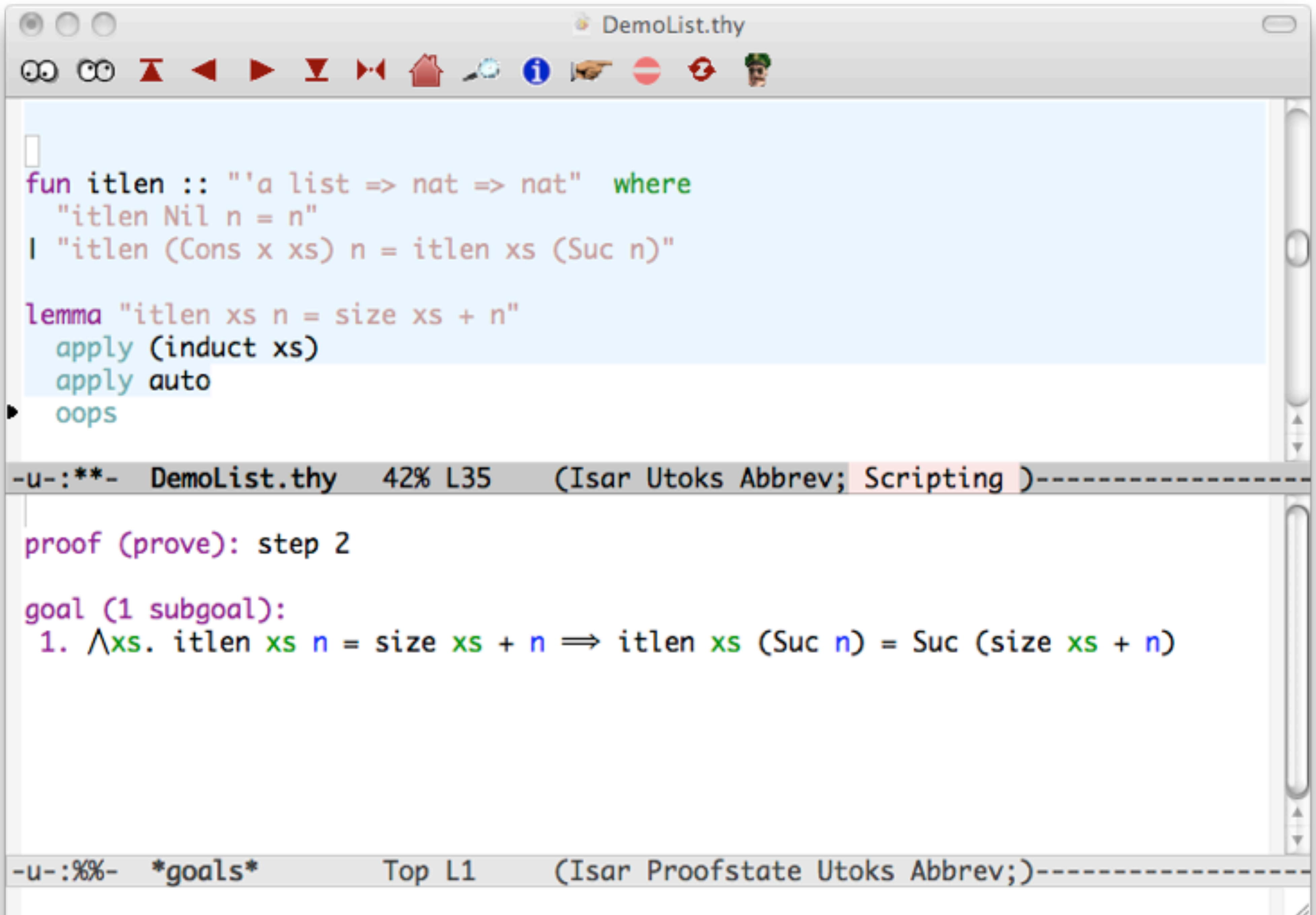
Interactive Formal Verification
4: Advanced Recursion, Induction and Simplification

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

A Failing Proof by Induction



A Failing Proof by Induction

length of a list
(tail-recursive)

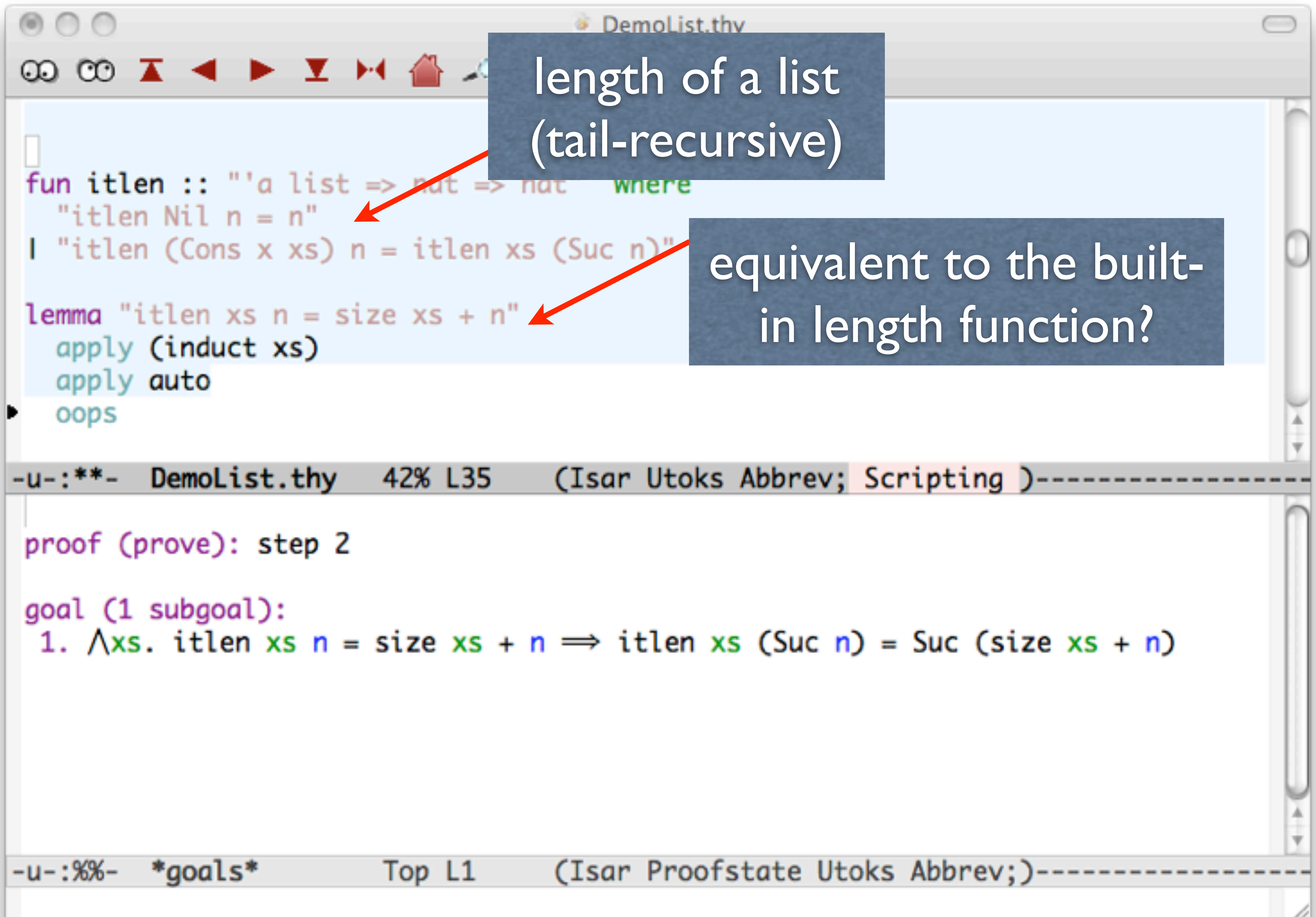
```
fun itlen :: "'a list => nat => nat"  where
  "itlen Nil n = n"
| "itlen (Cons x xs) n = itlen xs (Suc n)"

lemma "itlen xs n = size xs + n"
  apply (induct xs)
  apply auto
  oops
```

```
-u-:**-  DemoList.thy    42% L35    (Isar Utoks Abbrev; Scripting )----------------
```

