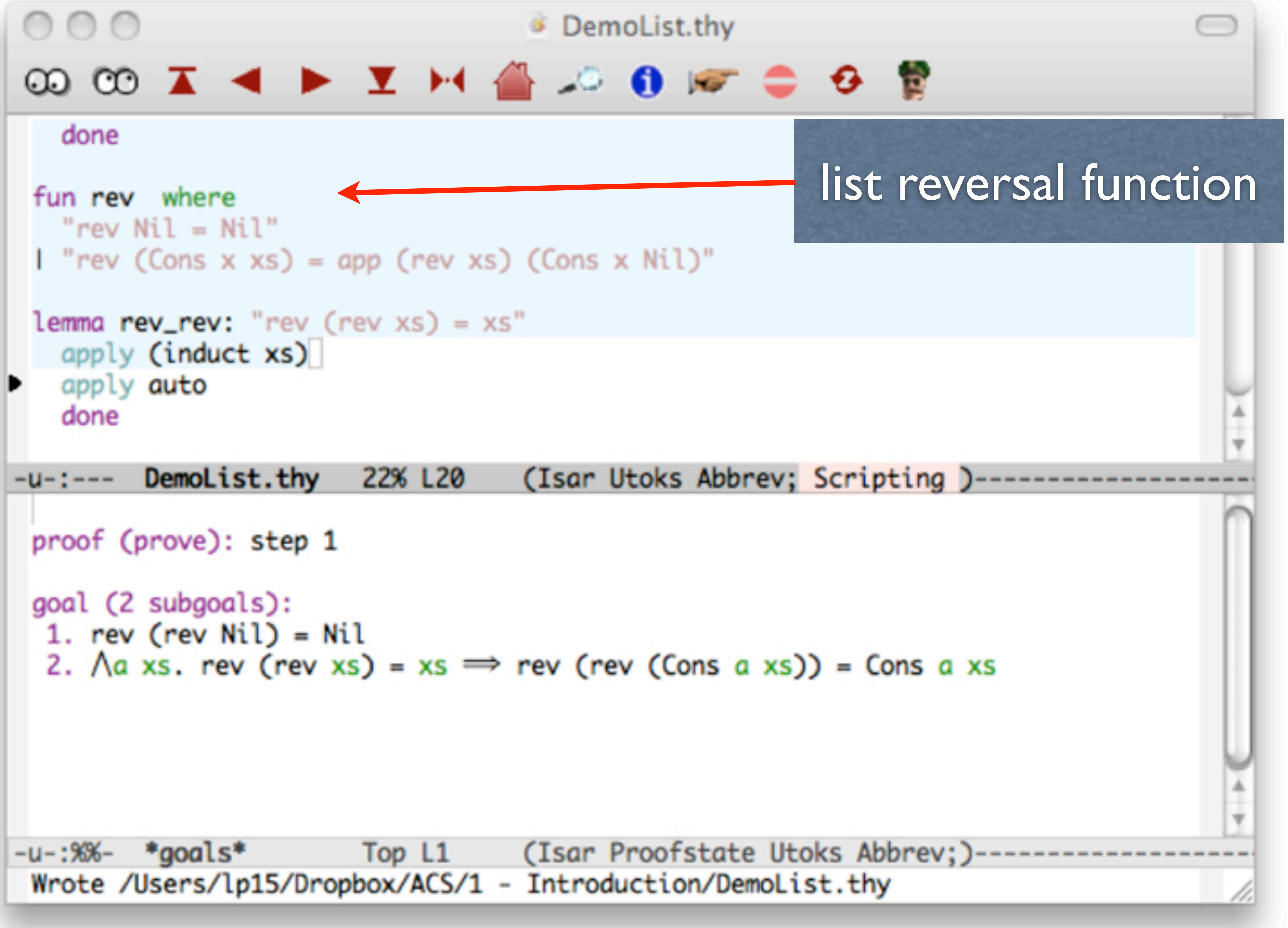
```
  done

fun rev  where
  "rev Nil = Nil"
| "rev (Cons x xs) = app (rev xs) (Cons x Nil)"

lemma rev_rev: "rev (rev xs) = xs"
  apply (induct xs)
  apply auto
  done
```

```
-u-:---  DemoList.thy    22% L20    (Isar Utoks Abbrev; Scripting )------------------
```

```
proof (prove): step 1

goal (2 subgoals):
 1. rev (rev Nil) = Nil
 2. ⋀a xs. rev (rev xs) = xs ⟹ rev (rev (Cons a xs)) = Cons a xs
```

```
-u-:%%-  *goals*         Top L1     (Isar Proofstate Utoks Abbrev;)------------------
Wrote /Users/lp15/Dropbox/ACS/1 - Introduction/DemoList.thy
```

Another Proof Attempt



```
  done

fun rev  where
  "rev Nil = Nil"
| "rev (Cons x xs) = app (rev xs) (Cons x Nil)"

lemma rev_rev: "rev (rev xs) = xs"
  apply (induct xs)
  apply auto
  done
```

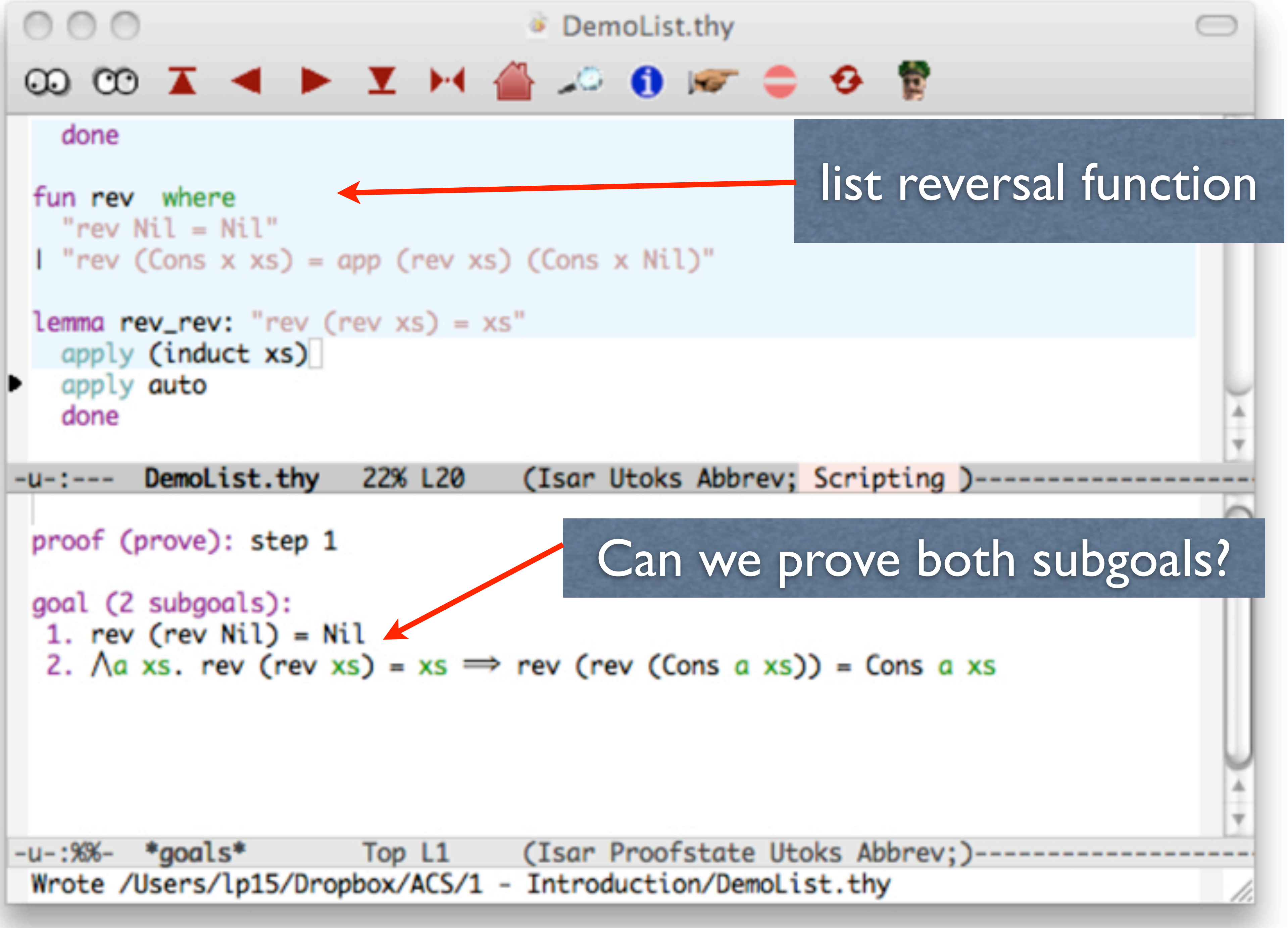
```
-u-:---  DemoList.thy    22% L20    (Isar Utoks Abbrev; Scripting )------------------
```

```
proof (prove): step 1

goal (2 subgoals):
 1. rev (rev Nil) = Nil
 2. ⋀a xs. rev (rev xs) = xs ⟹ rev (rev (Cons a xs)) = Cons a xs
```

```
-u-:%%-  *goals*       Top L1     (Isar Proofstate Utoks Abbrev;)------------------
Wrote /Users/lp15/Dropbox/ACS/1 - Introduction/DemoList.thy
```

