Isabelle Tutorial 1: Verifying Functional Programs

Lawrence C Paulson
Computer Laboratory
University of Cambridge

Isabelle

- Isabelle is a generic interactive theorem prover, developed by Lawrence Paulson (Cambridge) and Tobias Nipkow (Munich). First release in 1986.
- Integrated tool support for
  - Automated provers
  - Counter-example finding
  - Code generation from logical terms
  - LaTeX document generation

Higher-Order Logic

- First-order logic extended with functions and sets
- Polymorphic types, including a type of truth values
- No distinction between terms and formulas
- ML-style functional programming

"HOL = functional programming + logic"

Basic Syntax of Formulas

formulas *A*, *B*, ... can be written as

(*A*)	t = u	~*A*
A & *B*	*A* | *B*	*A* --> *B*
A <-> *B*	ALL *x*. *A*	EX *x*. *A*

(Among many others)

Isabelle also supports symbols such as

≤ ≥ ≠ ∧ ∨ → ↔ ∀ ∃

Basic Syntax of Terms

- The typed λ-calculus:
  - constants, c
  - variables, x and *flexible* variables, ?x
  - abstractions λx. t
  - function applications t u
- Numerous infix operators and binding operators for arithmetic, set theory, etc.

Types

- Every term has a type; Isabelle infers the types of terms automatically. We write $t :: \tau$
- Types can be *polymorphic*, with a system of type classes (inspired by the Haskell language) that allows sophisticated overloading.
- A formula is simply a term of type `bool`.
- There are types of ordered pairs and functions.
- Other important types are those of the natural numbers (`nat`) and integers (`int`).

Function Types

- Infix operators are curried functions
  - `+ :: nat => nat => nat`
  - `& :: bool => bool => bool`
  - Curried function notation: $\lambda x\ y.\ t$
- Function arguments can be paired
  - Example: `nat*nat => nat`
  - Paired function notation: $\lambda(x,y).\ t$

Arithmetic Types

- `nat`: the natural numbers (nonnegative integers)
  - inductively defined: `0`, `Suc` *n*
  - operators include `+ - * div mod`
  - relations include `< ≤ dvd` (divisibility)
- `int`: the integers, with `+ - * div mod` ...
- `rat`, `real`: `+ - * / sin cos ln` ...
- arithmetic constants and laws for these types

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!