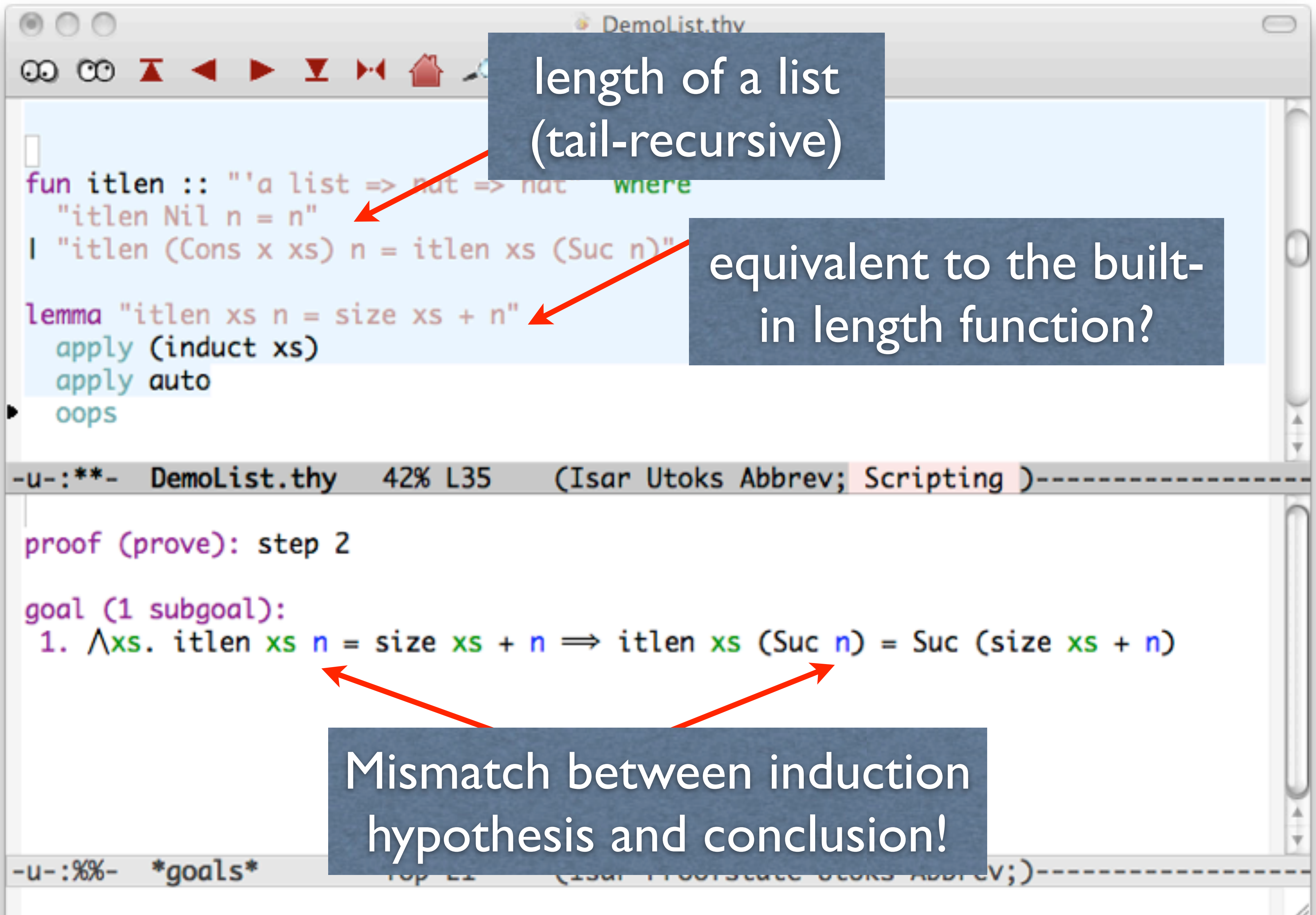
```
proof (prove): step 2

goal (1 subgoal):
 1. ⋀xs. itlen xs n = size xs + n ⟹ itlen xs (Suc n) = Suc (size xs + n)
```