list reversal function

Another Proof Attempt



```
  done

fun rev where
  "rev Nil = Nil"
| "rev (Cons x xs) = app (rev xs) (Cons x Nil)"

lemma rev_rev: "rev (rev xs) = xs"
  apply (induct xs)
  apply auto
  done
```

```
-u-:---  DemoList.thy    22% L20    (Isar Utoks Abbrev; Scripting )-----------------
```

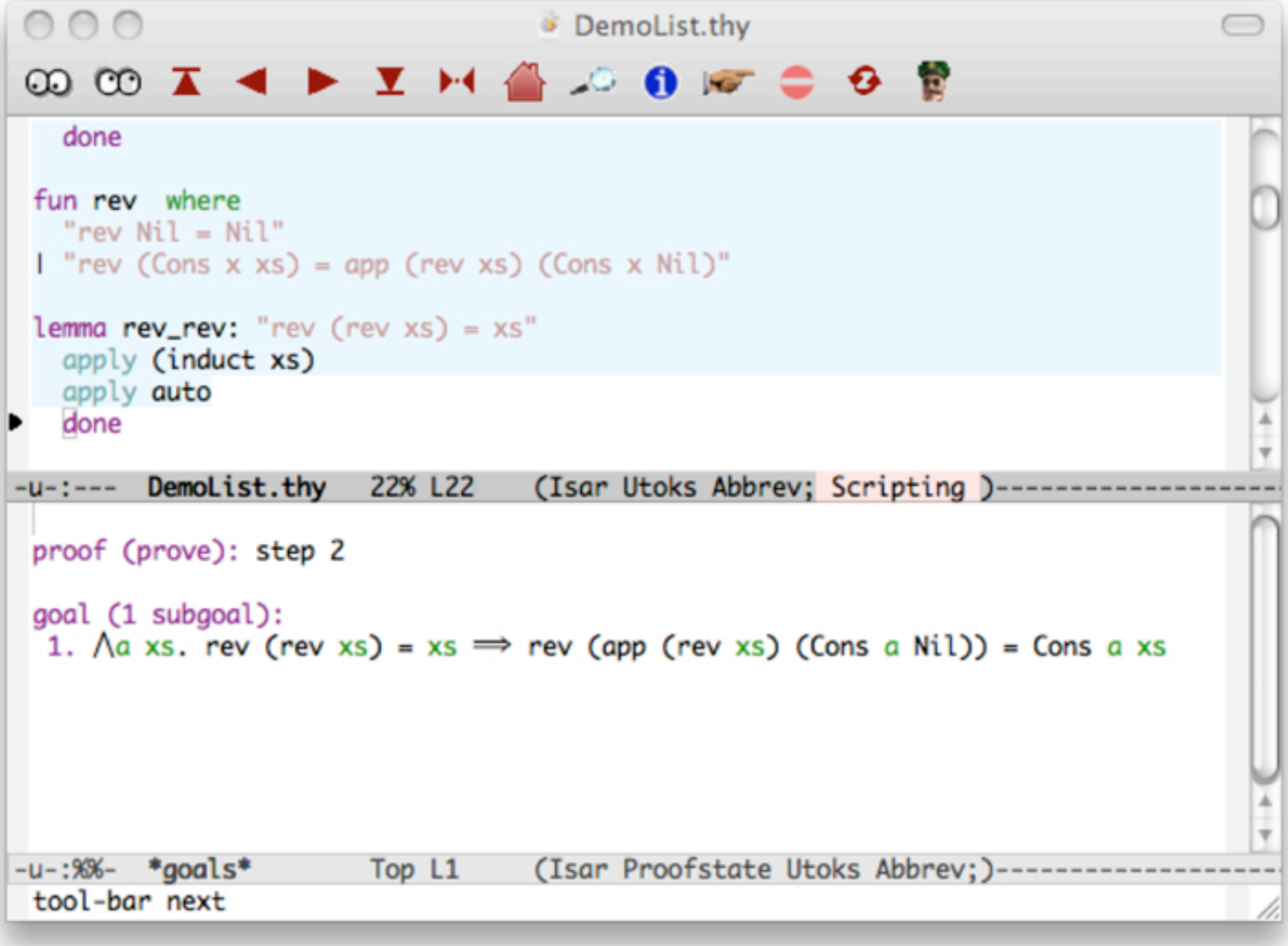
```
proof (prove): step 1

goal (2 subgoals):
 1. rev (rev Nil) = Nil
 2. ⋀a xs. rev (rev xs) = xs ⟹ rev (rev (Cons a xs)) = Cons a xs
```

```
-u-:%%-  *goals*         Top L1     (Isar Proofstate Utoks Abbrev;)-----------------
Wrote /Users/lp15/Dropbox/ACS/1 - Introduction/DemoList.thy
```

list reversal function

Can we prove both subgoals?

Stuck!