name of the new theory

A Tiny Theory

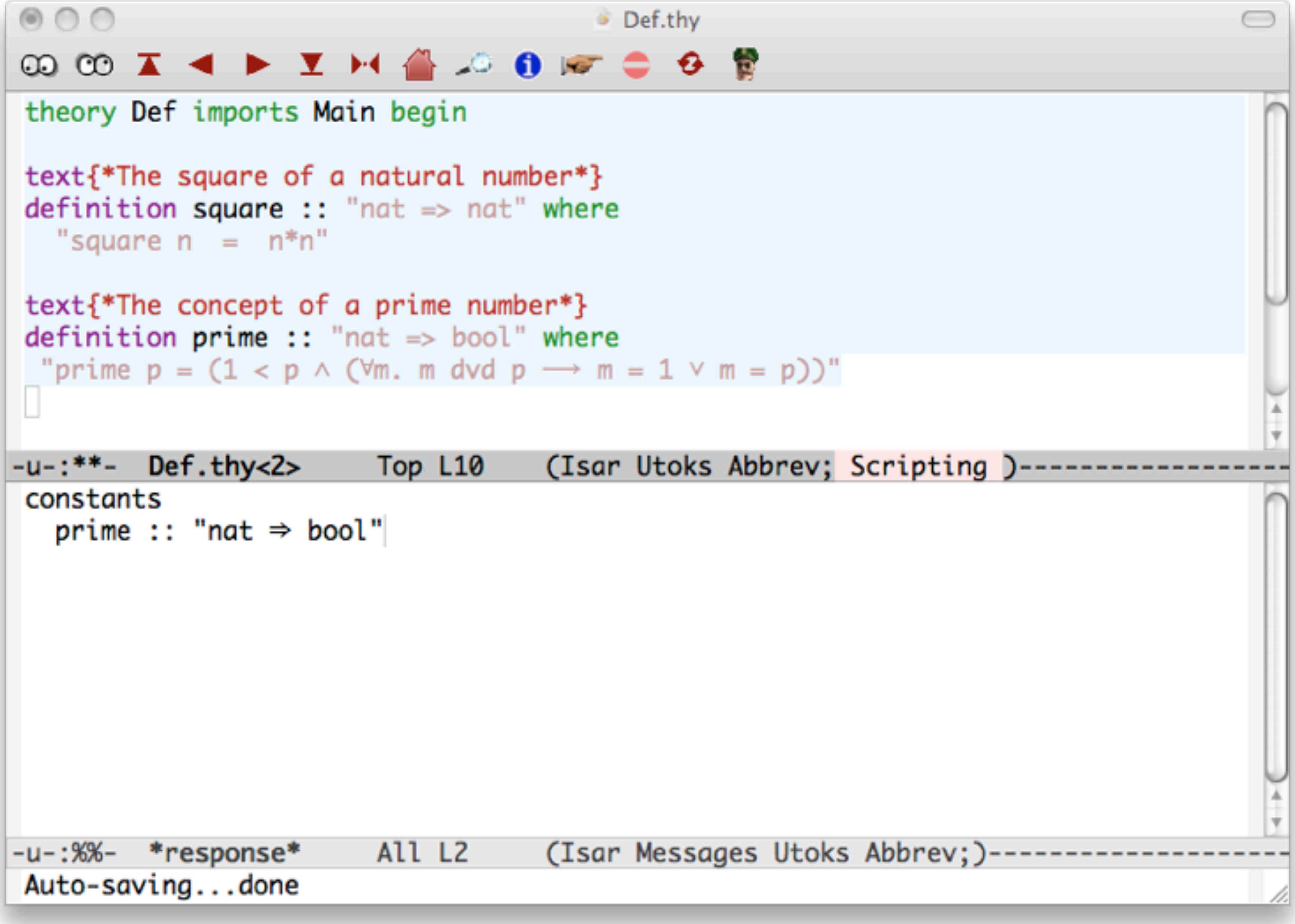
the theory it builds upon

declarations of types, constants, etc

proving a theorem

```
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
```

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 )-----------------
```

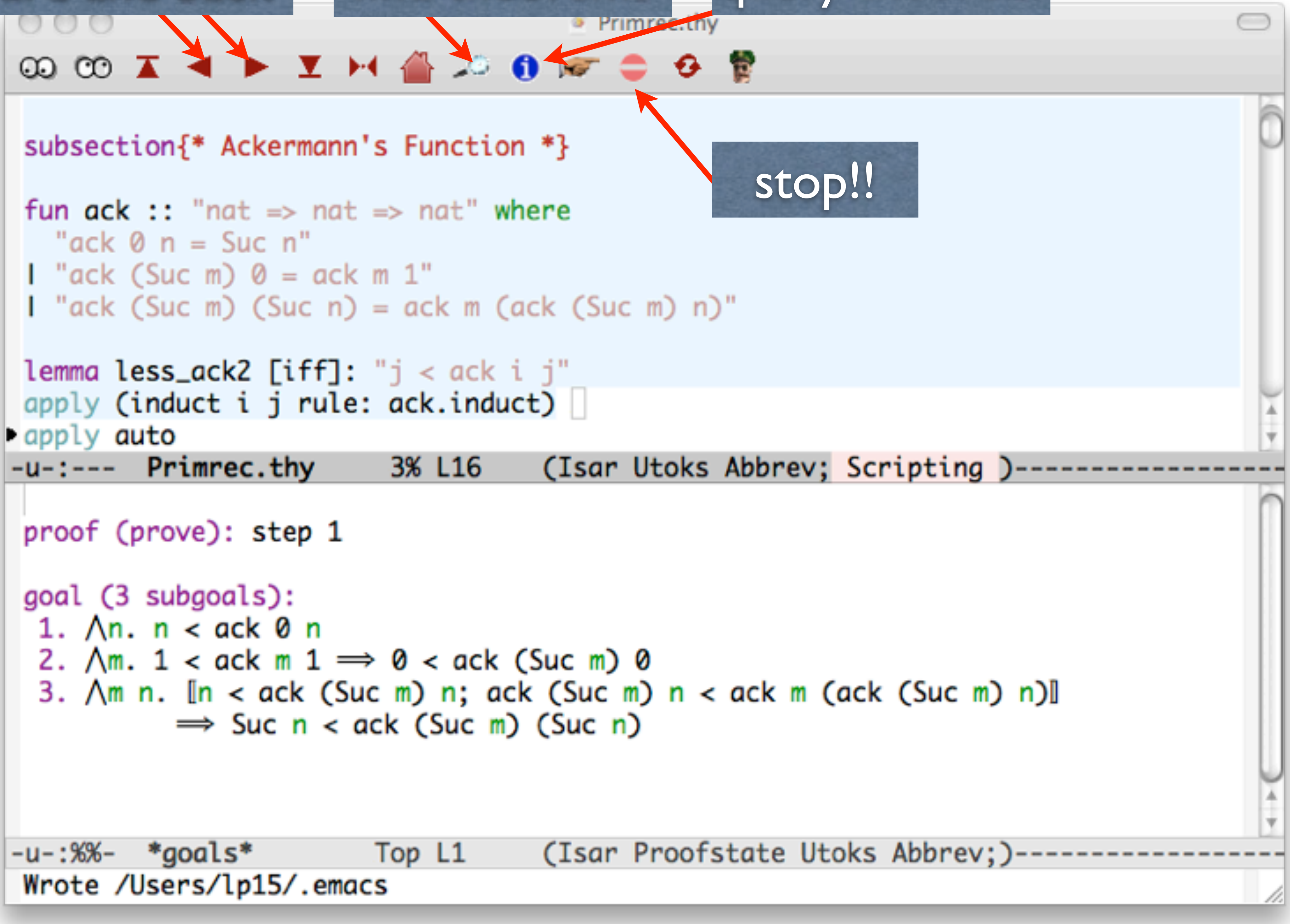
```
constants
  prime :: "nat ⇒ bool"
```

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

Basics of Proof General

- You create or visit an Isabelle theory file within the text editor, Emacs.
- Moving *forward* executes Isabelle commands; the processed text turns blue.
- Moving *backward* undoes those commands.
- *Go to end* processes the entire theory; you can also *go to start*, or go to an arbitrary point in the file.
- *Go to home* takes you to the end of the blue (processed) region.

Proof General Tools



forward and back

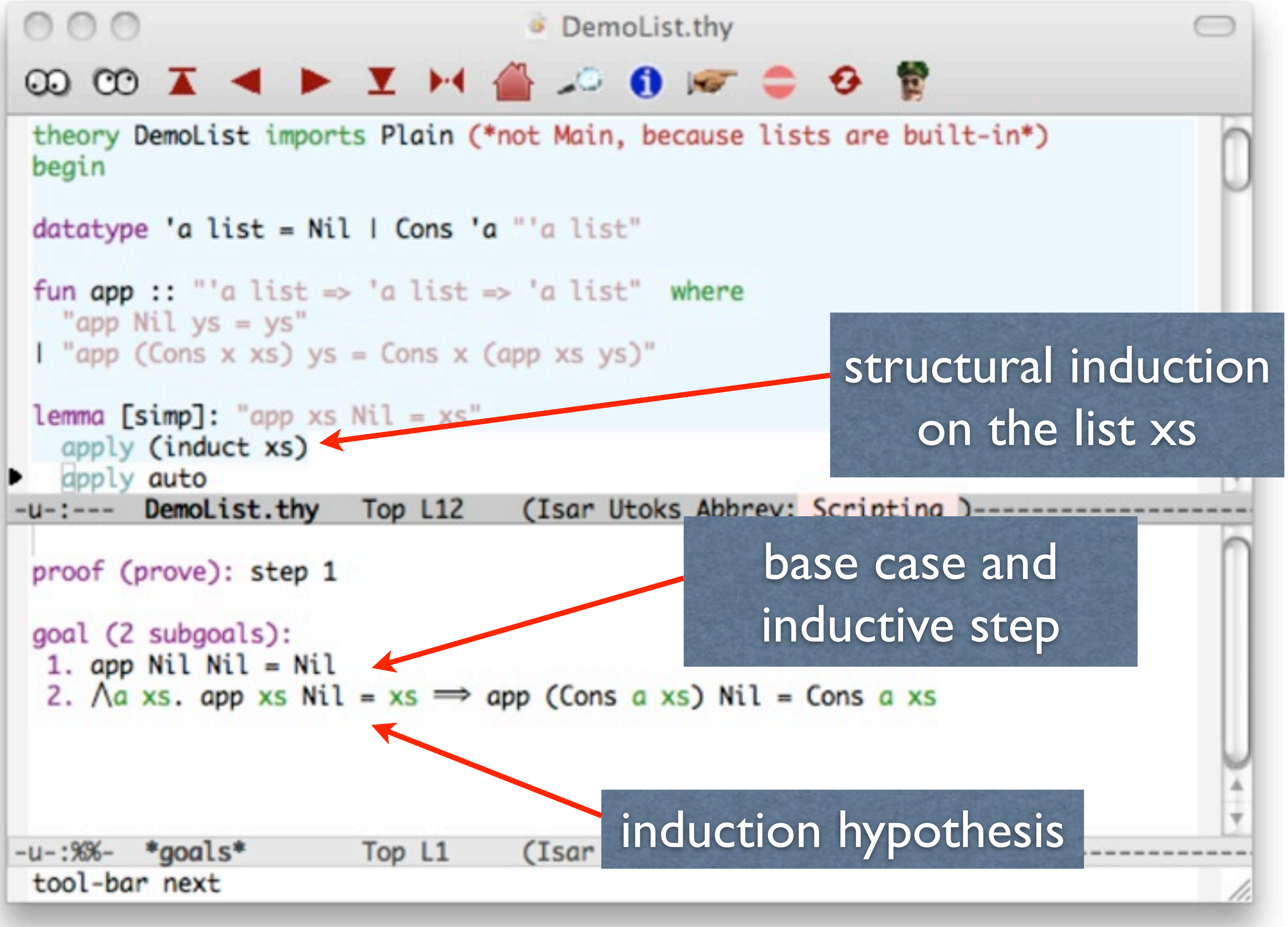
find theorems

query theorem

stop!!

