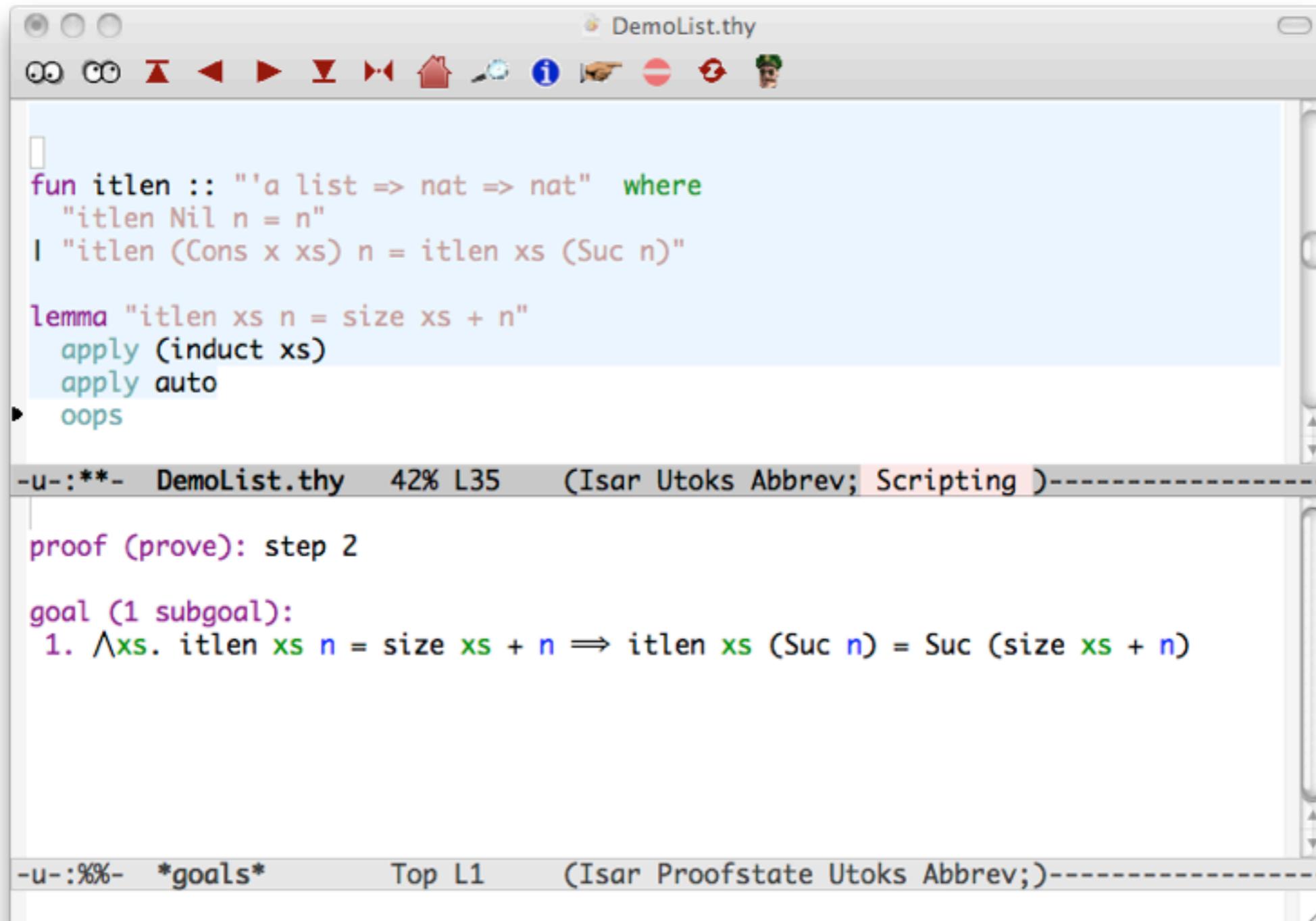


Interactive Formal Verification

4: Advanced Recursion, Induction and Simplification

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

A Failing Proof by Induction



The screenshot shows a window titled "DemoList.thy" with a toolbar at the top. The main text area contains the following code:

```
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
```