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

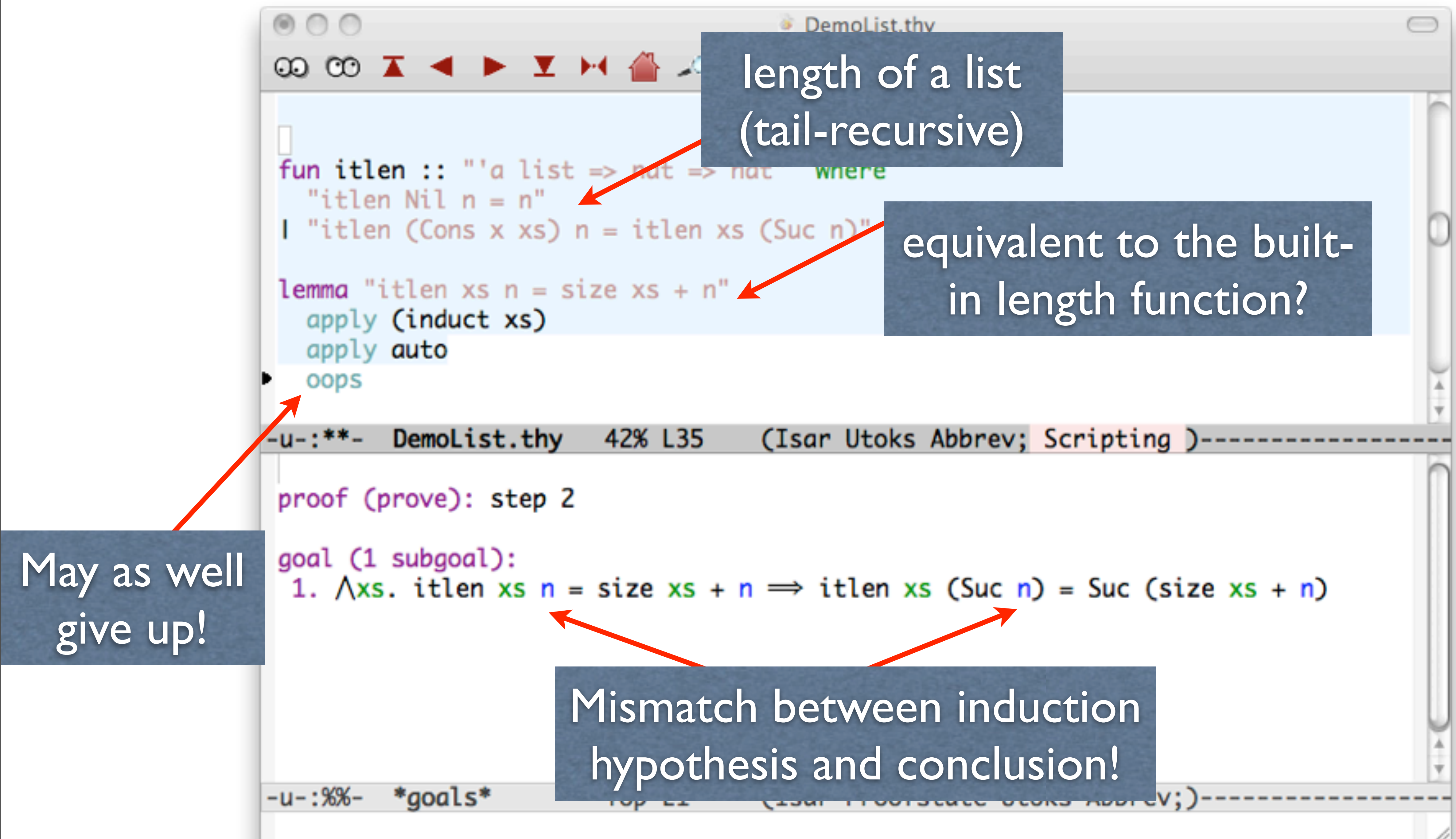
A Failing Proof by Induction



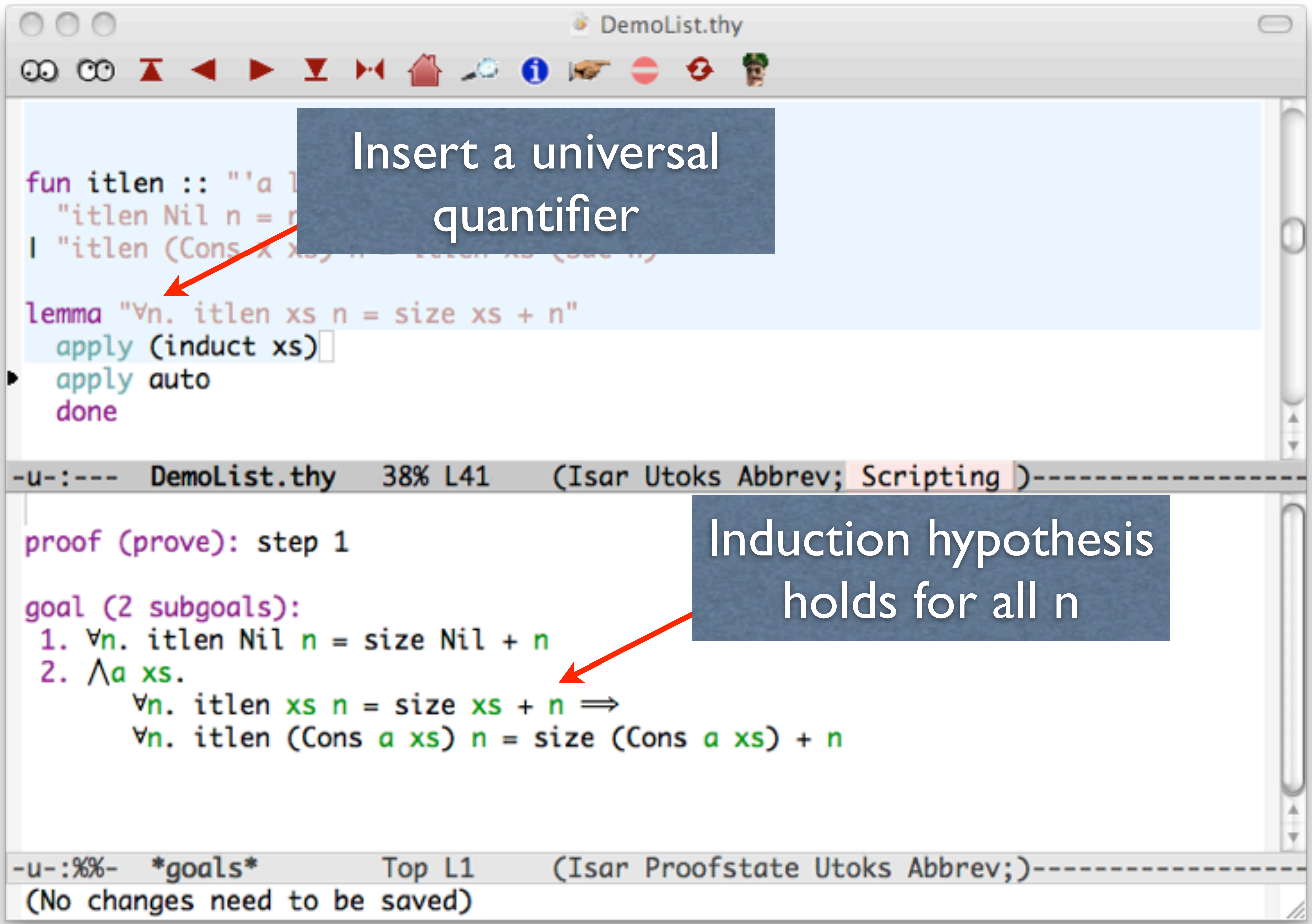
A Failing Proof by Induction



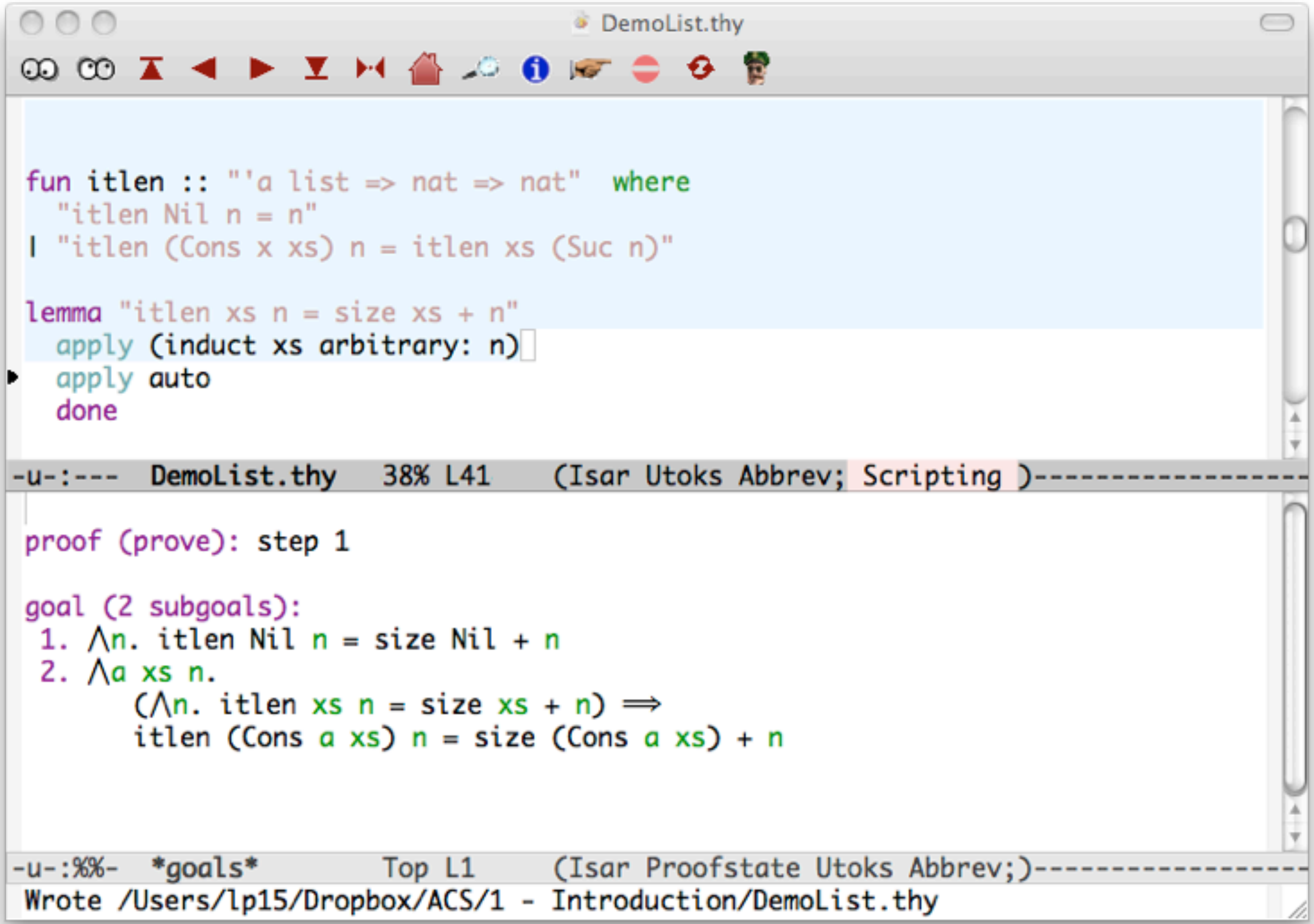
A Failing Proof by Induction



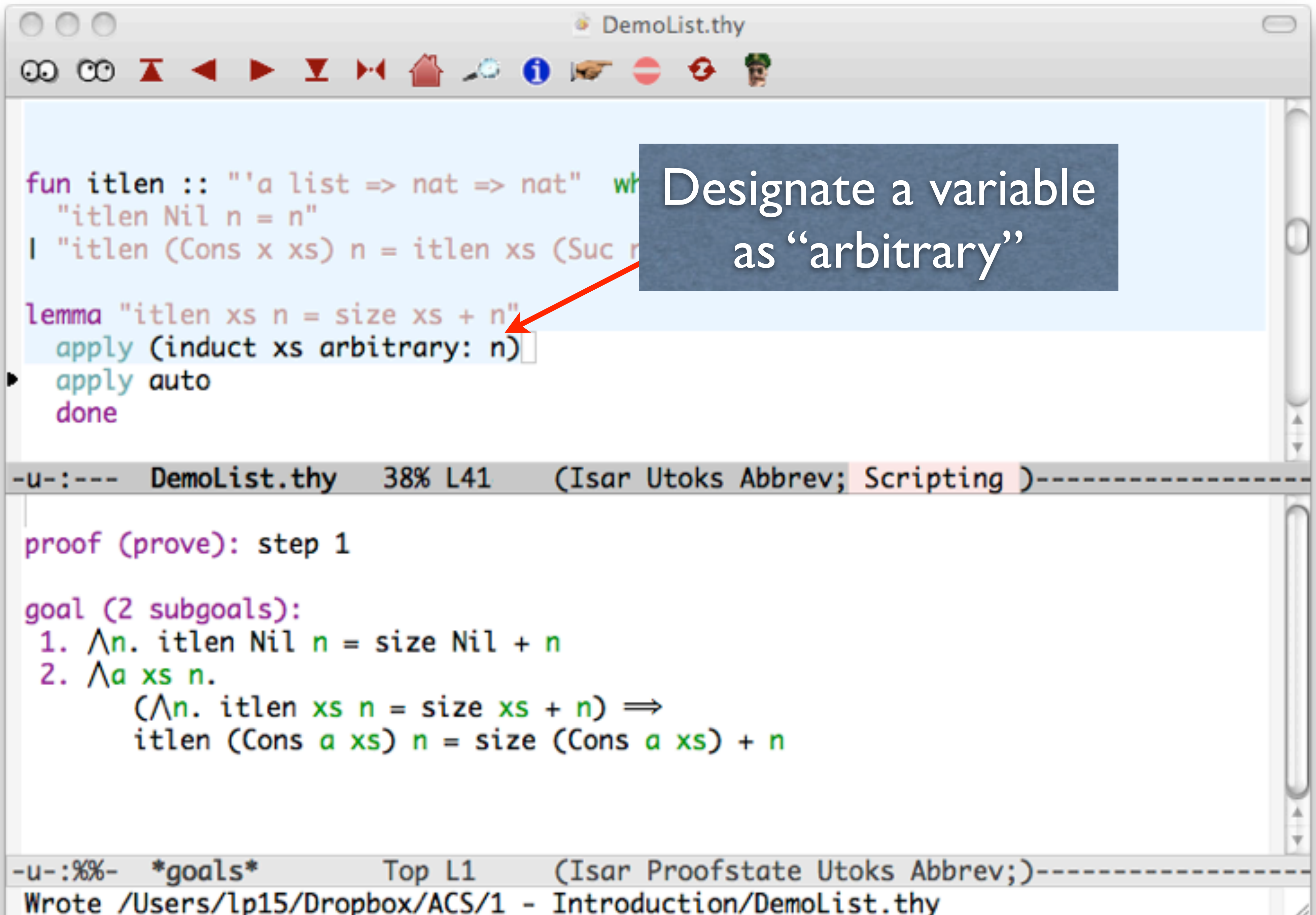