```
subsection{* Ackermann's Function *}

fun ack :: "nat => nat => nat" where
  "ack 0 n = Suc n"
| "ack (Suc m) 0 = ack m 1"
| "ack (Suc m) (Suc n) = ack m (ack (Suc m) n)"

lemma less_ack2 [iff]: "j < ack i j"
apply (induct i j rule: ack.induct)
apply auto
-u-:---  Primrec.thy     3% L16     (Isar Utoks Abbrev; Scripting )-----------------

proof (prove): step 1

goal (3 subgoals):
 1. ⋀n. n < ack 0 n
 2. ⋀m. 1 < ack m 1 ⟹ 0 < ack (Suc m) 0
 3. ⋀m n. ⟦n < ack (Suc m) n; ack (Suc m) n < ack m (ack (Suc m) n)⟧
          ⟹ Suc n < ack (Suc m) (Suc n)

-u-:%%-  *goals*         Top L1     (Isar Proofstate Utoks Abbrev;)-----------------
Wrote /Users/lp15/.emacs
```

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 )------------------
```

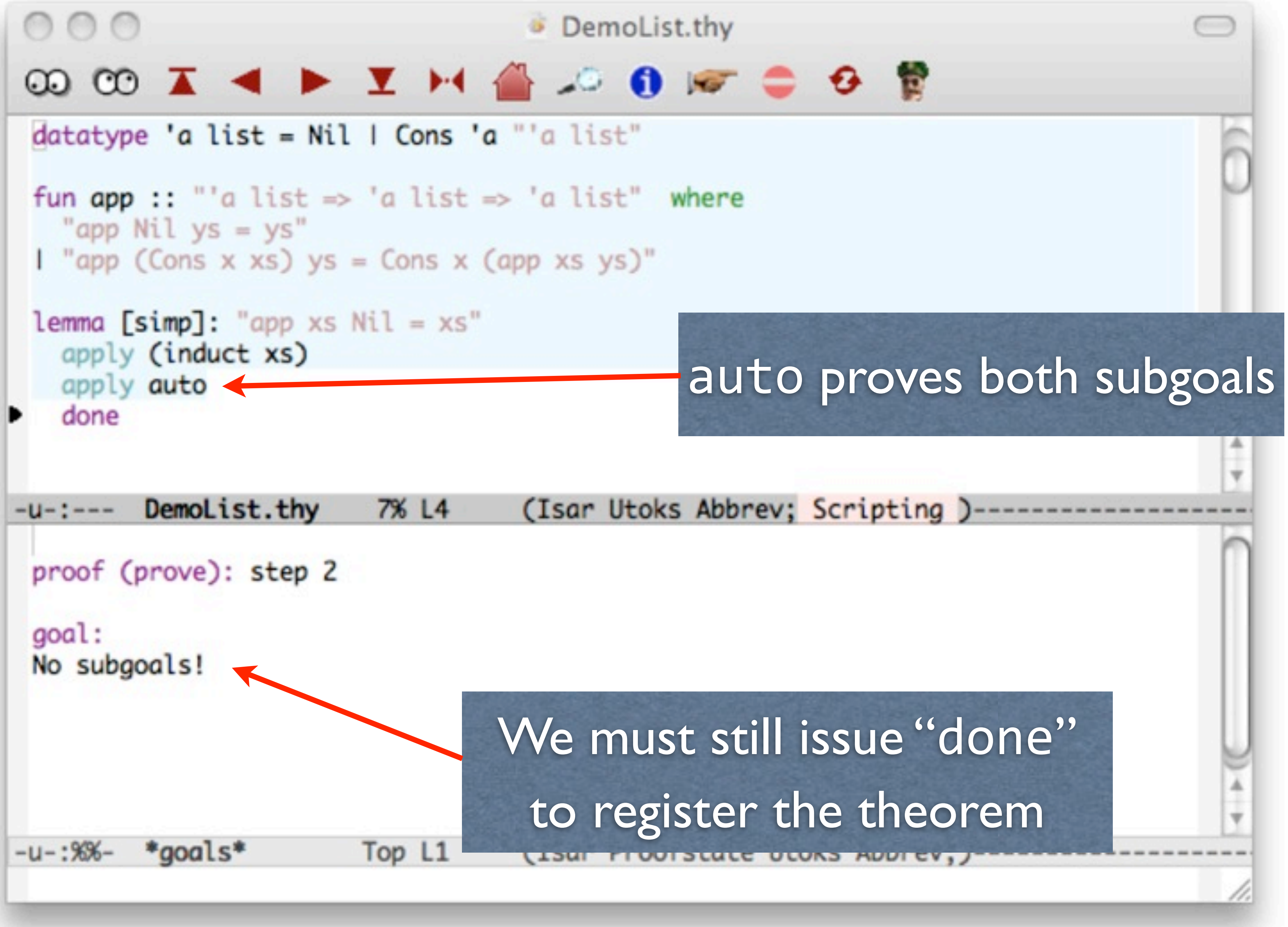
```
proof (prove): step 1

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 )------------------
```

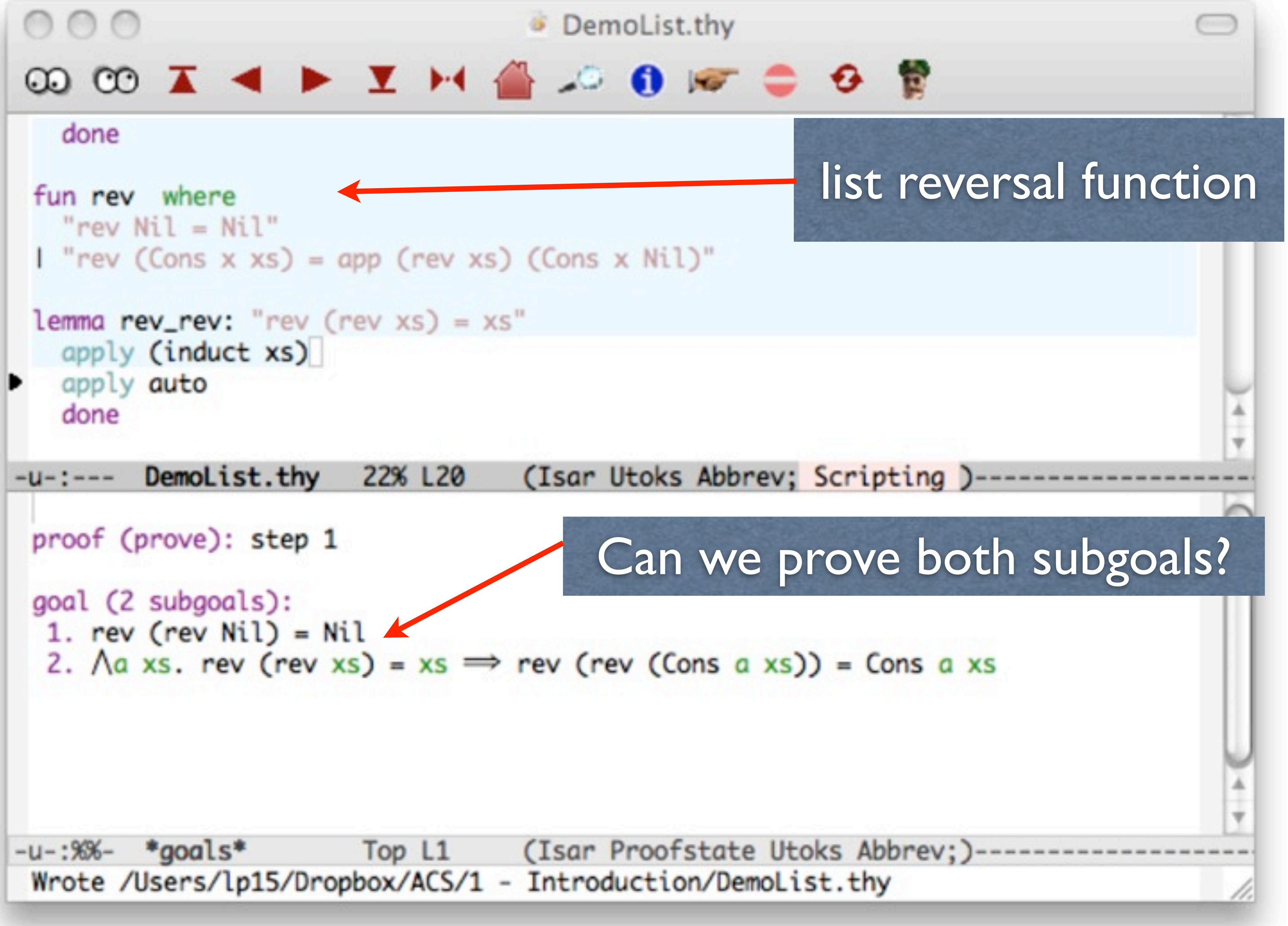
```
proof (prove): step 2

goal:
No subgoals!
```