Below the code is a status bar: `-u-:**- DemoList.thy 42% L35 (Isar Utoks Abbrev; Scripting)-----`

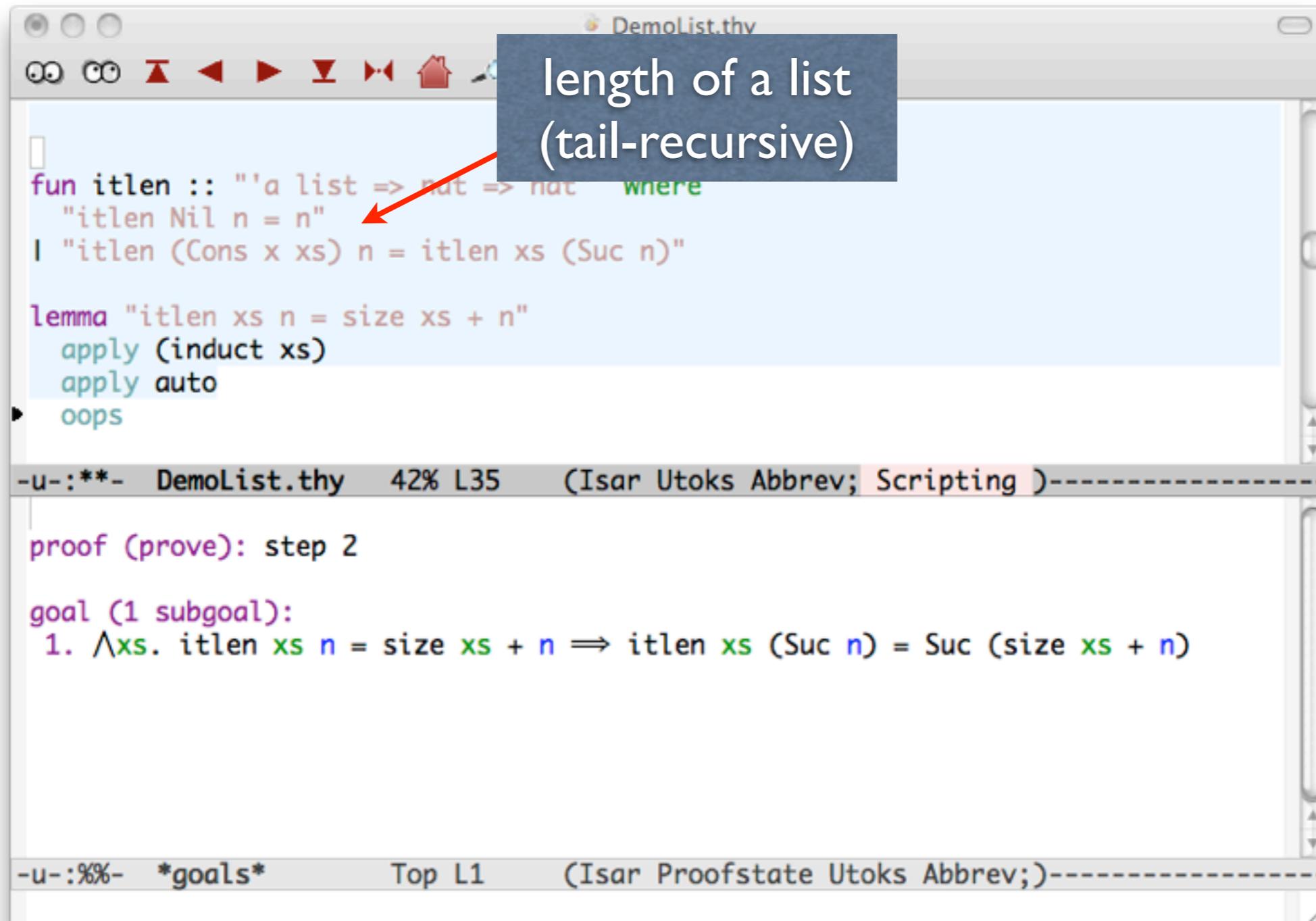
The next section shows the proof state:

```
proof (prove): step 2

goal (1 subgoal):
  1.  $\wedge xs. \text{itlen } xs \ n = \text{size } xs + n \implies \text{itlen } xs \ (\text{Suc } n) = \text{Suc } (\text{size } xs + n)$ 
```