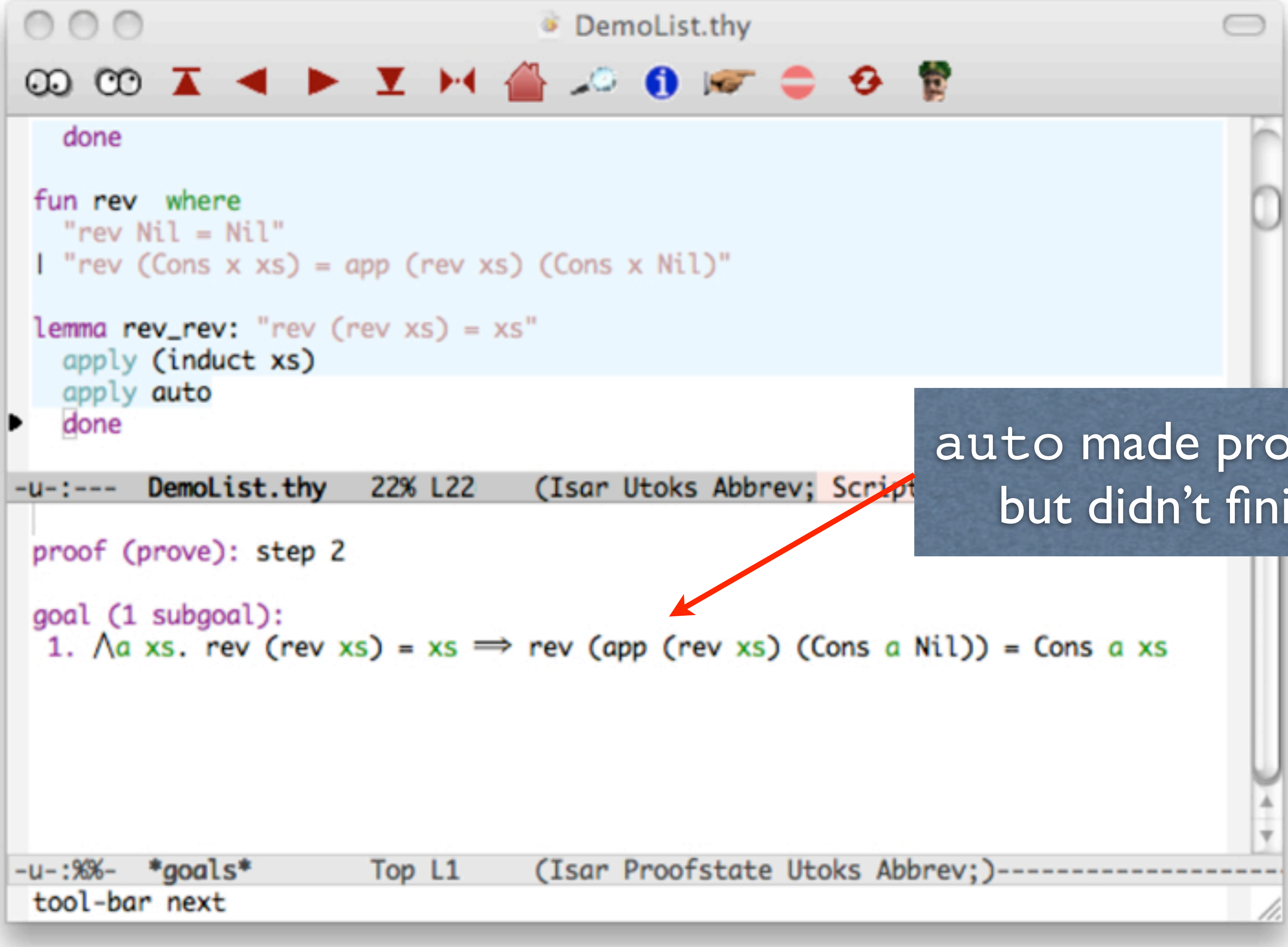
```
  done

fun rev  where
  "rev Nil = Nil"
| "rev (Cons x xs) = app (rev xs) (Cons x Nil)"

lemma rev_rev: "rev (rev xs) = xs"
  apply (induct xs)
  apply auto
  done
```

```
-u-:---  DemoList.thy   22% L22    (Isar Utoks Abbrev; Scripting )------------------
```

```
proof (prove): step 2

goal (1 subgoal):
 1. ⋀a xs. rev (rev xs) = xs ⟹ rev (app (rev xs) (Cons a Nil)) = Cons a xs
```

```
-u-:%%-  *goals*        Top L1     (Isar Proofstate Utoks Abbrev;)------------------
tool-bar next
```

Stuck!



```
  done

fun rev  where
  "rev Nil = Nil"
| "rev (Cons x xs) = app (rev xs) (Cons x Nil)"

lemma rev_rev: "rev (rev xs) = xs"
  apply (induct xs)
  apply auto
  done
```

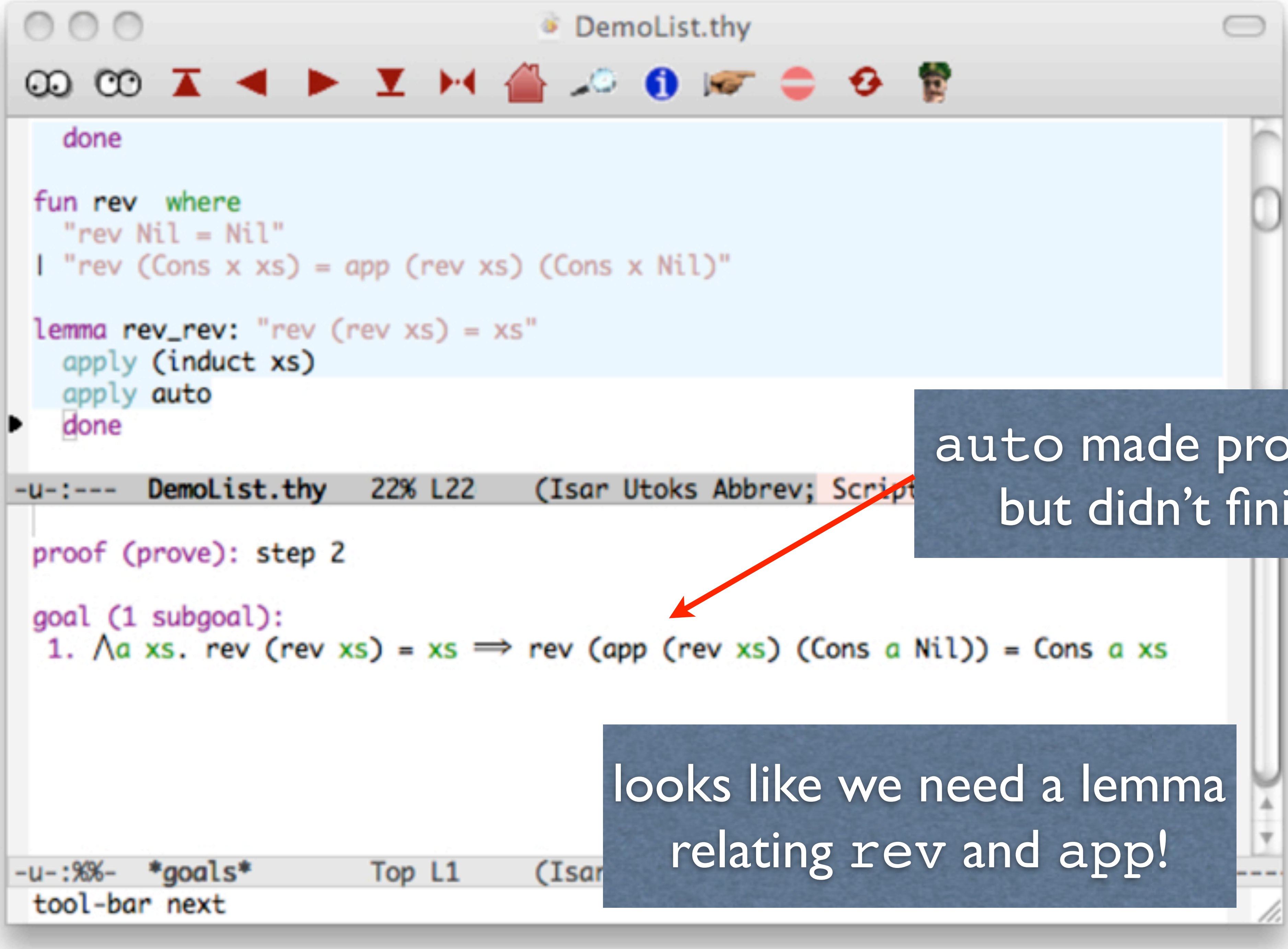
```
-u-:---  DemoList.thy   22% L22   (Isar Utoks Abbrev; Script
```

```
proof (prove): step 2

goal (1 subgoal):
 1. ⋀a xs. rev (rev xs) = xs ⟹ rev (app (rev xs) (Cons a Nil)) = Cons a xs
```

```
-u-:%%-  *goals*     Top L1    (Isar Proofstate Utoks Abbrev;)-------------------
tool-bar next
```

auto made progress
but didn't finish

Stuck!