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

auto proves both subgoals

We must still issue “done” to register the theorem

Another Proof Attempt

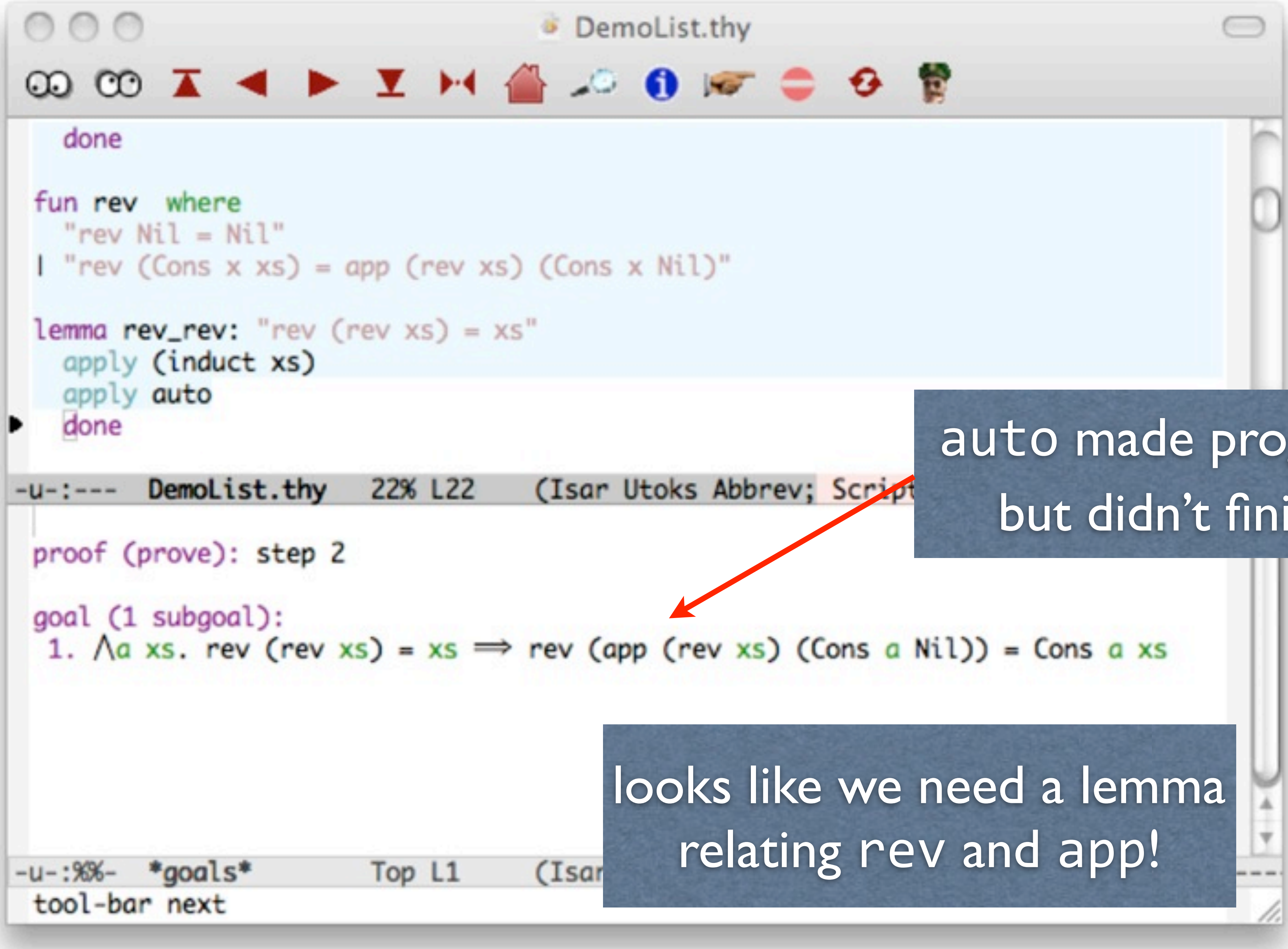


```
  done

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
```

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; Script
```

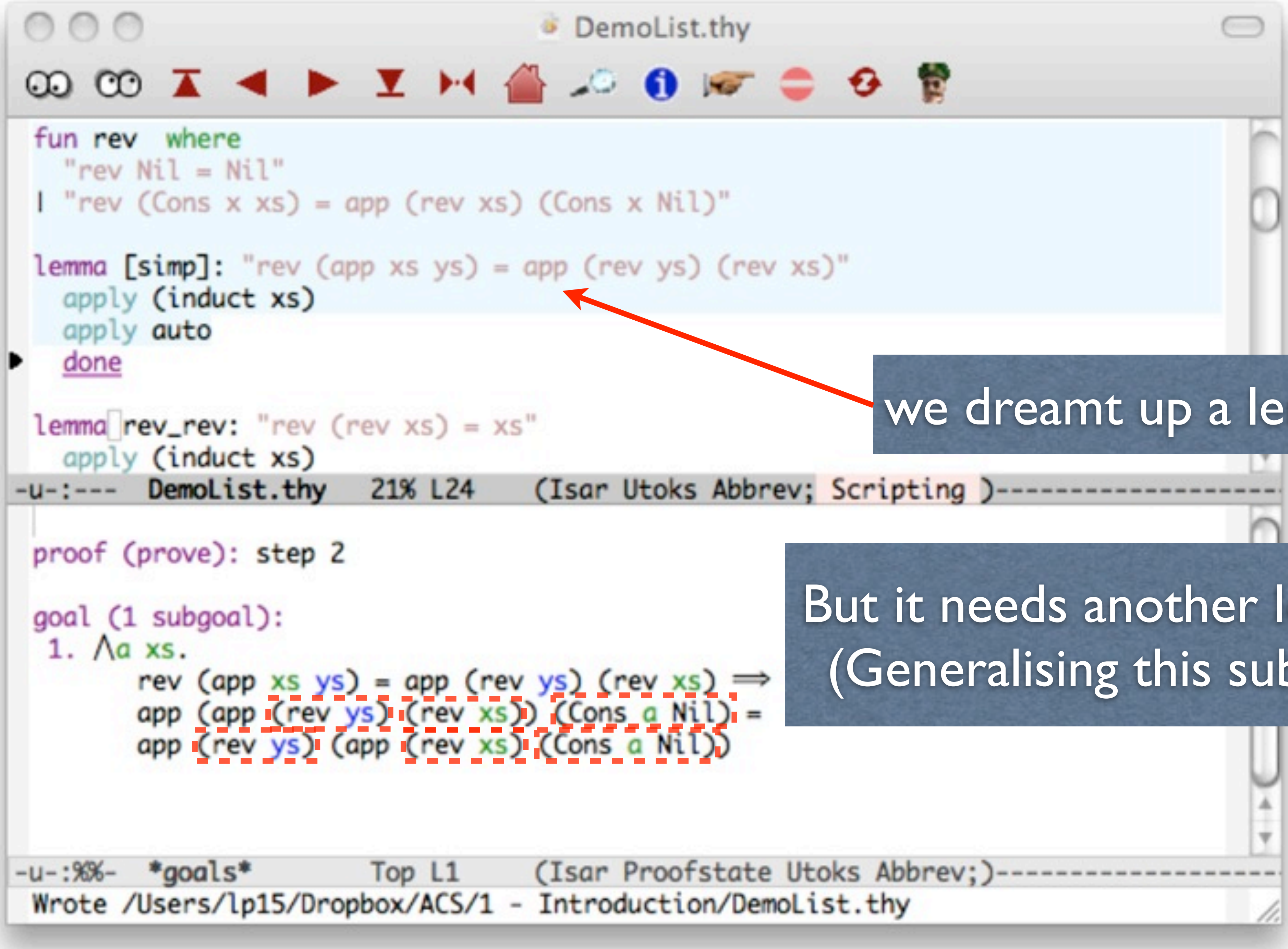
```
proof (prove): step 2