At the bottom, another status bar reads: `-u-:%%- *goals* Top L1 (Isar Proofstate Utoks Abbrev;)-----`

A Failing Proof by Induction



```
Demolist.thy
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.  $\wedge xs. \text{itlen } xs \ n = \text{size } xs + n \implies \text{itlen } xs \ (\text{Suc } n) = \text{Suc } (\text{size } xs + n)$ 

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

A Failing Proof by Induction

```
Demolist.thy
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.  $\wedge xs. \text{itlen } xs \ n = \text{size } xs + n \implies \text{itlen } xs \ (\text{Suc } n) = \text{Suc } (\text{size } xs + n)$ 

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

A Failing Proof by Induction

```
Demolist.thy
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.  $\wedge xs. \text{itlen } xs \ n = \text{size } xs + n \implies \text{itlen } xs \ (\text{Suc } n) = \text{Suc } (\text{size } xs + n)$ 

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

length of a list
(tail-recursive)

equivalent to the built-in length function?

Mismatch between induction hypothesis and conclusion!

A Failing Proof by Induction

```
Demolist.thy
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.  $\wedge xs. \text{itlen } xs \ n = \text{size } xs + n \implies \text{itlen } xs \ (\text{Suc } n) = \text{Suc } (\text{size } xs + n)$ 
-u-:%%- *goals* Top L1 (Isar Proofstate Utoks Abbrev;)
```

length of a list
(tail-recursive)

equivalent to the built-
in length function?

May as well
give up!

Mismatch between induction
hypothesis and conclusion!

Generalising the Induction

The screenshot shows a theorem prover interface with a code editor and a proof state window. The code editor contains the following text:

```
fun itlen :: "'a list => nat"
  "itlen Nil n = 0"
| "itlen (Cons x xs) n = itlen xs (Suc n)"