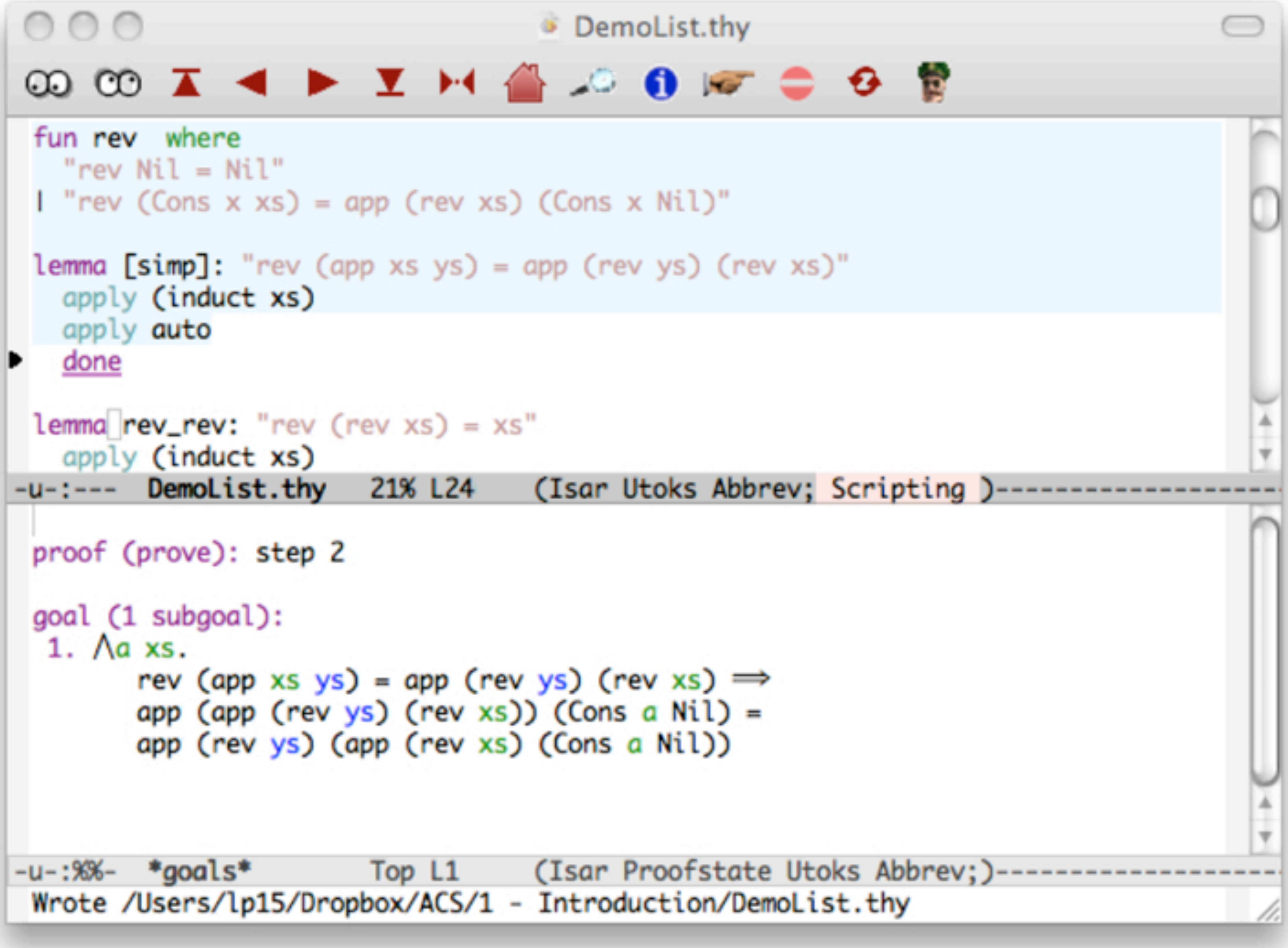
```
  done

fun rev  where
  "rev Nil = Nil"
| "rev (Cons x xs) = app (rev xs) (Cons x Nil)"

lemma rev_rev: "rev (rev xs) = xs"
  apply (induct xs)
  apply auto
  done
```

```
proof (prove): step 2

goal (1 subgoal):
 1. ⋀a xs. rev (rev xs) = xs ⟹ rev (app (rev xs) (Cons a Nil)) = Cons a xs
```

auto made progress but didn't finish

looks like we need a lemma relating `rev` and app!

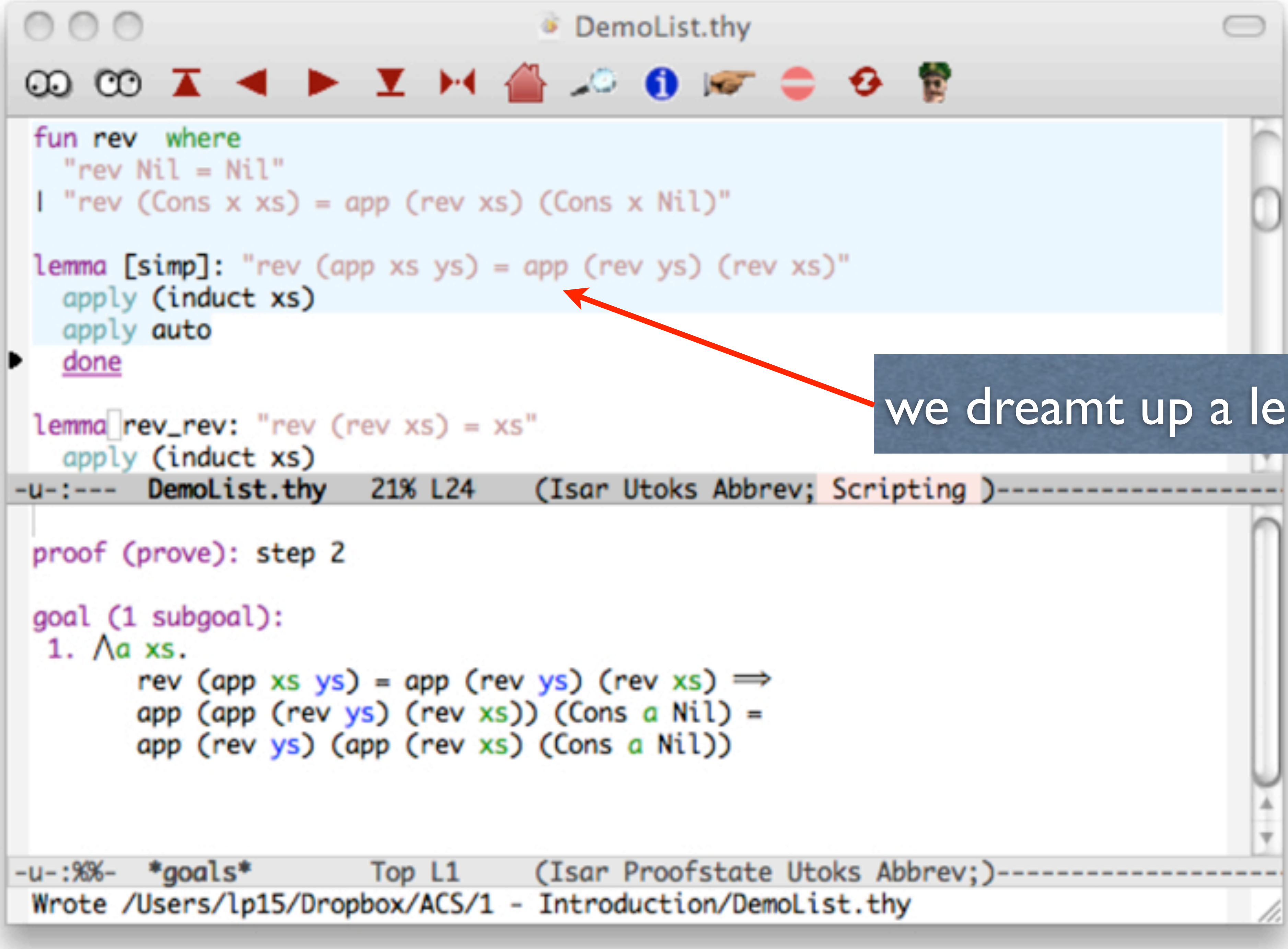
Stuck Again!



```
fun rev  where
  "rev Nil = Nil"
| "rev (Cons x xs) = app (rev xs) (Cons x Nil)"

lemma [simp]: "rev (app xs ys) = app (rev ys) (rev xs)"
  apply (induct xs)
  apply auto
  done

lemma rev_rev: "rev (rev xs) = xs"
  apply (induct xs)
-u-:---  DemoList.thy    21% L24     (Isar Utoks Abbrev; Scripting )------------------
```

```
proof (prove): step 2

goal (1 subgoal):
 1. ⋀a xs.
       rev (app xs ys) = app (rev ys) (rev xs) ⟹
       app (app (rev ys) (rev xs)) (Cons a Nil) =
       app (rev ys) (app (rev xs) (Cons a Nil))

-u-:%%-  *goals*        Top L1     (Isar Proofstate Utoks Abbrev;)------------------
Wrote /Users/lp15/Dropbox/ACS/1 - Introduction/DemoList.thy
```