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
tool-bar next
```

auto made progress
but didn't finish

looks like we need a lemma
relating rev and app!

Stuck Again!



DemoList.thy

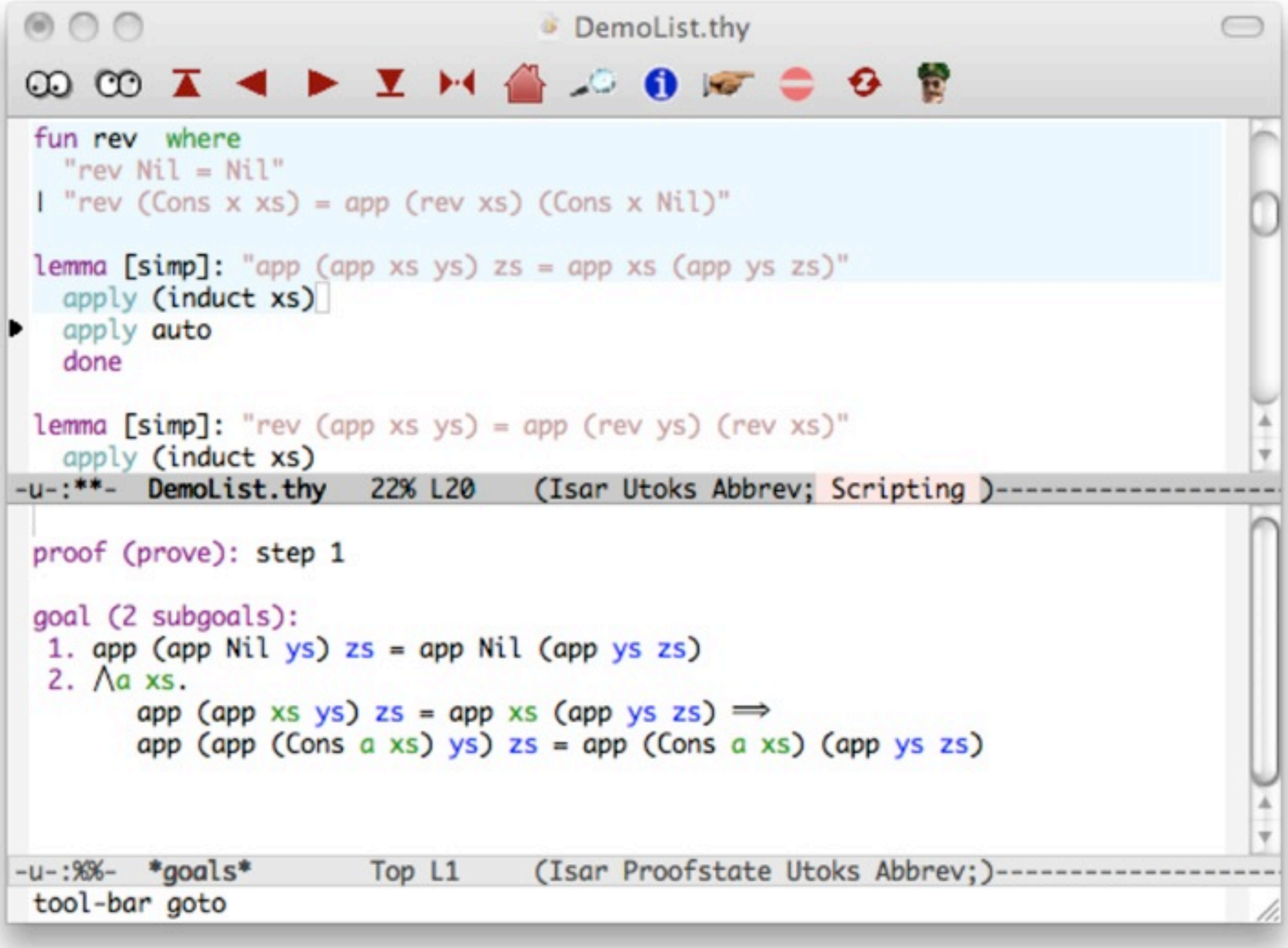
```
fun rev  where
  "rev Nil = Nil"
| "rev (Cons x xs) = app (rev xs) (Cons x Nil)"

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...

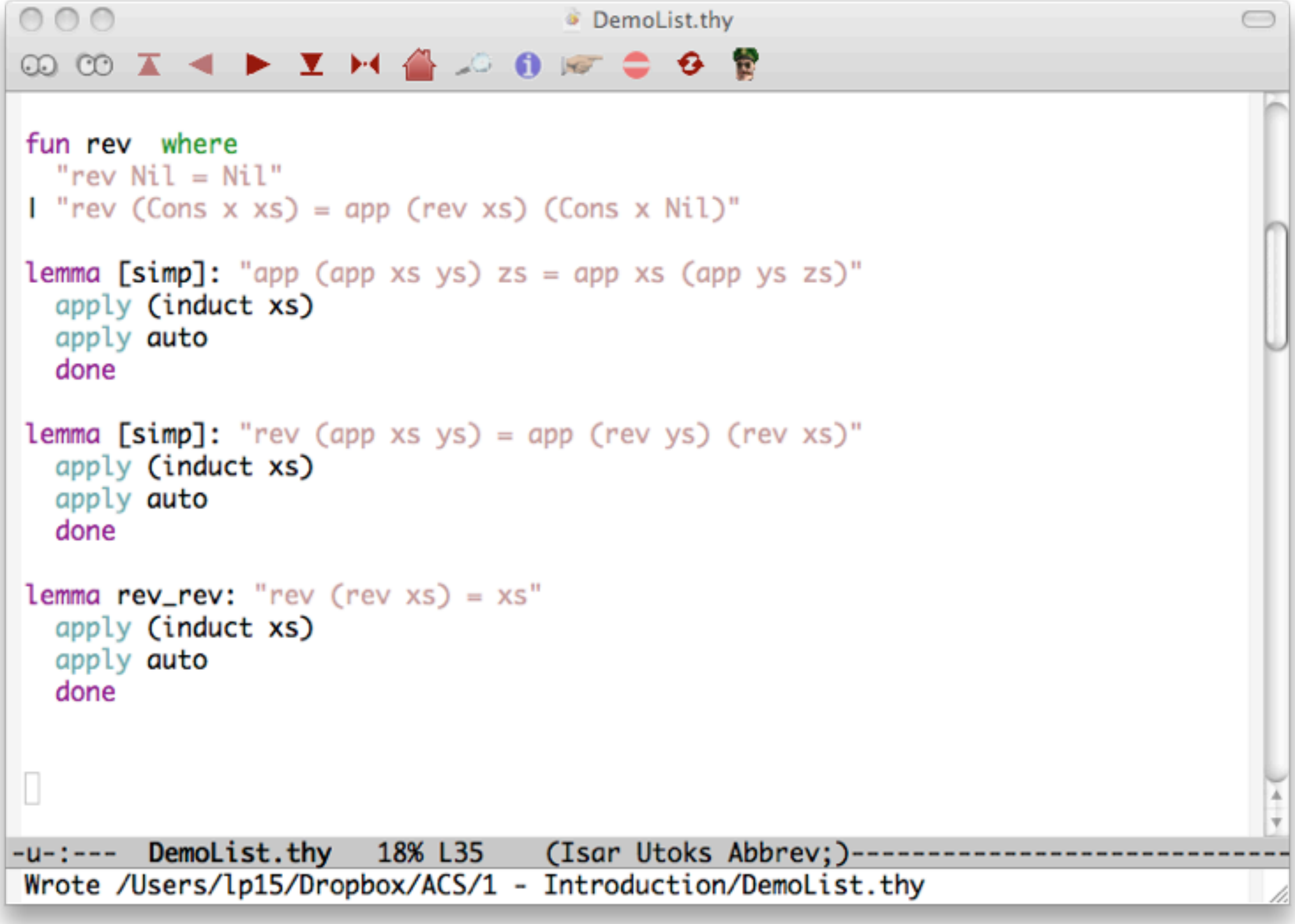
But it needs another lemma!
(Generalising this subgoal)