Generalising the Induction



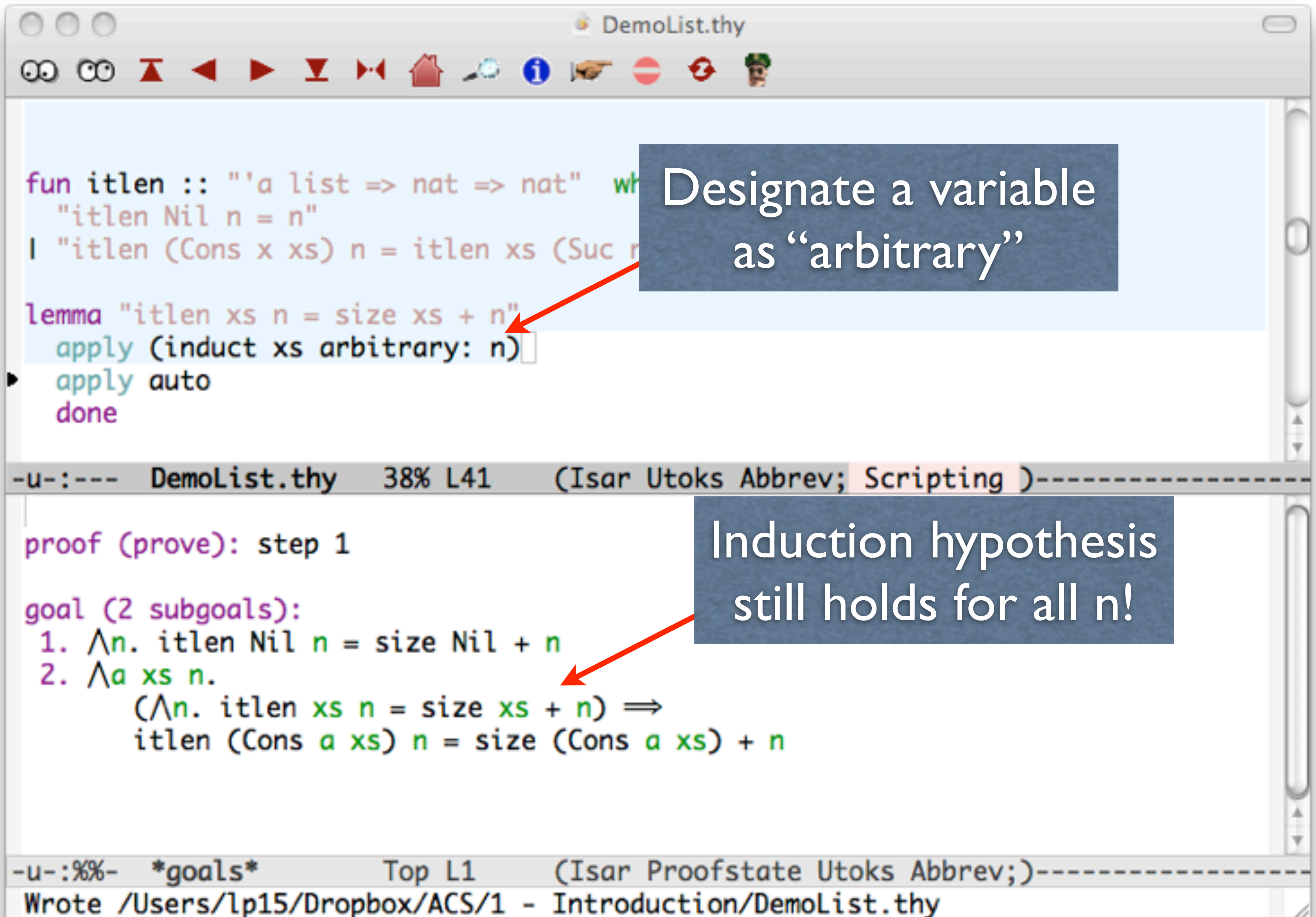
Generalising: Another Way



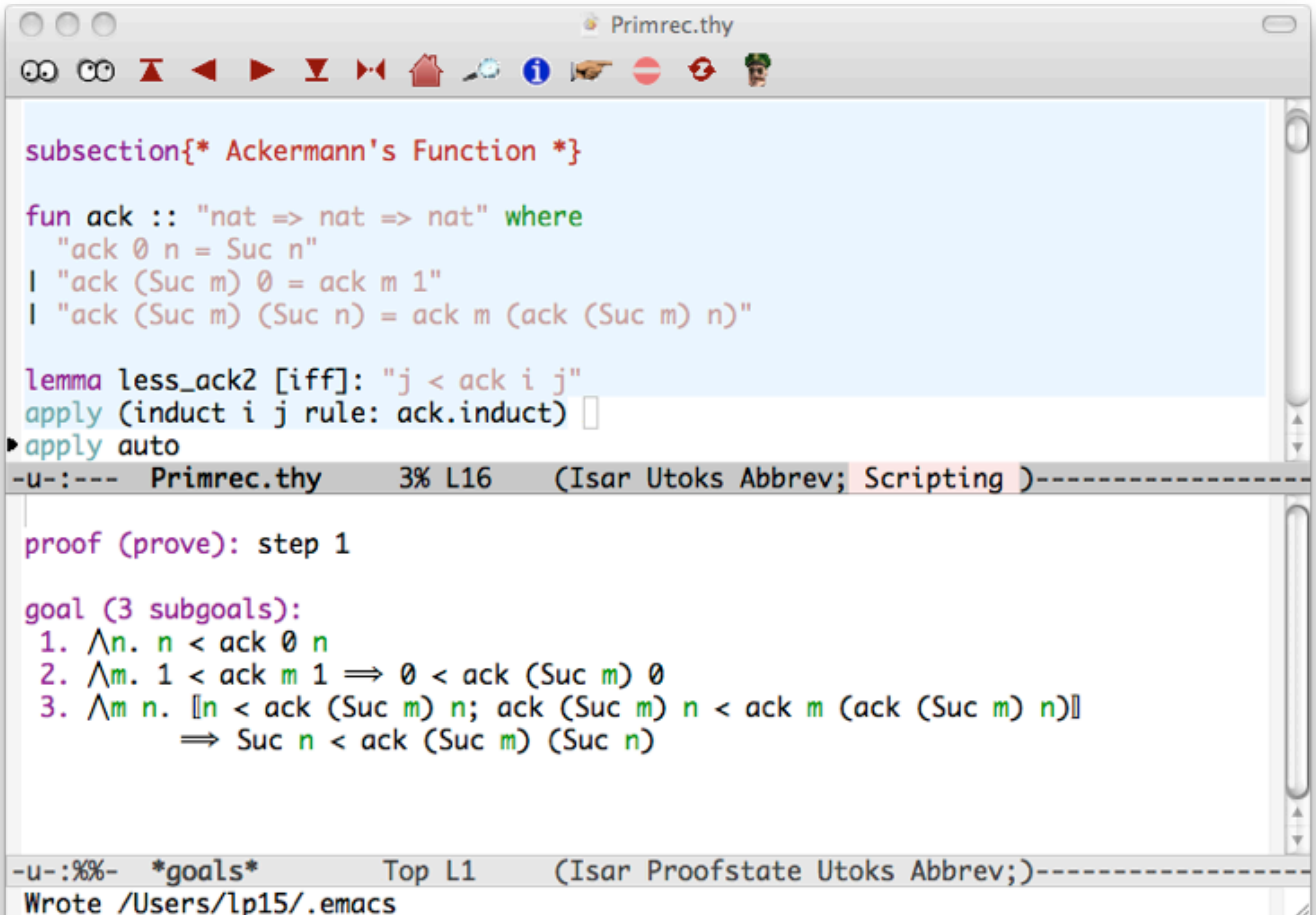
Generalising: Another Way



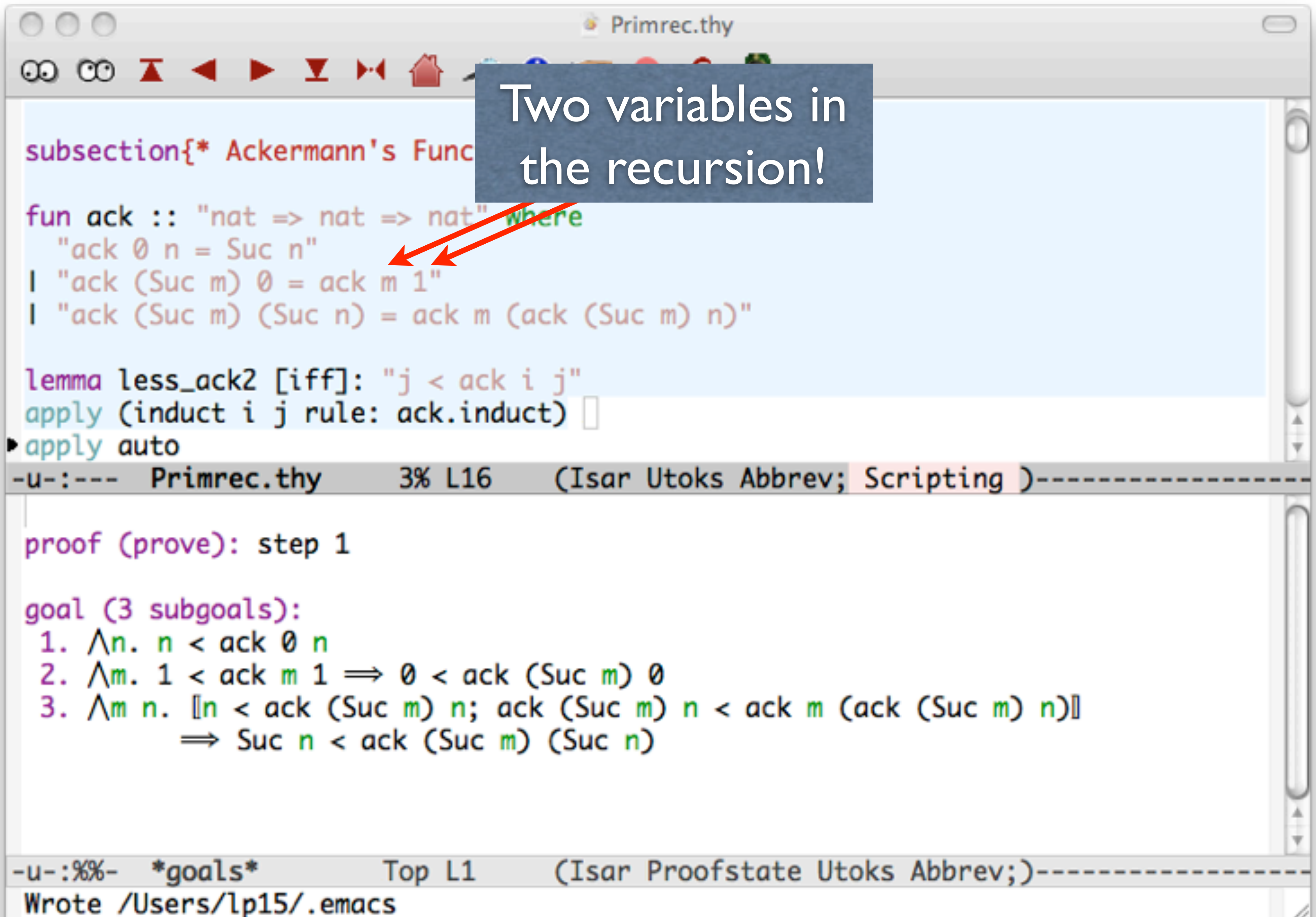
Generalising: Another Way



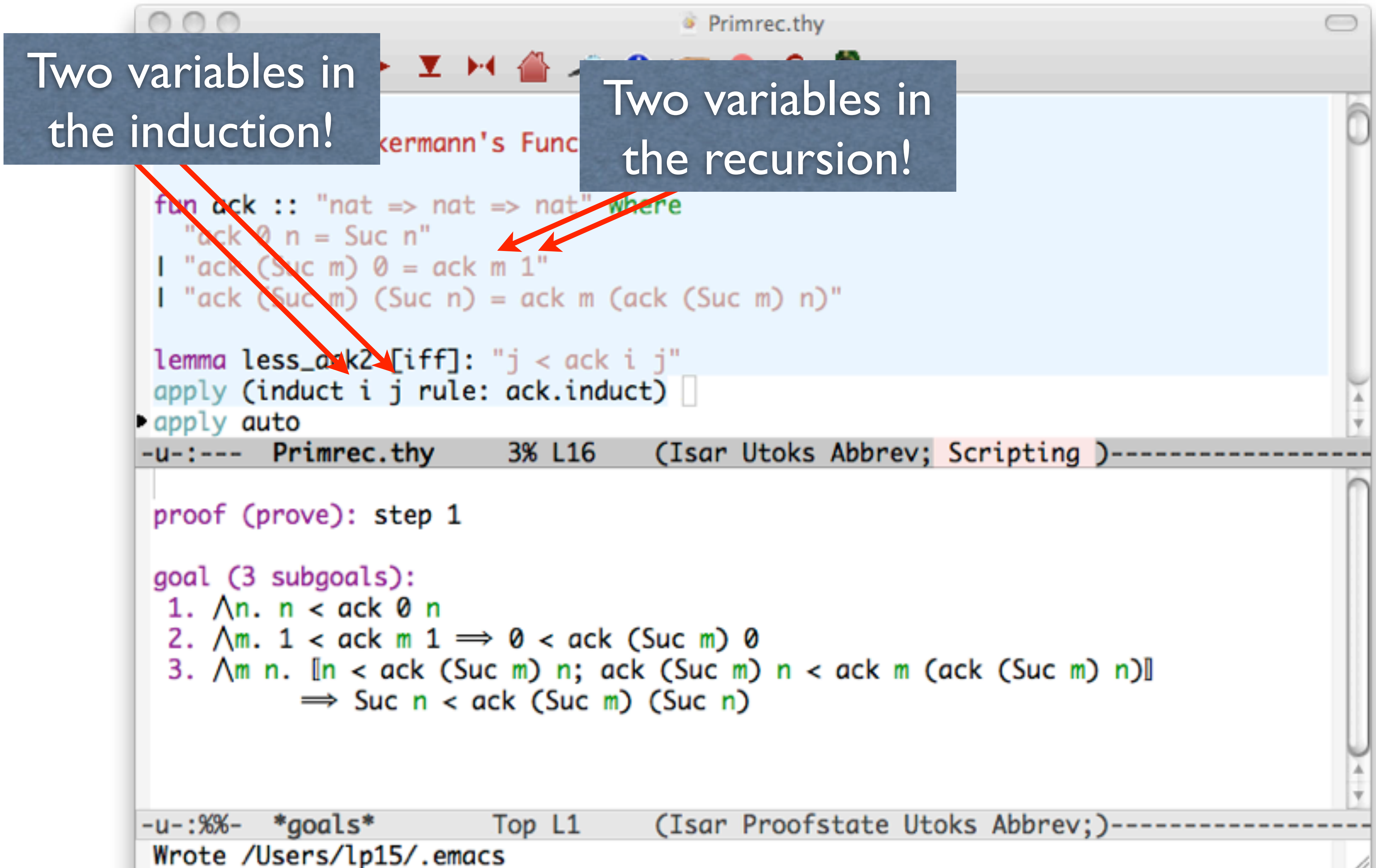
Unusual Recursions