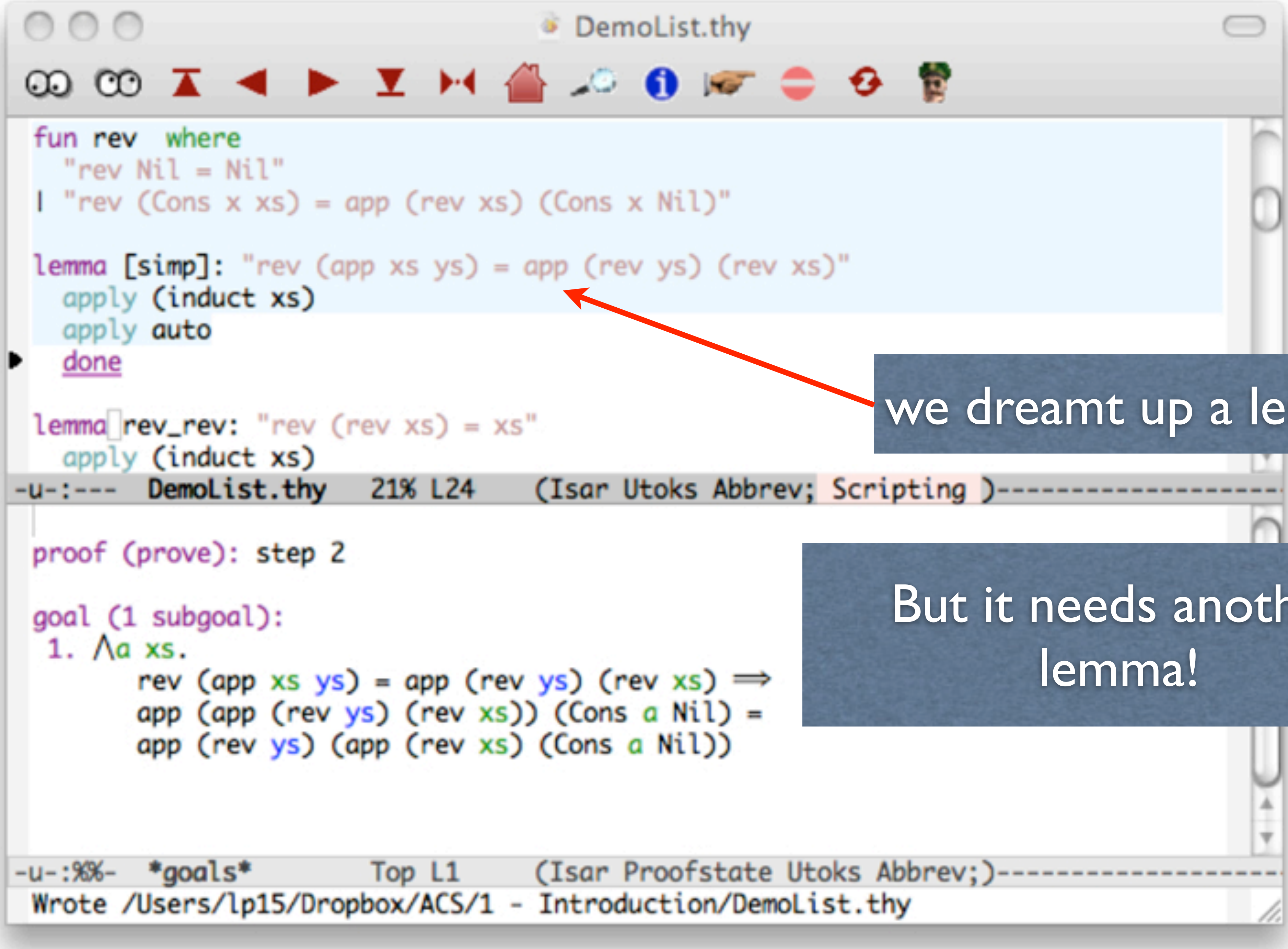
Stuck Again!



```
fun rev  where
  "rev Nil = Nil"
| "rev (Cons x xs) = app (rev xs) (Cons x Nil)"

lemma [simp]: "rev (app xs ys) = app (rev ys) (rev xs)"
  apply (induct xs)
  apply auto
  done

lemma rev_rev: "rev (rev xs) = xs"
  apply (induct xs)
-u-:---  DemoList.thy   21% L24    (Isar Utoks Abbrev; Scripting )------------------

proof (prove): step 2

goal (1 subgoal):
 1. ⋀a xs.
       rev (app xs ys) = app (rev ys) (rev xs) ⟹
       app (app (rev ys) (rev xs)) (Cons a Nil) =
       app (rev ys) (app (rev xs) (Cons a Nil))

-u-:%%-  *goals*        Top L1     (Isar Proofstate Utoks Abbrev;)------------------
Wrote /Users/lp15/Dropbox/ACS/1 - Introduction/DemoList.thy
```

we dreamt up a lemma...

Stuck Again!