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
```

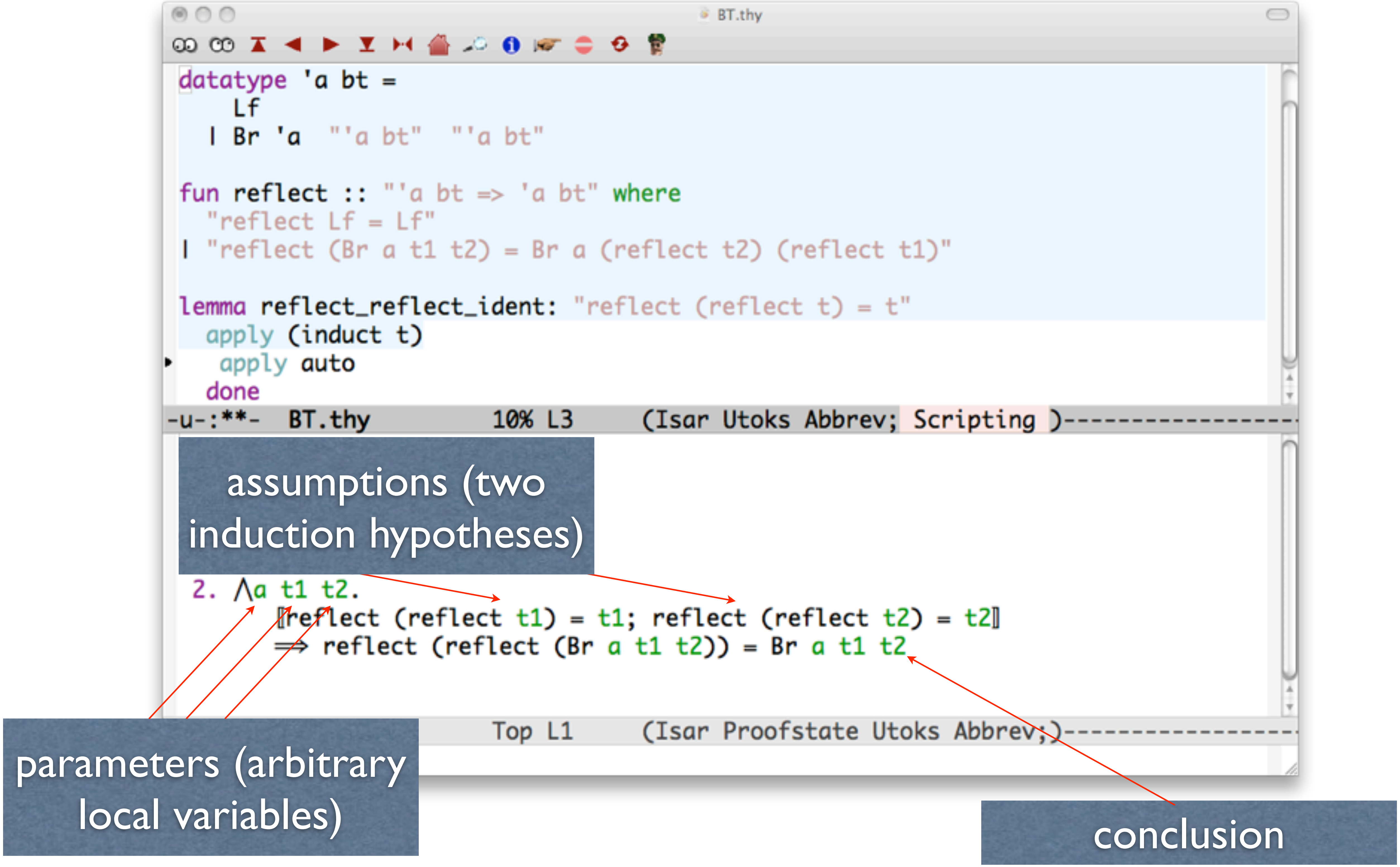
```
-u-:---  DemoList.thy   18% L35    (Isar Utoks Abbrev;)----------------------------
Wrote /Users/lp15/Dropbox/ACS/1 - Introduction/DemoList.thy
```

Now, a deeper look...

Goals and Subgoals

- We start with one subgoal: the statement to be proved.
- Proof *tactics* and *methods* typically replace a single subgoal by zero or more new subgoals.
  - But certain methods, notably `auto` and `simp_all`, operate on *all* outstanding subgoals.
- We finish when no subgoals remain. *The theorem is proved!*

Structure of a Subgoal



```
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
-u-:**-  BT.thy        10% L3      (Isar Utoks Abbrev; Scripting )-----------------
```

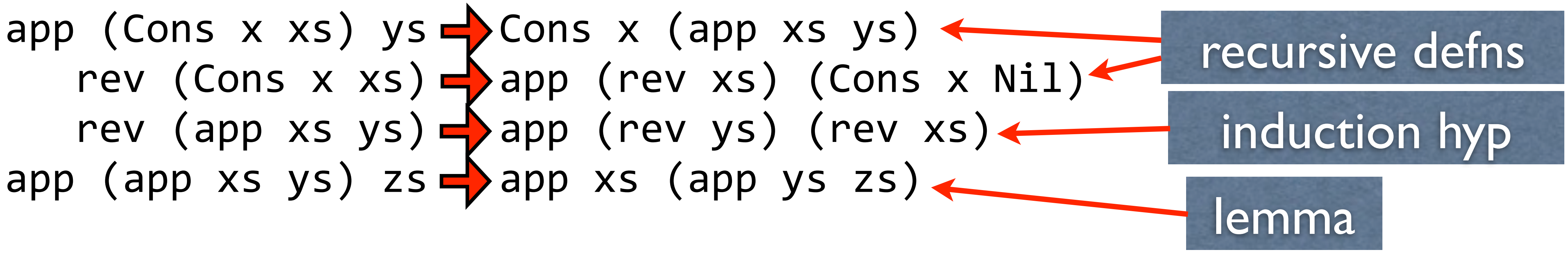
assumptions (two induction hypotheses)

```
 2. ⋀a t1 t2.
       ⟦reflect (reflect t1) = t1; reflect (reflect t2) = t2⟧
       ⟹ reflect (reflect (Br a t1 t2)) = Br a t1 t2
                        Top L1      (Isar Proofstate Utoks Abbrev;)-----------------
```

parameters (arbitrary local variables)

conclusion

Proof by Rewriting



```
app (Cons x xs) ys → Cons x (app xs ys)
   rev (Cons x xs) → app (rev xs) (Cons x Nil)
   rev (app xs ys) → app (rev ys) (rev xs)
app (app xs ys) zs → app xs (app ys zs)
```

recursive defns

induction hyp

lemma

```
rev (app (Cons a xs) ys) = app (rev ys) (rev (Cons a xs))
```

```
rev (app (Cons a xs) ys) =
rev (Cons a (app xs ys)) =
app (rev (app xs ys)) (Cons a Nil) =
app (app (rev ys) (rev xs)) (Cons a Nil) =
app (rev ys) (app (rev xs) (Cons a Nil))
```