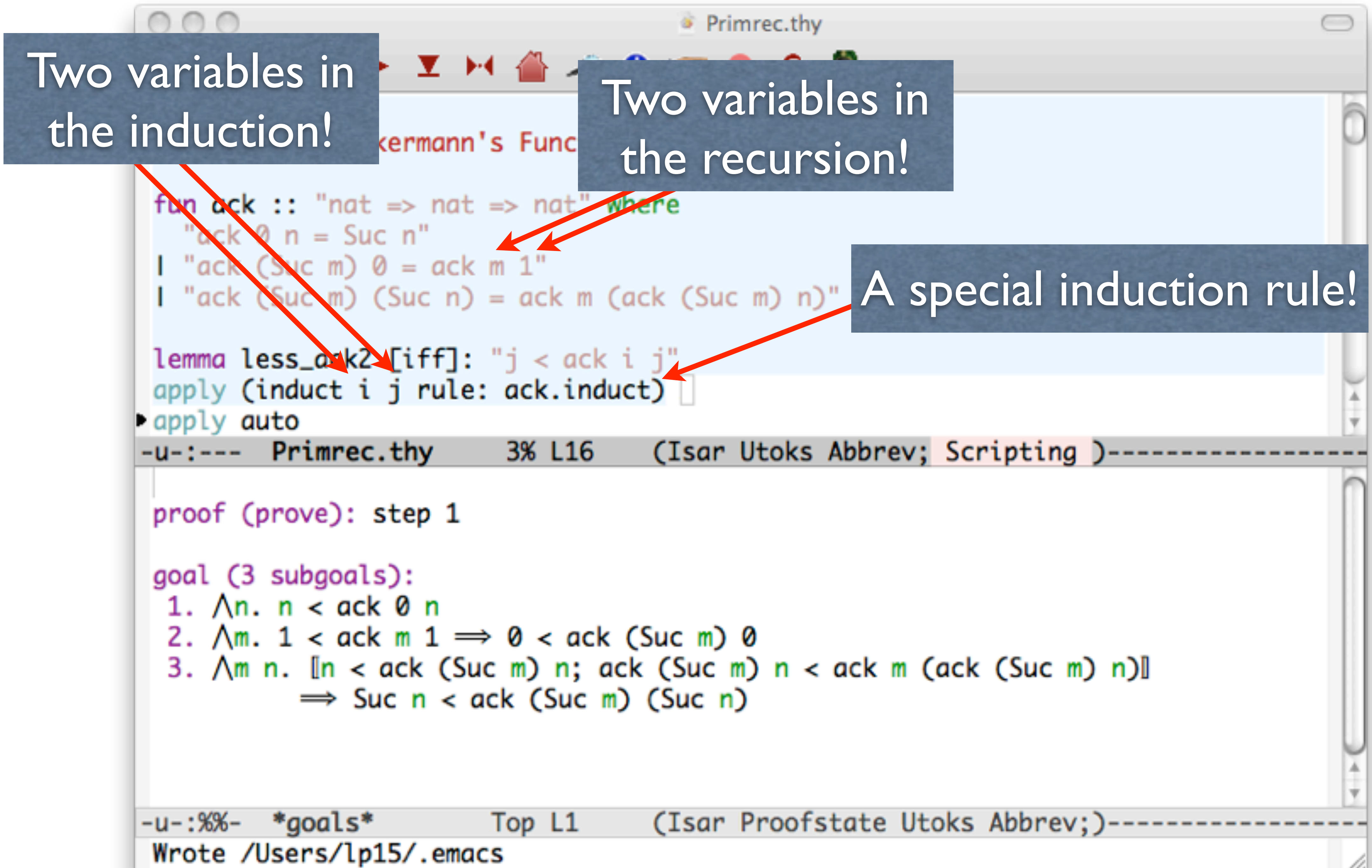
Unusual Recursions



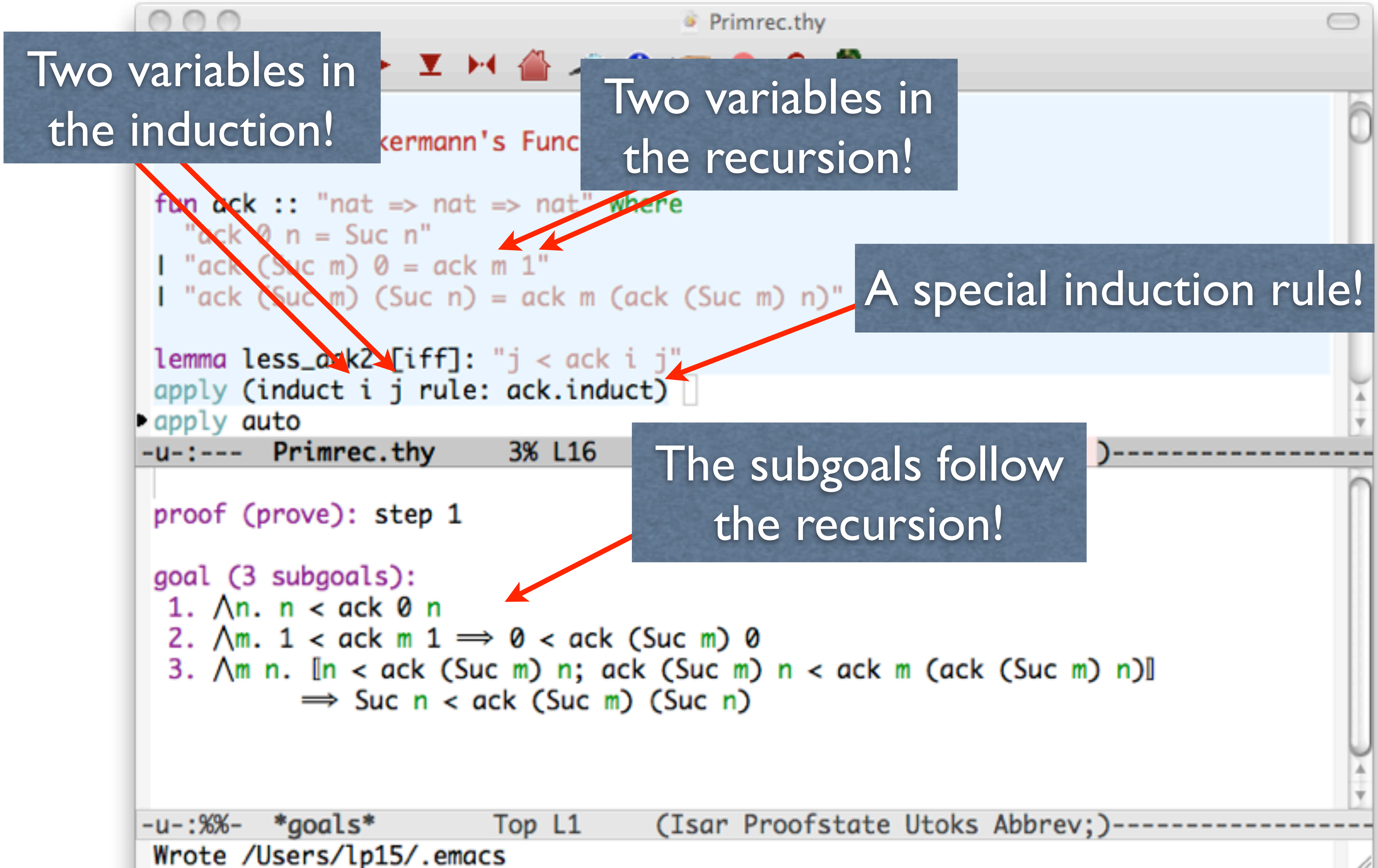
Unusual Recursions



Unusual Recursions



Unusual Recursions



Recursion: Key Points

Recursion: Key Points

- Recursion in one variable, following the structure of a datatype declaration, is called *primitive*.