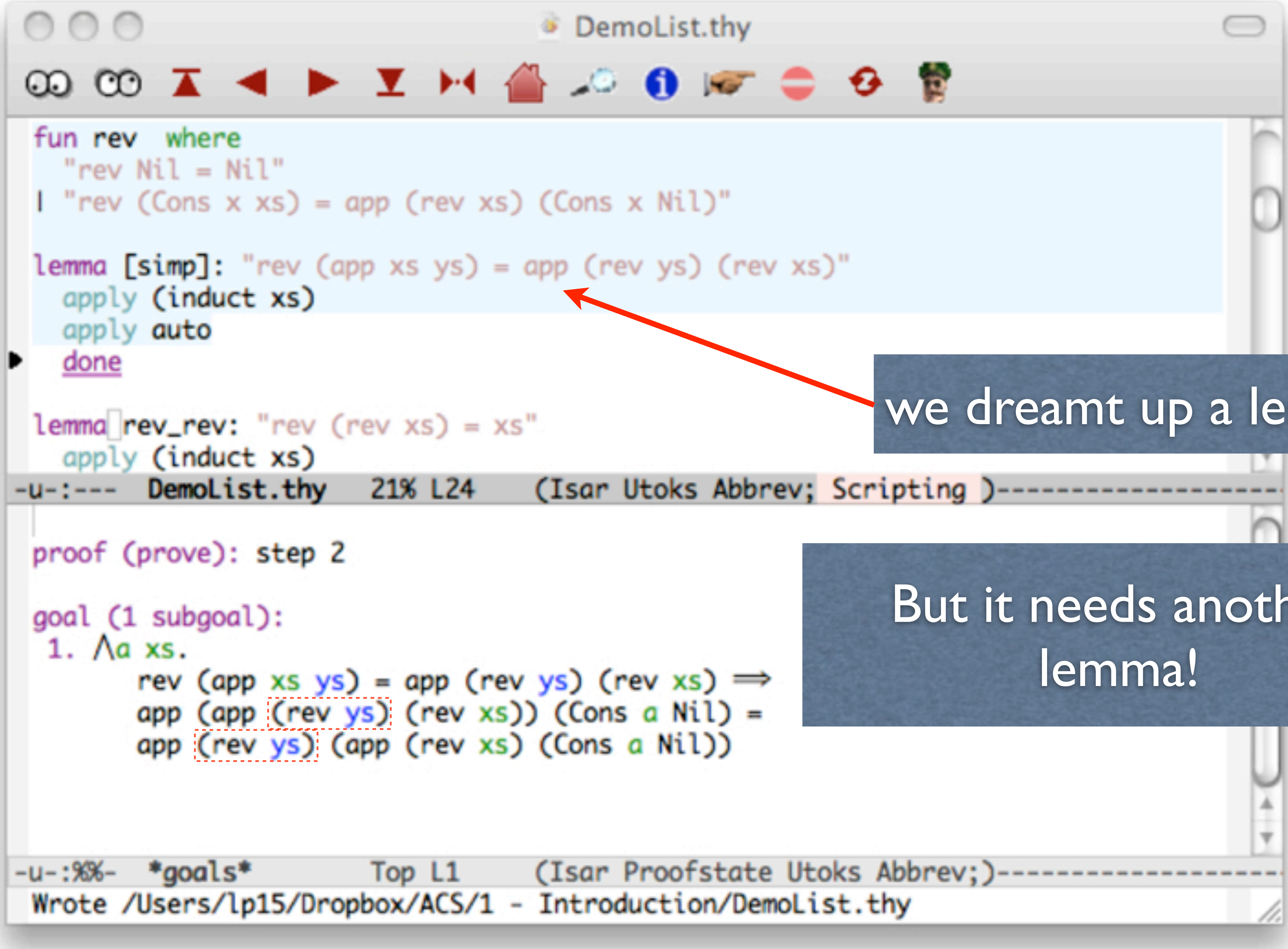
```
fun rev  where
  "rev Nil = Nil"
| "rev (Cons x xs) = app (rev xs) (Cons x Nil)"

lemma [simp]: "rev (app xs ys) = app (rev ys) (rev xs)"
  apply (induct xs)
  apply auto
  done

lemma rev_rev: "rev (rev xs) = xs"
  apply (induct xs)
-u-:---  DemoList.thy   21% L24    (Isar Utoks Abbrev; Scripting )------------------
```

```
proof (prove): step 2

goal (1 subgoal):
 1. ⋀a xs.
       rev (app xs ys) = app (rev ys) (rev xs) ⟹
       app (app (rev ys) (rev xs)) (Cons a Nil) =
       app (rev ys) (app (rev xs) (Cons a Nil))
-u-:%%-  *goals*        Top L1     (Isar Proofstate Utoks Abbrev;)------------------
Wrote /Users/lp15/Dropbox/ACS/1 - Introduction/DemoList.thy
```

we dreamt up a lemma...

But it needs another lemma!

Stuck Again!

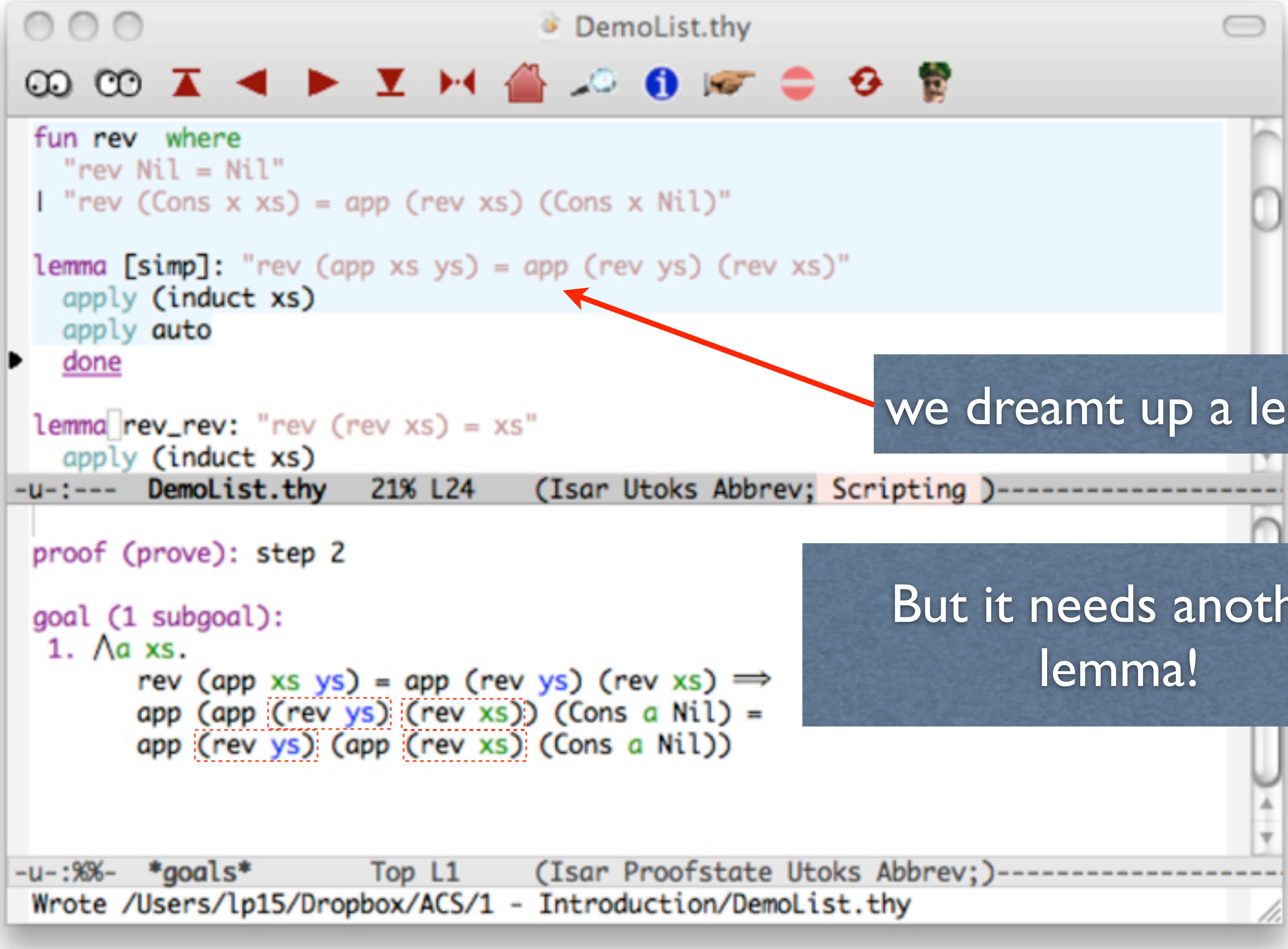


```
fun rev  where
  "rev Nil = Nil"
| "rev (Cons x xs) = app (rev xs) (Cons x Nil)"

lemma [simp]: "rev (app xs ys) = app (rev ys) (rev xs)"
  apply (induct xs)
  apply auto
  done

lemma rev_rev: "rev (rev xs) = xs"
  apply (induct xs)
-u-:---  DemoList.thy   21% L24    (Isar Utoks Abbrev; Scripting )------------------

proof (prove): step 2

goal (1 subgoal):
 1. ⋀a xs.
       rev (app xs ys) = app (rev ys) (rev xs) ⟹
       app (app (rev ys) (rev xs)) (Cons a Nil) =
       app (rev ys) (app (rev xs) (Cons a Nil))

-u-:%%-  *goals*        Top L1     (Isar Proofstate Utoks Abbrev;)------------------
Wrote /Users/lp15/Dropbox/ACS/1 - Introduction/DemoList.thy
```

we dreamt up a lemma...