lemma "∀n. itlen xs n = size xs + n"
  apply (induct xs)
  apply auto
  done
```

A blue callout box with the text "Insert a universal quantifier" has a red arrow pointing to the `forall` quantifier in the lemma statement.

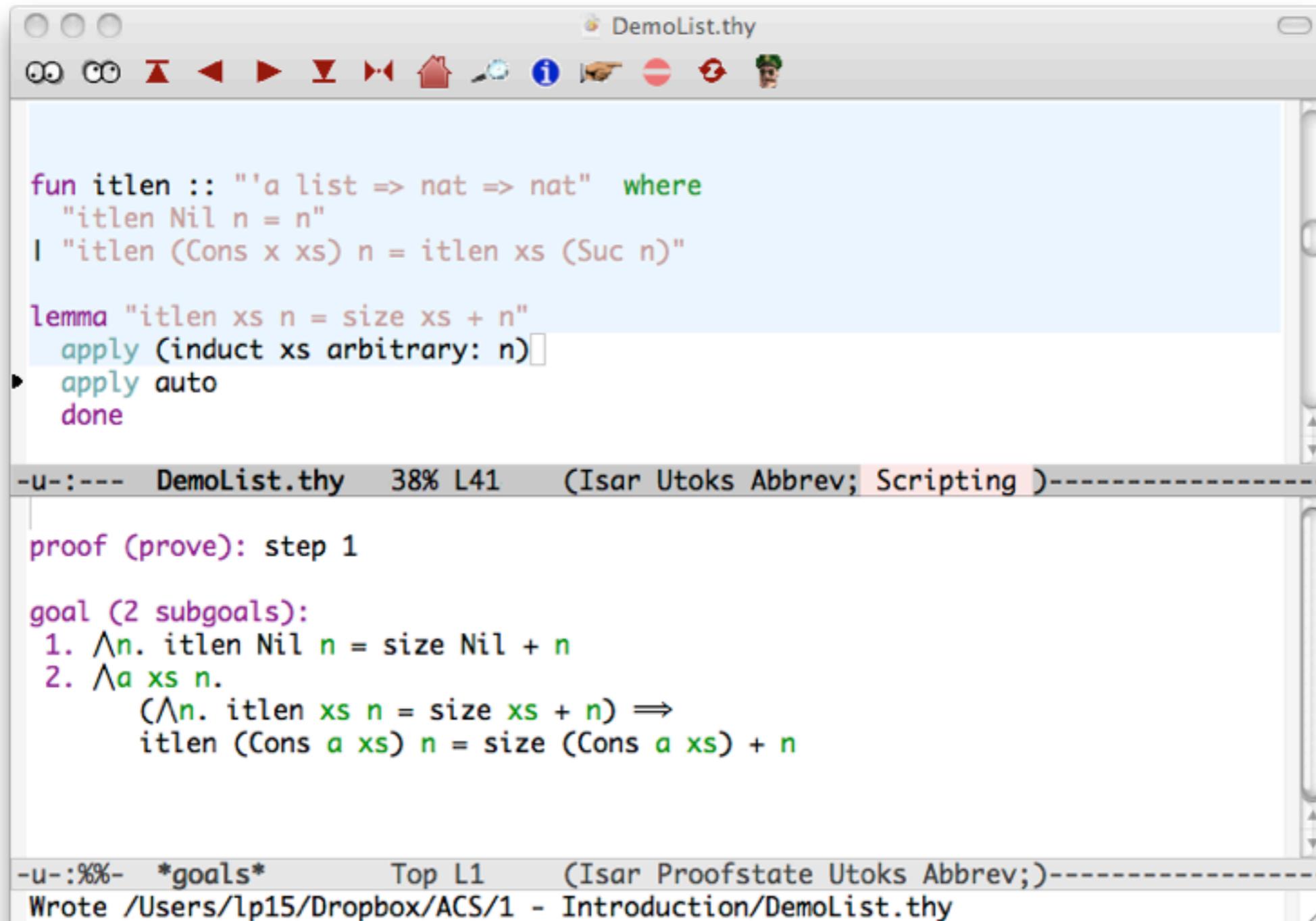
The proof state window shows the following text:

```
proof (prove): step 1
goal (2 subgoals):
1. ∀n. itlen Nil n = size Nil + n
2. ∀a xs.
   ∀n. itlen xs n = size xs + n ⇒
   ∀n. itlen (Cons a xs) n = size (Cons a xs) + n
```

A blue callout box with the text "Induction hypothesis holds for all n" has a red arrow pointing to the `forall` quantifier in the second subgoal.

The status bar at the bottom of the proof state window reads: `-u-:%%- *goals* Top L1 (Isar Proofstate Utoks Abbrev;)----- (No changes need to be saved)`

Generalising: Another Way