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**.

Special Induction Rules

Special Induction Rules

- They follow the function's recursion exactly.

Special Induction Rules

- They follow the function's recursion exactly.
- For Ackermann, they reduce $P\ x\ y$ to
  - $P\ 0\ n$, for arbitrary n
  - $P\ (Suc\ m)\ 0$ assuming $P\ m\ 1$, for arbitrary m
  - $P\ (Suc\ m)\ (Suc\ n)$ assuming $P\ (Suc\ m)\ n$ and $P\ m\ (ack\ (Suc\ m)\ n)$, for arbitrary m and n

Special Induction Rules

- They follow the function's recursion exactly.
- For Ackermann, they reduce $P\ x\ y$ to
  - $P\ 0\ n$, for arbitrary n
  - $P\ (Suc\ m)\ 0$ assuming $P\ m\ 1$, for arbitrary m
  - $P\ (Suc\ m)\ (Suc\ n)$ assuming $P\ (Suc\ m)\ n$ and $P\ m\ (ack\ (Suc\ m)\ n)$, for arbitrary m and n
- **Usually** they do what you want. Trial and error is tempting, but ultimately you will need to think!

Another Unusual Recursion

MergeSort.thy

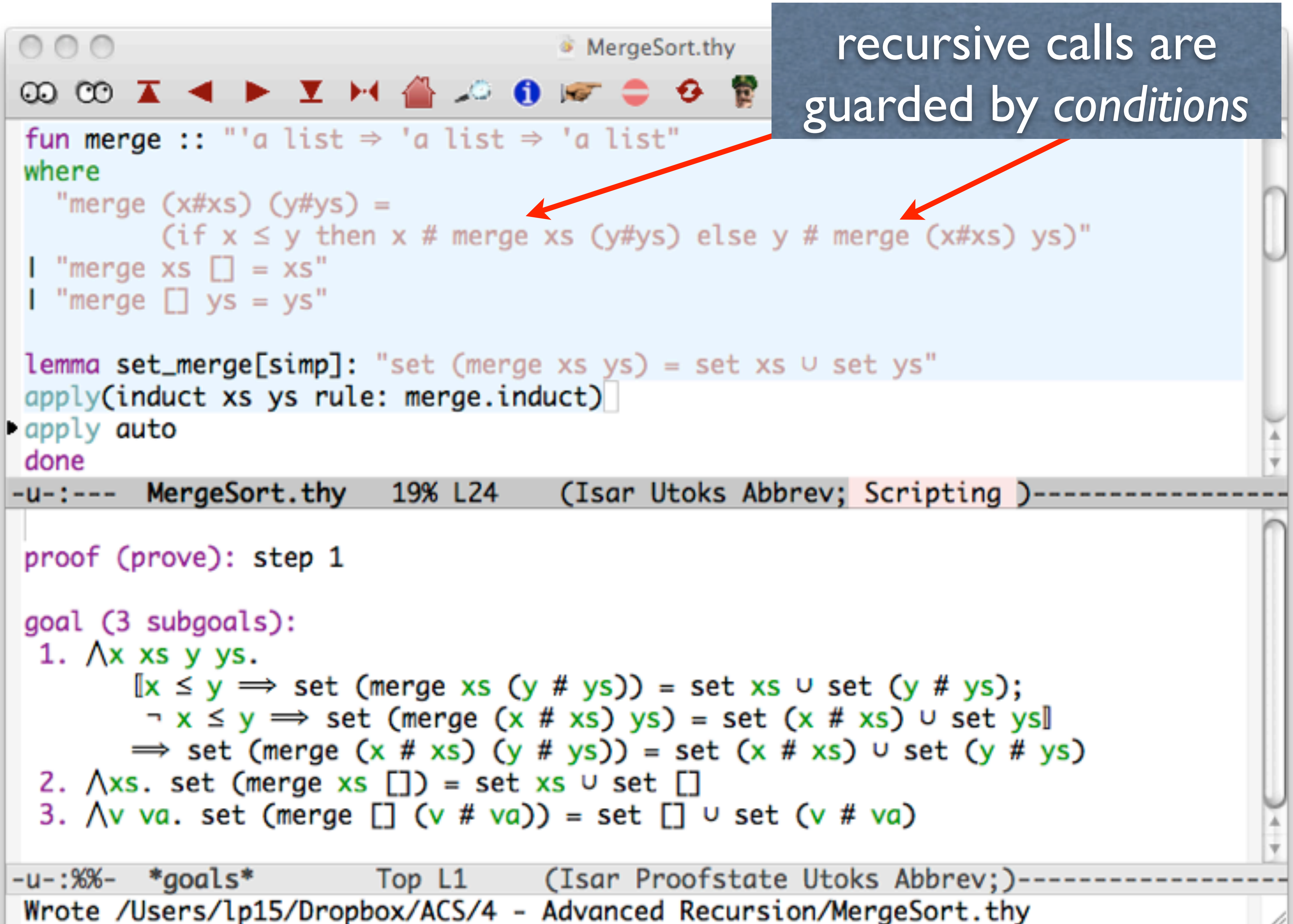
```
fun merge :: "'a list ⇒ 'a list ⇒ 'a list"
where
  "merge (x#xs) (y#ys) =
        (if x ≤ y then x # merge xs (y#ys) else y # merge (x#xs) ys)"
| "merge xs [] = xs"
| "merge [] ys = ys"

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% L24   (Isar Utoks Abbrev; Scripting )----------------
```

```
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
```

Another Unusual Recursion



Another Unusual Recursion

MergeSort.thy

```
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
```

recursive calls are guarded by *conditions*

```
-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)
```

2 induction hypotheses, guarded by conditions!

```
-u-:%%-  *goals*        Top L1      (Isar Proofstate Utoks Abbrev;)----------------
Wrote /Users/lp15/Dropbox/ACS/4 - Advanced Recursion/MergeSort.thy
```

Proof Outline

```
set (merge (x#xs) (y#ys)) = set (x # xs) ∪ set (y # ys)

          set (if x ≤ y then x # merge xs (y#ys)
               else y # merge (x#xs) ys)          =   ...
                              =
        (x ≤ y → set(x # merge xs (y#ys)) = ...) &
        (¬ x ≤ y → set(y # merge (x#xs) ys) = ...)
                              =
       (x ≤ y → {x} ∪ set(merge xs (y#ys)) = ...) &
       (¬ x ≤ y → {y} ∪ set(merge (x#xs) ys) = ...)
                              =
      (x ≤ y → {x} ∪ set xs ∪ set (y # ys) = ...) &
      (¬ x ≤ y → {y} ∪ set (x # xs) ∪ set ys = ...)
```

Proof Outline

```
set (merge (x#xs) (y#ys)) = set (x # xs) ∪ set (y # ys)

    set (if x ≤ y then x # merge xs (y#ys)
          else y # merge (x#xs) ys)           =    ...
                         =
   (x ≤ y → set(x # merge xs (y#ys)) = ...) &
   (¬ x ≤ y → set(y # merge (x#xs) ys) = ...)
                         =
  (x ≤ y → {x} ∪ set(merge xs (y#ys)) = ...) &
  (¬ x ≤ y → {y} ∪ set(merge (x#xs) ys) = ...)
                         =
 (x ≤ y → {x} ∪ set xs ∪ set (y # ys) = ...) &
 (¬ x ≤ y → {y} ∪ set (x # xs) ∪ set ys = ...)
```

Proof Outline

```
set (merge (x#xs) (y#ys)) = set (x # xs) ∪ set (y # ys)

        set (if x ≤ y then x # merge xs (y#ys)
             else y # merge (x#xs) ys)              =    ...
                              =
      (x ≤ y → set(x # merge xs (y#ys)) = ...) &
      (¬ x ≤ y → set(y # merge (x#xs) ys) = ...)
                              =
     (x ≤ y → {x} ∪ set(merge xs (y#ys)) = ...) &
     (¬ x ≤ y → {y} ∪ set(merge (x#xs) ys) = ...)
                              =
    (x ≤ y → {x} ∪ set xs ∪ set (y # ys) = ...) &
    (¬ x ≤ y → {y} ∪ set (x # xs) ∪ set ys = ...)
```

Proof Outline

```
set (merge (x#xs) (y#ys)) = set (x # xs) ∪ set (y # ys)

        set (if x ≤ y then x # merge xs (y#ys)
             else y # merge (x#xs) ys)              =    ...
                              =
      (x ≤ y → set(x # merge xs (y#ys)) = ...) &
      (¬ x ≤ y → set(y # merge (x#xs) ys) = ...)
                              =
     (x ≤ y → {x} ∪ set(merge xs (y#ys)) = ...) &
     (¬ x ≤ y → {y} ∪ set(merge (x#xs) ys) = ...)
                              =
    (x ≤ y → {x} ∪ set xs ∪ set (y # ys) = ...) &
    (¬ x ≤ y → {y} ∪ set (x # xs) ∪ set ys = ...)
```

Proof Outline

```
set (merge (x#xs) (y#ys)) = set (x # xs) ∪ set (y # ys)

        set (if x ≤ y then x # merge xs (y#ys)
             else y # merge (x#xs) ys)              =    ...
                              =
     (x ≤ y → set(x # merge xs (y#ys)) = ...) &
     (¬ x ≤ y → set(y # merge (x#xs) ys) = ...)
                              =
    (x ≤ y → {x} ∪ set(merge xs (y#ys)) = ...) &
    (¬ x ≤ y → {y} ∪ set(merge (x#xs) ys) = ...)
                              =
   (x ≤ y → {x} ∪ set xs ∪ set (y # ys) = ...) &
   (¬ x ≤ y → {y} ∪ set (x # xs) ∪ set ys = ...)
```

The Case Expression

The Case Expression

- Similar to that found in the functional language ML.

The Case Expression

- Similar to that found in the functional language ML.
- Automatically generated for every datatype.