But it needs another lemma!

Stuck Again!

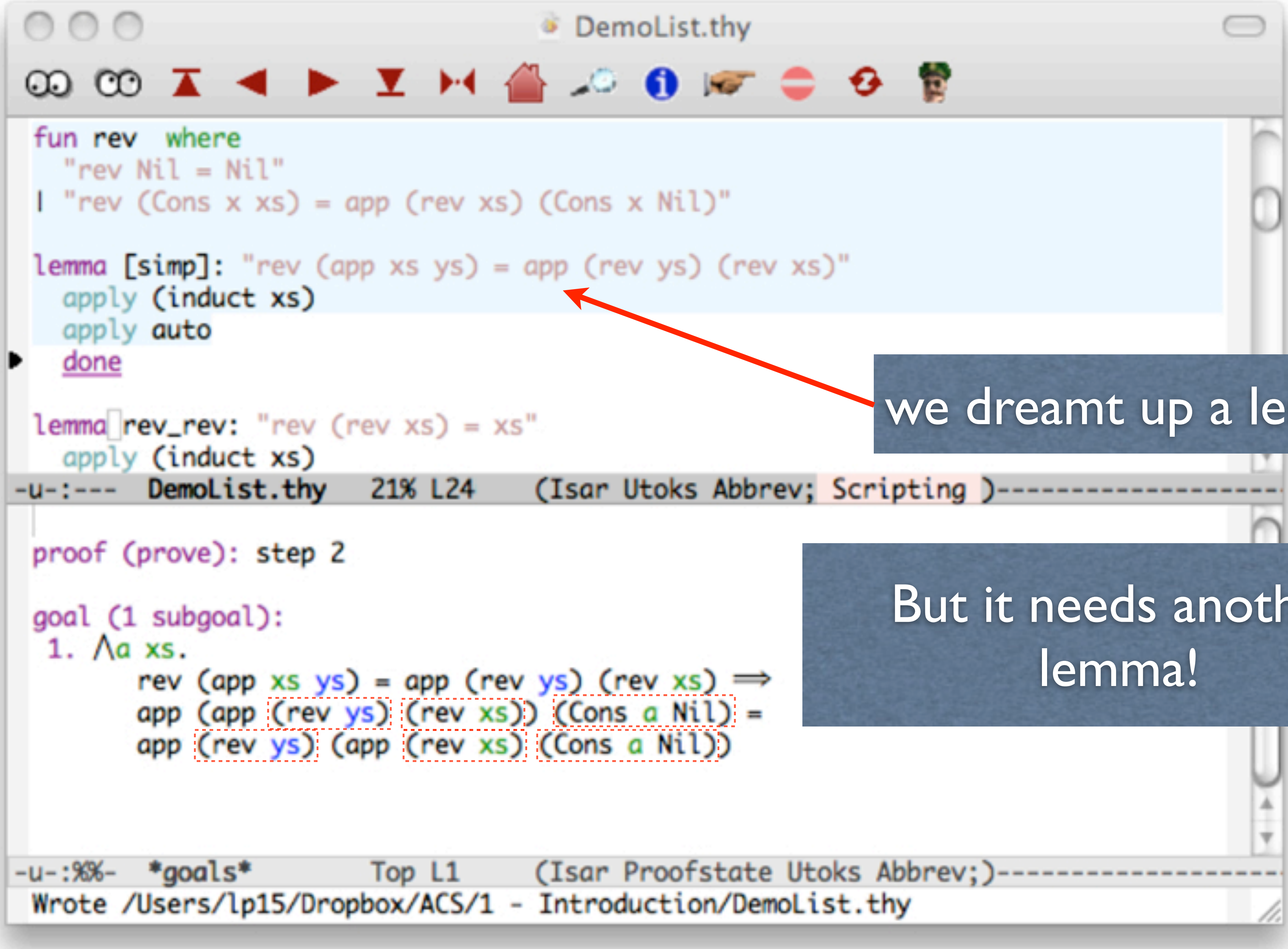


```
fun rev  where
  "rev Nil = Nil"
| "rev (Cons x xs) = app (rev xs) (Cons x Nil)"

lemma [simp]: "rev (app xs ys) = app (rev ys) (rev xs)"
  apply (induct xs)
  apply auto
  done

lemma rev_rev: "rev (rev xs) = xs"
  apply (induct xs)
-u-:---  DemoList.thy   21% L24    (Isar Utoks Abbrev; Scripting )-----------------

proof (prove): step 2

goal (1 subgoal):
 1. ⋀a xs.
       rev (app xs ys) = app (rev ys) (rev xs) ⟹
       app (app (rev ys) (rev xs)) (Cons a Nil) =
       app (rev ys) (app (rev xs) (Cons a Nil))
-u-:%%-  *goals*        Top L1     (Isar Proofstate Utoks Abbrev;)-----------------
Wrote /Users/lp15/Dropbox/ACS/1 - Introduction/DemoList.thy
```

we dreamt up a lemma...

But it needs another lemma!

Stuck Again!

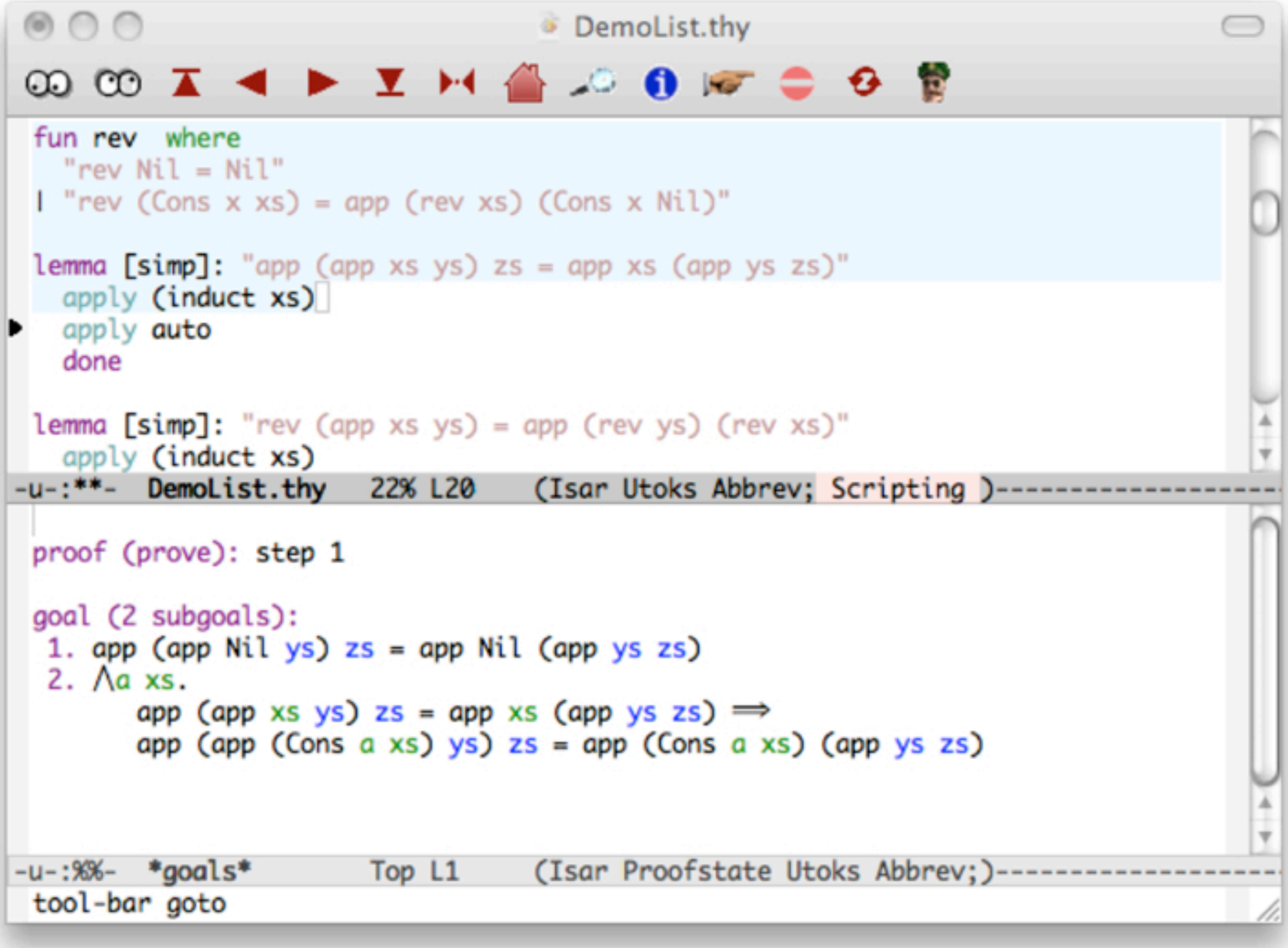


```
fun rev  where
  "rev Nil = Nil"
| "rev (Cons x xs) = app (rev xs) (Cons x Nil)"

lemma [simp]: "rev (app xs ys) = app (rev ys) (rev xs)"
  apply (induct xs)
  apply auto
  done

lemma rev_rev: "rev (rev xs) = xs"
  apply (induct xs)
-u-:---  DemoList.thy   21% L24    (Isar Utoks Abbrev; Scripting )------------------

proof (prove): step 2

goal (1 subgoal):
 1. ⋀a xs.
       rev (app xs ys) = app (rev ys) (rev xs) ⟹
       app (app (rev ys) (rev xs)) (Cons a Nil) =
       app (rev ys) (app (rev xs) (Cons a Nil))

-u-:%%-  *goals*        Top L1     (Isar Proofstate Utoks Abbrev;)------------------
Wrote /Users/lp15/Dropbox/ACS/1 - Introduction/DemoList.thy
```