```
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 arbitrary: n)
  apply auto
  done

proof (prove): step 1

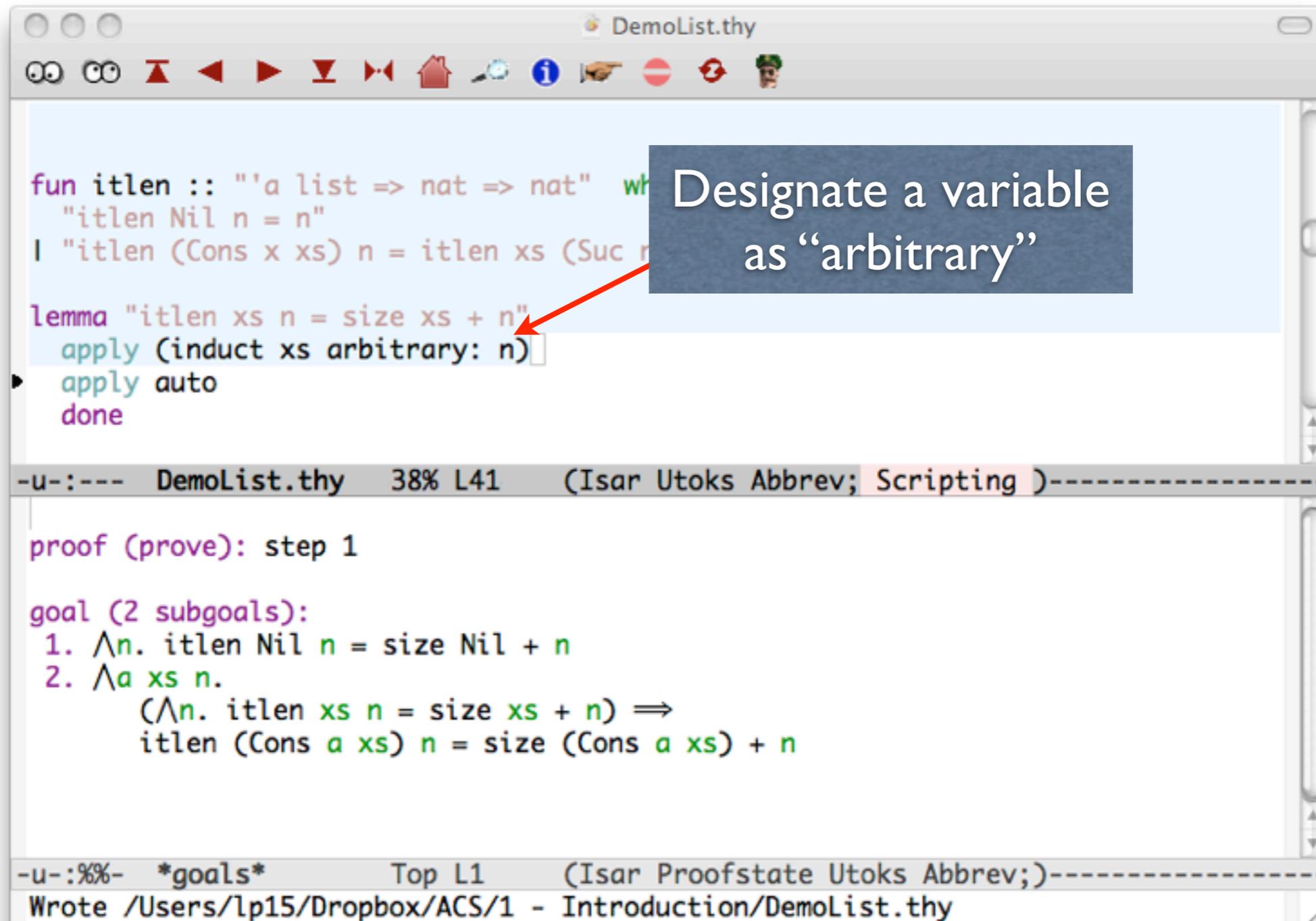
goal (2 subgoals):
1.  $\forall n. \text{itlen Nil } n = \text{size Nil} + n$ 
2.  $\forall a \text{ xs } n. (\forall n. \text{itlen xs } n = \text{size xs} + n) \Rightarrow \text{itlen (Cons a xs) } n = \text{size (Cons a xs)} + n$ 
```

-u-:--- DemoList.thy 38% L41 (Isar Utoks Abbrev; Scripting)-----

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

Wrote /Users/lp15/Dropbox/ACS/1 - Introduction/DemoList.thy

Generalising: Another Way



The screenshot shows a window titled "DemoList.thy" with a toolbar at the top. The main area contains the following code:

```
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 arbitrary: n)
  apply auto
  done
```

A blue callout box with the text "Designate a variable as 'arbitrary'" has a red arrow pointing to the `arbitrary: n` part of the `induct` command.

The bottom part of the window shows the proof state:

```
proof (prove): step 1

goal (2 subgoals):
  1.  $\forall n. \text{itlen Nil } n = \text{size Nil} + n$ 
  2.  $\forall a \text{ xs } n. (\forall n. \text{itlen xs } n = \text{size xs} + n) \Rightarrow \text{itlen (Cons a xs) } n = \text{size (Cons a xs)} + n$ 
```

The status bar at the bottom indicates the current position: `-u-:%%- *goals* Top L1 (Isar Proofstate Utoks Abbrev;)-----` and the file path: `Wrote /Users/lp15/Dropbox/ACS/1 - Introduction/DemoList.thy`

Generalising: Another Way

```
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 arbitrary: n)
  apply auto
  done

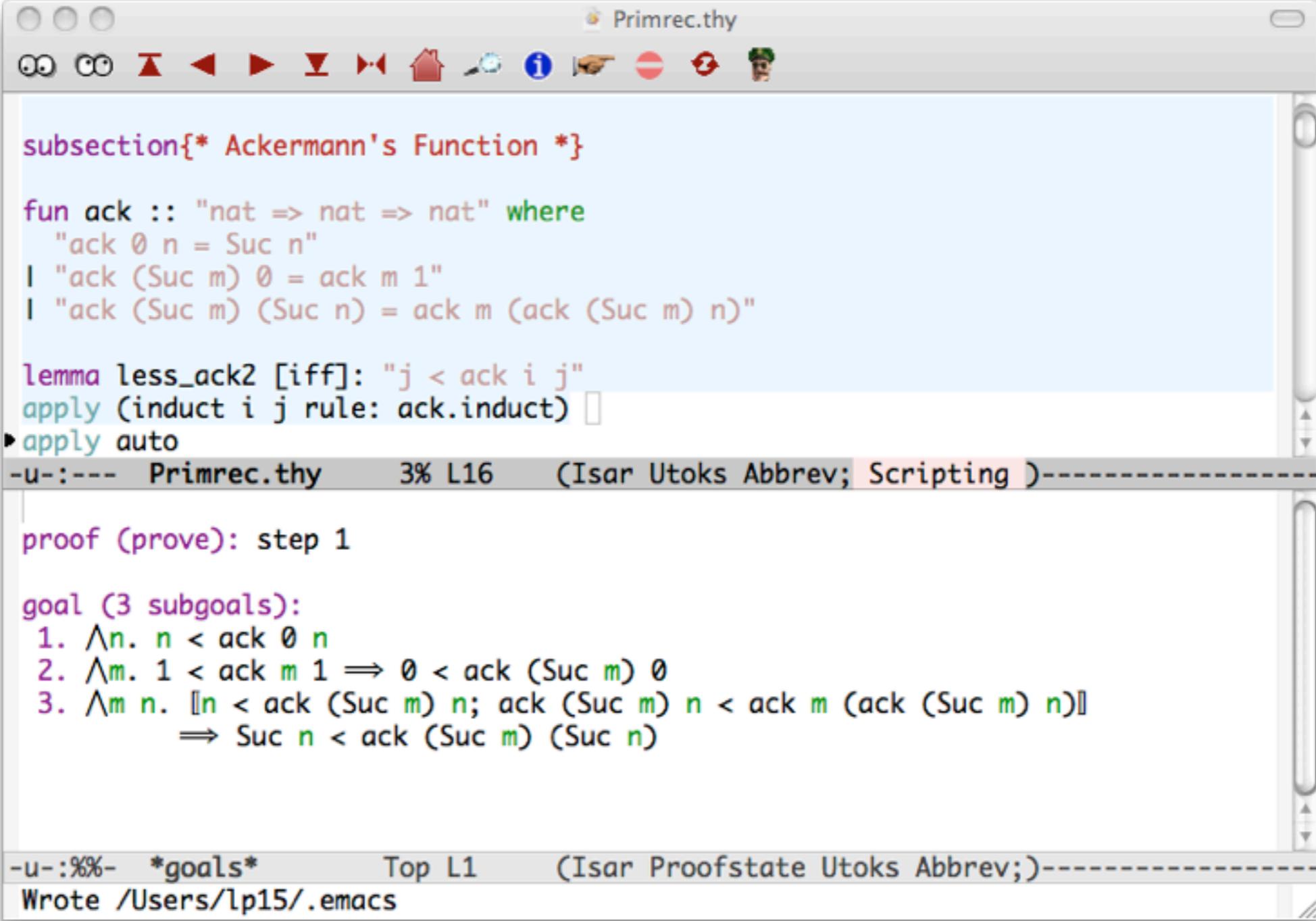
proof (prove): step 1
goal (2 subgoals):
1.  $\forall n. \text{itlen Nil } n = \text{size Nil} + n$ 
2.  $\forall a \text{ xs } n. (\forall n. \text{itlen xs } n = \text{size xs} + n) \Rightarrow \text{itlen (Cons a xs) } n = \text{size (Cons a xs)} + n$ 
```

-u-:--- DemoList.thy 38% L41 (Isar Utoks Abbrev; Scripting)-----

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

Wrote /Users/lp15/Dropbox/ACS/1 - Introduction/DemoList.thy

Unusual Recursions