```
app (rev ys) (rev (Cons a xs)) =
app (rev ys) (app (rev xs) (Cons a Nil))
```

Conditional Rewrite Rules

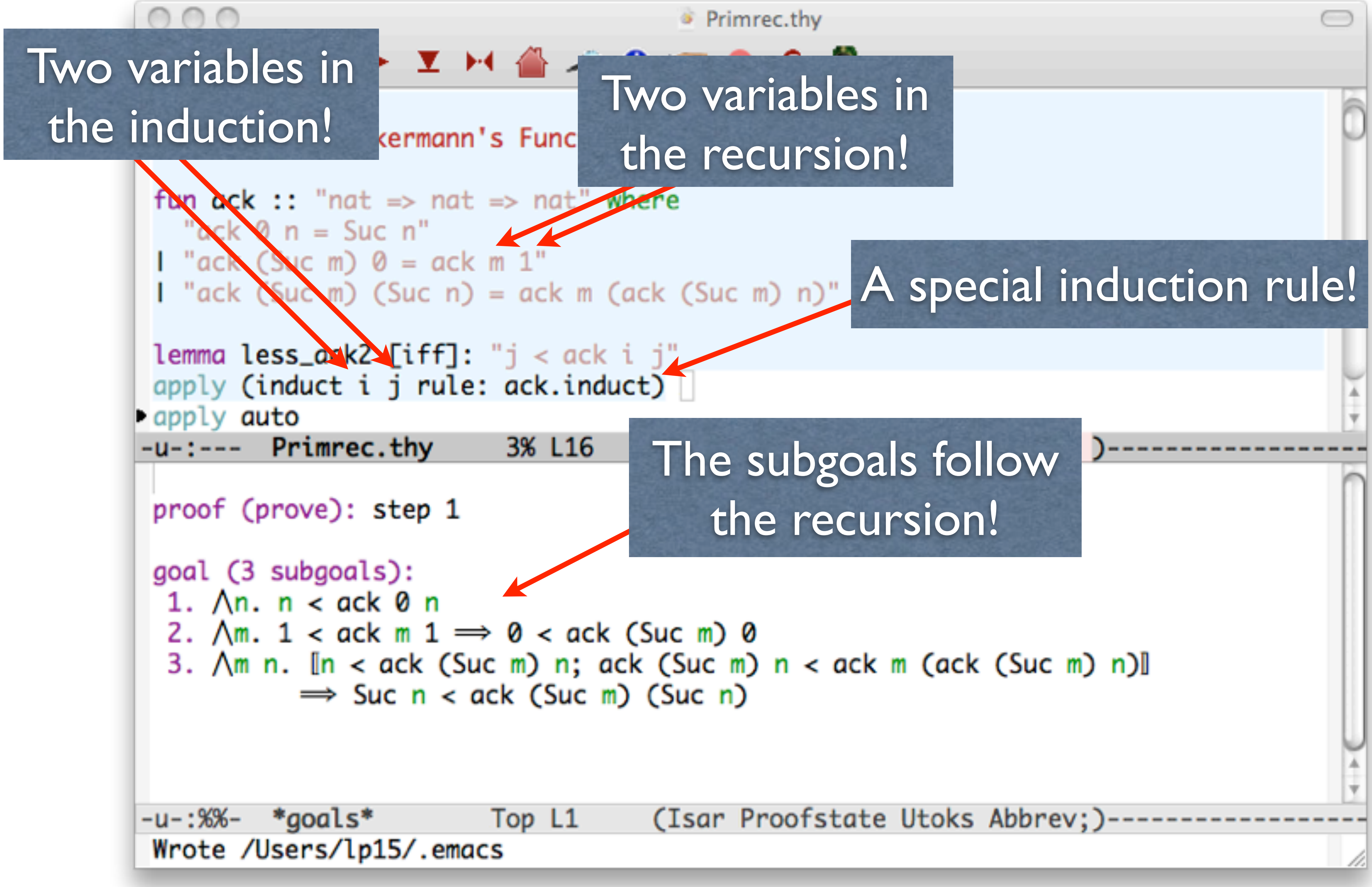
```
xs ≠ [] ⇒ hd (xs @ ys) = hd xs

n ≤ m ⇒ (Suc m) - n = Suc (m - n)

[|a ≠ 0; b ≠ 0|] ⇒ b / (a*b) = 1 / a
```

- *First* match the left-hand side, *then* **recursively** prove the conditions by simplification.
- If successful, applying the resulting rewrite rule.

The Methods `simp` and `auto`

- `simp` performs *rewriting* (along with simple arithmetic simplification) on the *first* subgoal
- `auto` simplifies *all subgoals*, not just the first.
- `auto` also applies all obvious *logical steps*
  - Splitting conjunctive goals and disjunctive assumptions
  - Performing obvious quantifier removal

Unusual Recursions



```
kermann's Func

fun ack :: "nat => nat => nat" where
  "ack 0 n = Suc n"
| "ack (Suc m) 0 = ack m 1"
| "ack (Suc m) (Suc n) = ack m (ack (Suc m) n)"

lemma less_ack2 [iff]: "j < ack i j"
apply (induct i j rule: ack.induct)
apply auto
-u-:---  Primrec.thy    3% L16                              )----------------