The Case Expression

- Similar to that found in the functional language ML.
- Automatically generated for every datatype.
- The simplifier can (upon request!) perform case-splits analogous to those for "`if`".

The Case Expression

- Similar to that found in the functional language ML.
- Automatically generated for every datatype.
- The simplifier can (upon request!) perform case-splits analogous to those for "`if`".
- Case splits in *assumptions* (not the conclusion) never happen unless requested.

Case-Splits for Lists

Case-Splits for Lists

```
fun ordered :: "'a list => bool"
where
  "ordered [] = True"
| "ordered [x] = True"
| "ordered (x#y#xs) = (x≤y & ordered (y#xs))"
```

Case-Splits for Lists

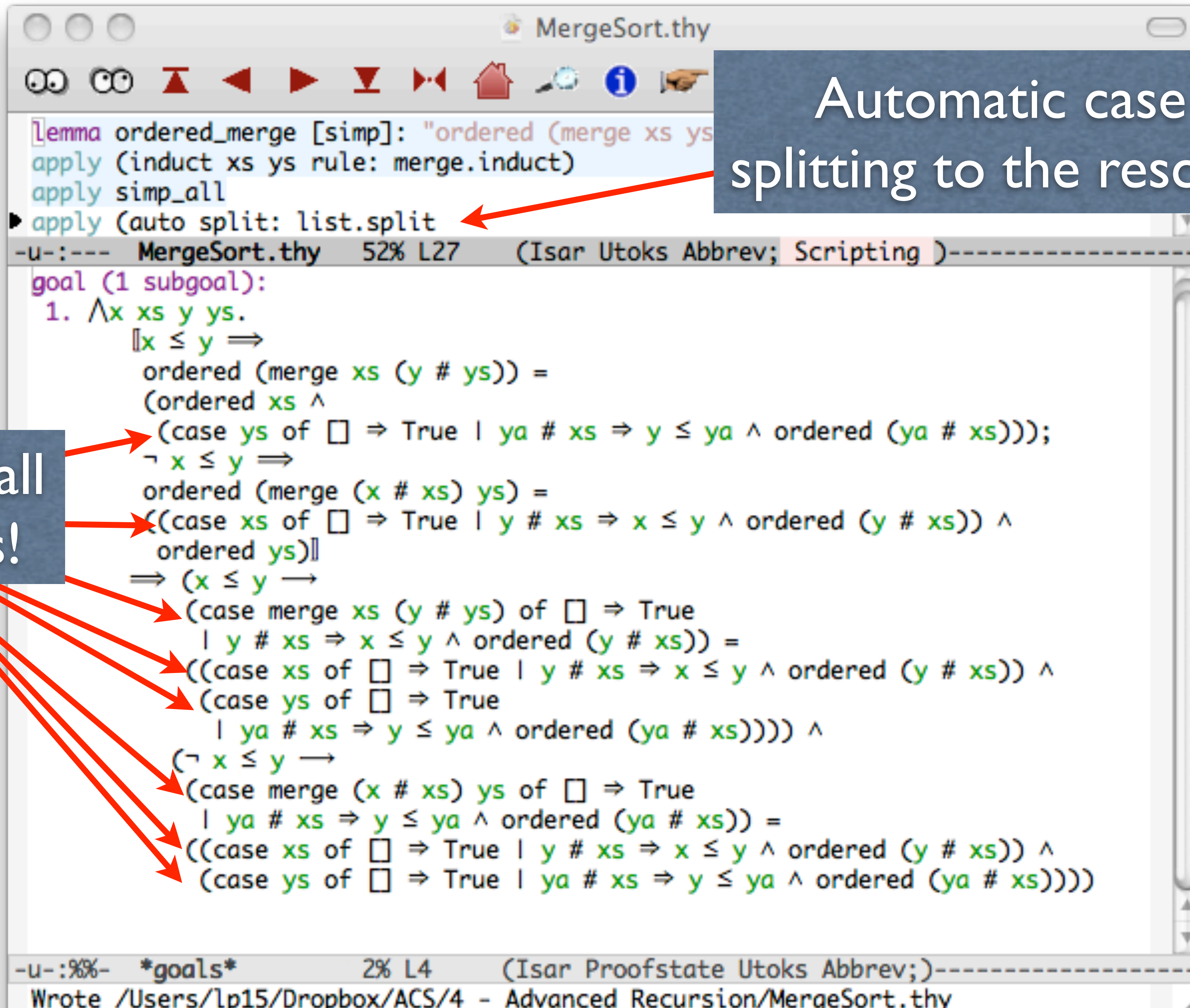
```
fun ordered :: "'a list => bool"
where
  "ordered [] = True"
| "ordered (x#l) =
     (case l of [] => True
        | Cons y xs => (x≤y & ordered y#xs)))"
```

Case-Splitting in Action

```
MergeSort.thy

lemma ordered_merge [simp]: "ordered (merge xs ys) = (ordered xs & ordered ys)"
apply (induct xs ys rule: merge.induct)
apply simp_all
apply (auto split: list.split
-u-:---  MergeSort.thy    52% L27    (Isar Utoks Abbrev; Scripting )----------------
goal (1 subgoal):
 1. ⋀x xs y ys.
       ⟦x ≤ y ⟹
        ordered (merge xs (y # ys)) =
        (ordered xs ∧
         (case ys of [] ⇒ True | ya # xs ⇒ y ≤ ya ∧ ordered (ya # xs)));
        ¬ x ≤ y ⟹
        ordered (merge (x # xs) ys) =
        ((case xs of [] ⇒ True | y # xs ⇒ x ≤ y ∧ ordered (y # xs)) ∧
         ordered ys)⟧
       ⟹ (x ≤ y ⟶
            (case merge xs (y # ys) of [] ⇒ True
             | y # xs ⇒ x ≤ y ∧ ordered (y # xs)) =
            ((case xs of [] ⇒ True | y # xs ⇒ x ≤ y ∧ ordered (y # xs)) ∧
             (case ys of [] ⇒ True
              | ya # xs ⇒ y ≤ ya ∧ ordered (ya # xs)))) ∧
           (¬ x ≤ y ⟶
            (case merge (x # xs) ys of [] ⇒ True
             | ya # xs ⇒ y ≤ ya ∧ ordered (ya # xs)) =
            ((case xs of [] ⇒ True | y # xs ⇒ x ≤ y ∧ ordered (y # xs)) ∧
             (case ys of [] ⇒ True | ya # xs ⇒ y ≤ ya ∧ ordered (ya # xs))))
-u-:%%-  *goals*     2% L4    (Isar Proofstate Utoks Abbrev;)------------------
Wrote /Users/lp15/Dropbox/ACS/4 - Advanced Recursion/MergeSort.thy
```

Help! Look at all the case-splits!