we dreamt up a lemma...

But it needs another lemma!

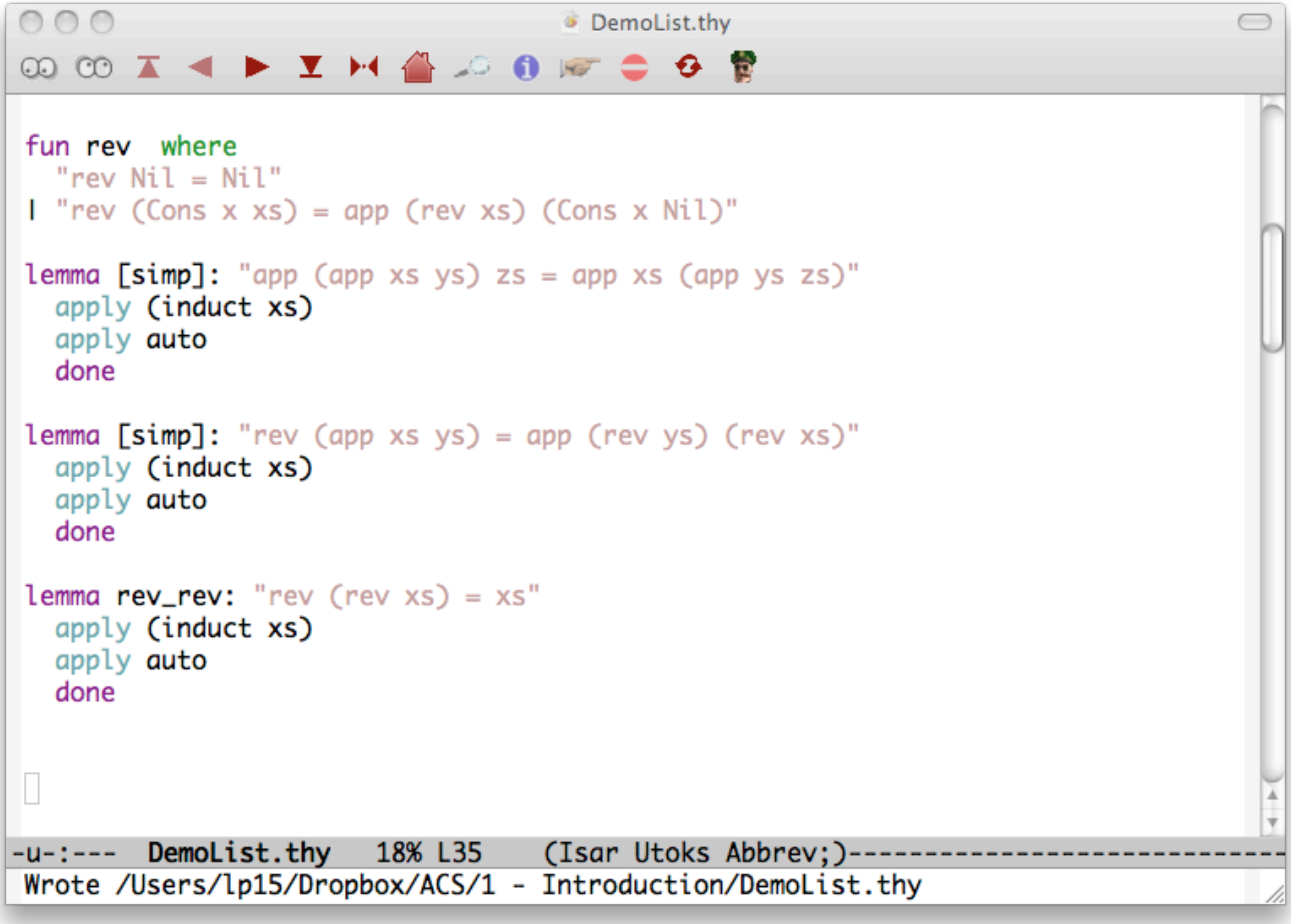
The Final Piece of the Jigsaw



```
fun rev  where
  "rev Nil = Nil"
| "rev (Cons x xs) = app (rev xs) (Cons x Nil)"

lemma [simp]: "app (app xs ys) zs = app xs (app ys zs)"
  apply (induct xs)
  apply auto
  done

lemma [simp]: "rev (app xs ys) = app (rev ys) (rev xs)"
  apply (induct xs)
-u-:**-  DemoList.thy    22% L20     (Isar Utoks Abbrev; Scripting )------------------
```

```
proof (prove): step 1

goal (2 subgoals):
 1. app (app Nil ys) zs = app Nil (app ys zs)
 2. ⋀a xs.
       app (app xs ys) zs = app xs (app ys zs) ⟹
       app (app (Cons a xs) ys) zs = app (Cons a xs) (app ys zs)

-u-:%%-  *goals*         Top L1      (Isar Proofstate Utoks Abbrev;)------------------
tool-bar goto
```

The Finished Proof



```
fun rev  where
  "rev Nil = Nil"
| "rev (Cons x xs) = app (rev xs) (Cons x Nil)"

lemma [simp]: "app (app xs ys) zs = app xs (app ys zs)"
  apply (induct xs)
  apply auto
  done

lemma [simp]: "rev (app xs ys) = app (rev ys) (rev xs)"
  apply (induct xs)
  apply auto
  done

lemma rev_rev: "rev (rev xs) = xs"
  apply (induct xs)
  apply auto
  done
```

```
-u-:---  DemoList.thy   18% L35    (Isar Utoks Abbrev;)---------------------------
Wrote /Users/lp15/Dropbox/ACS/1 - Introduction/DemoList.thy
```