proof (prove): step 1

goal (3 subgoals):
 1. ⋀n. n < ack 0 n
 2. ⋀m. 1 < ack m 1 ⟹ 0 < ack (Suc m) 0
 3. ⋀m n. ⟦n < ack (Suc m) n; ack (Suc m) n < ack m (ack (Suc m) n)⟧
          ⟹ Suc n < ack (Suc m) (Suc n)
-u-:%%-  *goals*       Top L1     (Isar Proofstate Utoks Abbrev;)----------------
Wrote /Users/lp15/.emacs
```

Two variables in the induction!

Two variables in the recursion!

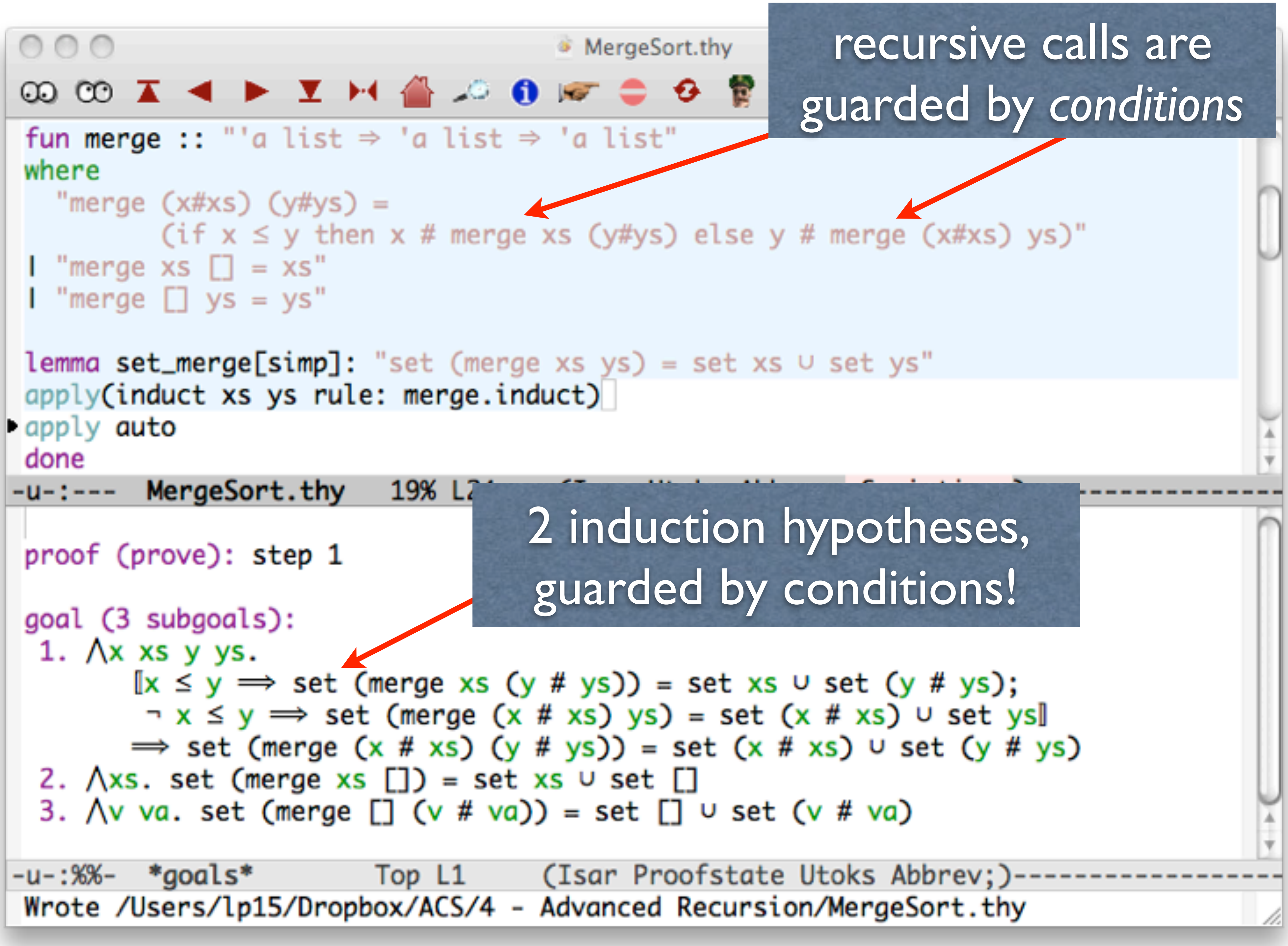
A special induction rule!

The subgoals follow the recursion!

Recursion: Key Points

- Recursion in one variable, following the structure of a datatype declaration, is called *primitive*.
- Recursion in multiple variables, terminating by size considerations, can be handled using `fun`.
  - `fun` produces a special induction rule.
  - `fun` can handle **nested recursion**.
  - `fun` also handles *pattern matching*, which it **completes**.

Another Unusual Recursion



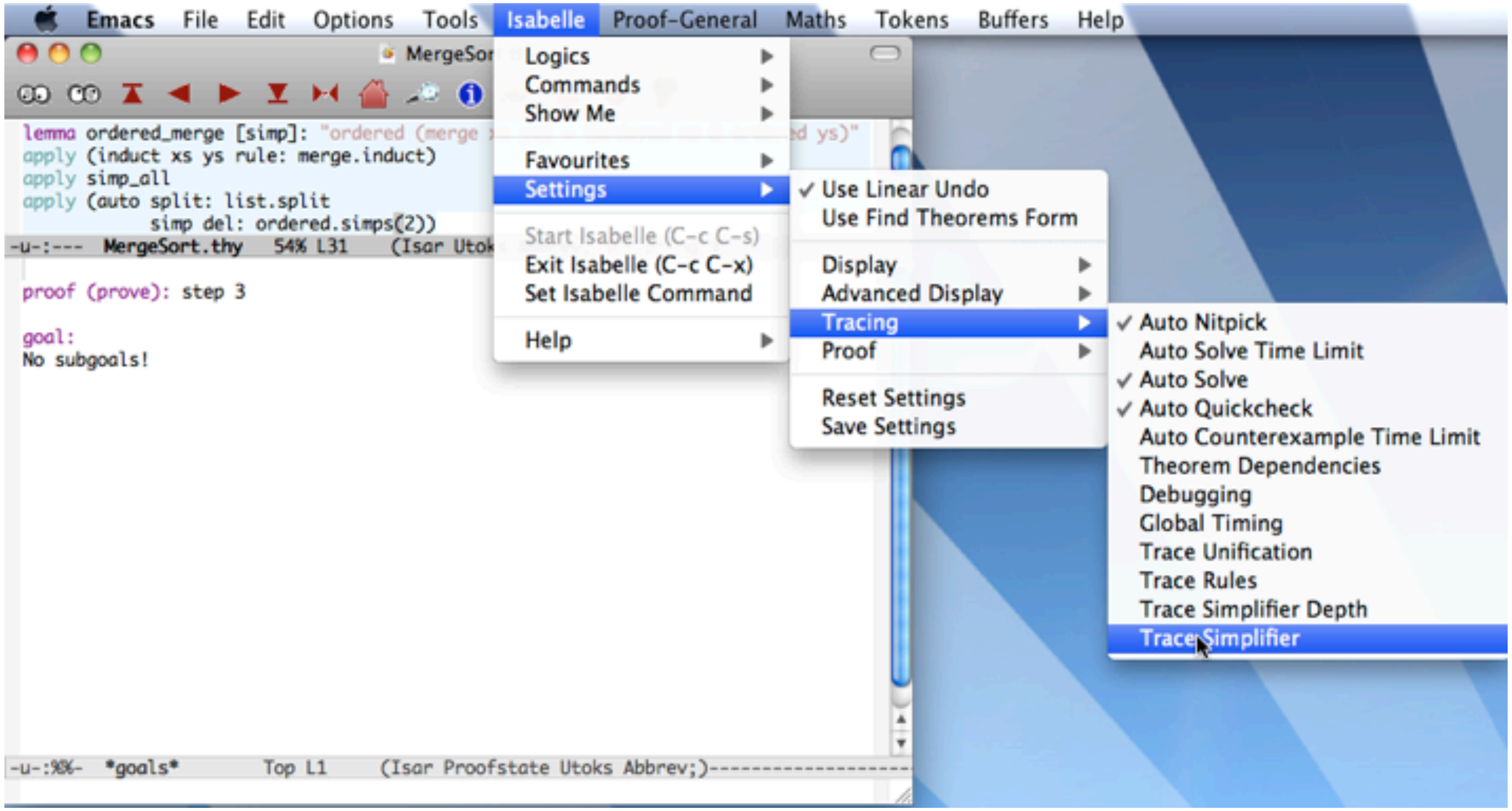
```
fun merge :: "'a list ⇒ 'a list ⇒ 'a list"
where
  "merge (x#xs) (y#ys) =
       (if x ≤ y then x # merge xs (y#ys) else y # merge (x#xs) ys)"
| "merge xs [] = xs"
| "merge [] ys = ys"

lemma set_merge[simp]: "set (merge xs ys) = set xs ∪ set ys"
apply(induct xs ys rule: merge.induct)
apply auto
done
-u-:---  MergeSort.thy    19% L2
```

```
proof (prove): step 1

goal (3 subgoals):
 1. ⋀x xs y ys.
       ⟦x ≤ y ⟹ set (merge xs (y # ys)) = set xs ∪ set (y # ys);
        ¬ x ≤ y ⟹ set (merge (x # xs) ys) = set (x # xs) ∪ set ys⟧
       ⟹ set (merge (x # xs) (y # ys)) = set (x # xs) ∪ set (y # ys)
 2. ⋀xs. set (merge xs []) = set xs ∪ set []
 3. ⋀v va. set (merge [] (v # va)) = set [] ∪ set (v # va)
-u-:%%-  *goals*        Top L1     (Isar Proofstate Utoks Abbrev;)----------------
Wrote /Users/lp15/Dropbox/ACS/4 - Advanced Recursion/MergeSort.thy
```

recursive calls are guarded by *conditions*

2 induction hypotheses, guarded by conditions!

A Helpful Tip



```
lemma ordered_merge [simp]: "ordered (merge            d ys)"
apply (induct xs ys rule: merge.induct)
apply simp_all
apply (auto split: list.split
          simp del: ordered.simps(2))
-u-:---  MergeSort.thy    54% L31     (Isar Utok
```

```
proof (prove): step 3

goal:
No subgoals!
```

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