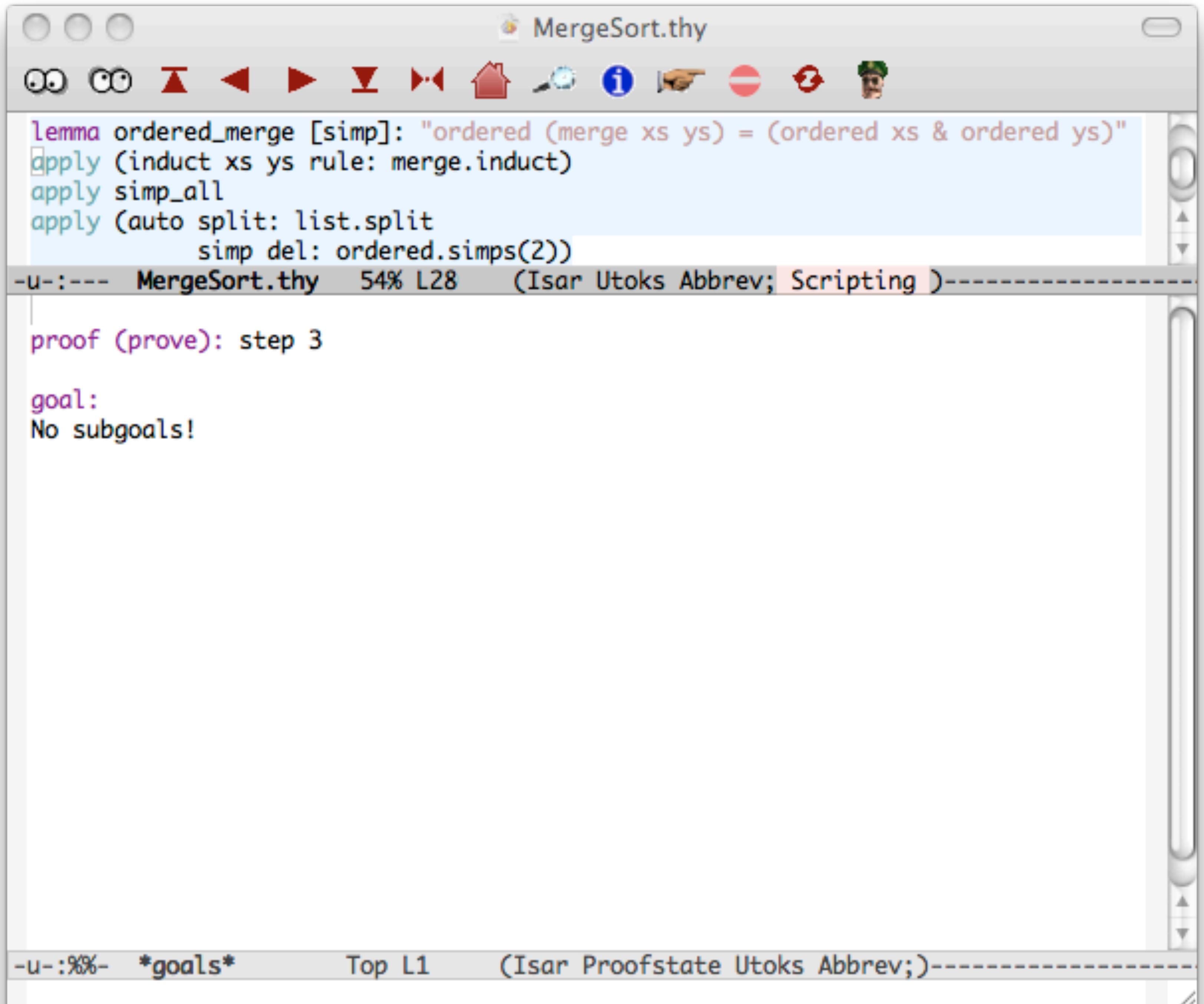
Case-Splitting in Action



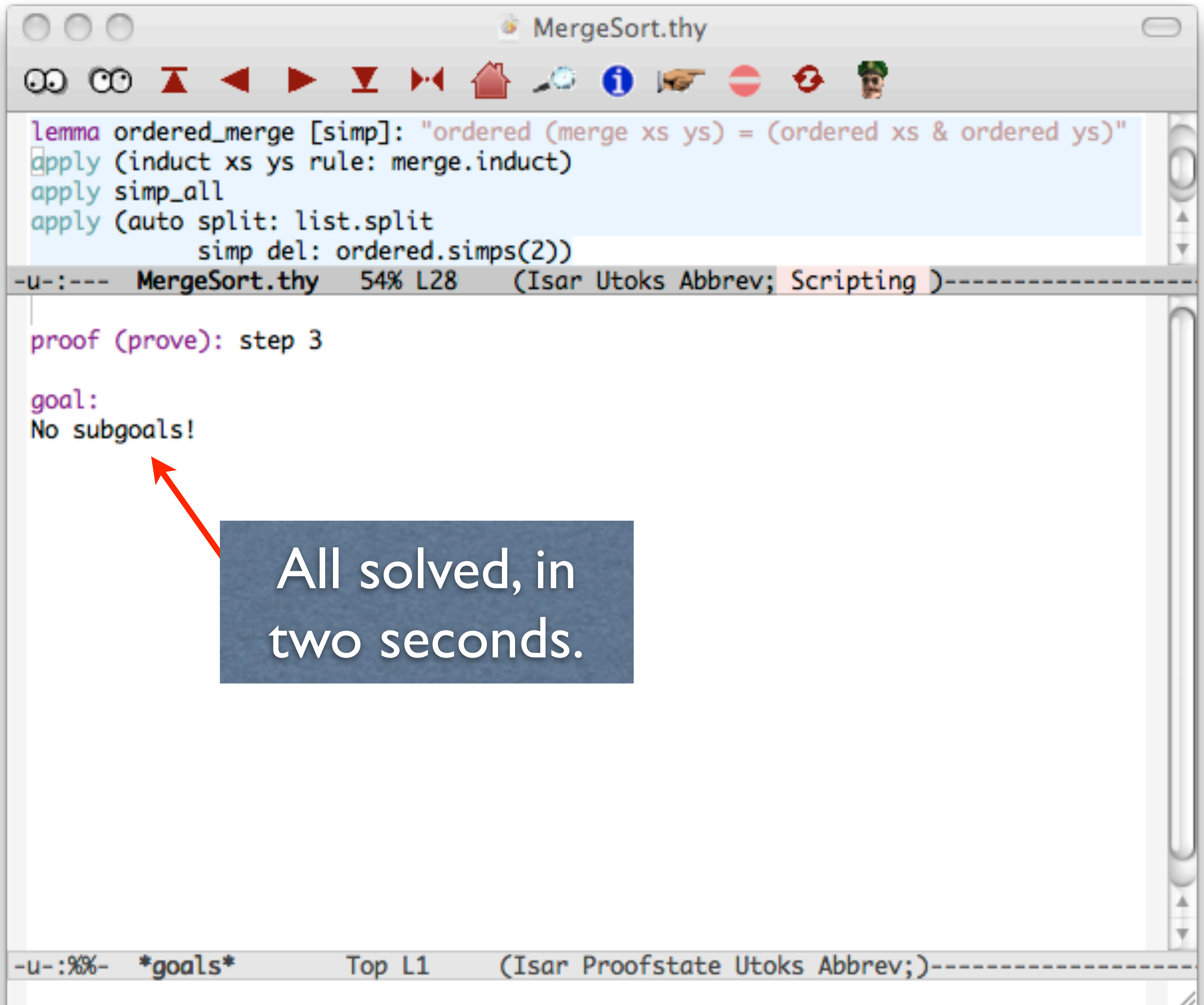
Automatic case splitting to the rescue!

Help! Look at all the case-splits!

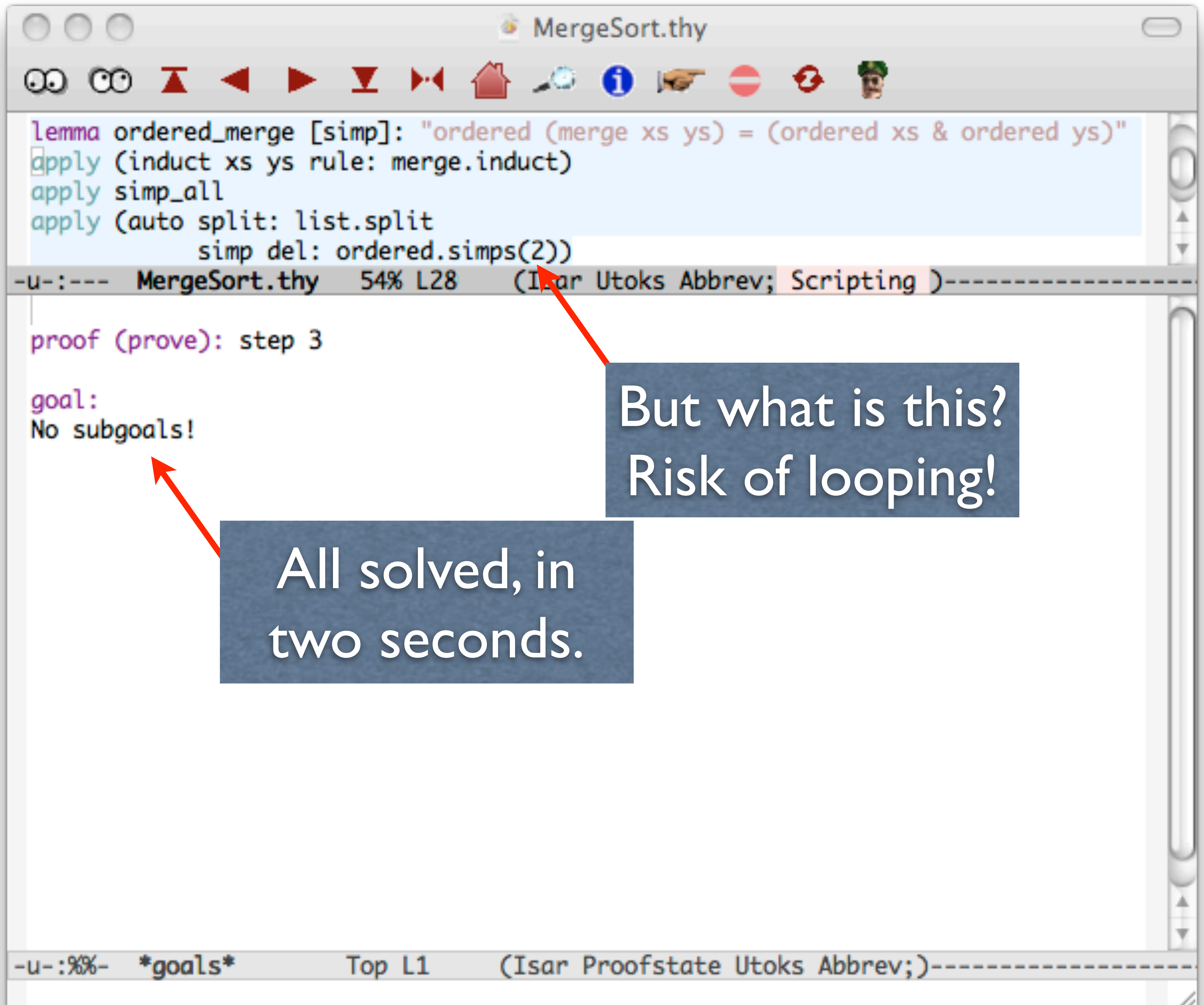
Completing the Proof



Completing the Proof



Completing the Proof



Case Splitting for Lists

Simplification will replace

$$P\,(\texttt{case}\ xs\ \texttt{of}\ \texttt{[]}\ \texttt{=>}\ a \mid \texttt{Cons}\ h\ tl\ \texttt{=>}\ b\ h\ tl)$$

by

$$(xs = \texttt{[]} \rightarrow P\ a) \wedge (\forall h\ tl.\ xs = h \# tl \rightarrow P\ (b\ h\ tl))$$

Case Splitting for Lists

Simplification will replace

$$P\,(\texttt{case}\ xs\ \texttt{of}\ \texttt{[]}\ \texttt{=>}\ a \mid \texttt{Cons}\ h\ tl\ \texttt{=>}\ b\ h\ tl)$$

by

$$(xs = \texttt{[]} \rightarrow P\ a) \wedge (\forall h\ tl.\ xs = h \# tl \rightarrow P\ (b\ h\ tl))$$

- It creates a case for each datatype constructor.

Case Splitting for Lists

Simplification will replace

$$P\ (\texttt{case}\ xs\ \texttt{of}\ \texttt{[]}\ \texttt{=>}\ a\ |\ \texttt{Cons}\ h\ tl\ \texttt{=>}\ b\ h\ tl)$$

by

$$(xs = \texttt{[]} \to P\ a) \wedge (\forall h\ tl.\ xs = h \# tl \to P\ (b\ h\ tl))$$

- It creates a case for each datatype constructor.
- Here it causes looping if combined with the second rewrite rule for `ordered`.

Summary

Summary

- Many forms of recursion are available.

Summary

- Many forms of recursion are available.
- The supplied induction rule often leads to simple proofs.

Summary

- Many forms of recursion are available.
- The supplied induction rule often leads to simple proofs.
- The “case” operator can often be dealt with using automatic case splitting...

Summary

- Many forms of recursion are available.
- The supplied induction rule often leads to simple proofs.
- The "case" operator can often be dealt with using automatic case splitting...
- but complex simplifications can run forever!

A Helpful Tip