```
Primrec.thy
subsubsection{* 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.  $\wedge n. n < \text{ack } 0 \ n$ 
2.  $\wedge m. 1 < \text{ack } m \ 1 \implies 0 < \text{ack } (\text{Suc } m) \ 0$ 
3.  $\wedge m \ n. [n < \text{ack } (\text{Suc } m) \ n; \text{ack } (\text{Suc } m) \ n < \text{ack } m \ (\text{ack } (\text{Suc } m) \ n)] \implies \text{Suc } n < \text{ack } (\text{Suc } m) \ (\text{Suc } n)$ 
-u-:%%- *goals*          Top L1      (Isar Proofstate Utoks Abbrev;)-----
Wrote /Users/lp15/.emacs
```

Unusual Recursions

```
Primrec.thy
subsection{* Ackermann'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 (Isar Utoks Abbrev; Scripting )-----

proof (prove): step 1

goal (3 subgoals):
1.  $\wedge n. n < \text{ack } 0 \ n$ 
2.  $\wedge m. 1 < \text{ack } m \ 1 \implies 0 < \text{ack } (\text{Suc } m) \ 0$ 
3.  $\wedge m \ n. [n < \text{ack } (\text{Suc } m) \ n; \text{ack } (\text{Suc } m) \ n < \text{ack } m \ (\text{ack } (\text{Suc } m) \ n)] \implies \text{Suc } n < \text{ack } (\text{Suc } m) \ (\text{Suc } n)$ 
- u-:%%- *goals* Top L1 (Isar Proofstate Utoks Abbrev;)-----
Wrote /Users/lp15/.emacs
```

Two variables in the recursion!

Unusual Recursions

Two variables in the induction!

Two variables in the recursion!

```
Primrec.thy
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 (Isar Utoks Abbrev; Scripting )-----

proof (prove): step 1

goal (3 subgoals):
1.  $\wedge n. n < \text{ack } 0 \ n$ 
2.  $\wedge m. 1 < \text{ack } m \ 1 \implies 0 < \text{ack } (\text{Suc } m) \ 0$ 
3.  $\wedge m \ n. [n < \text{ack } (\text{Suc } m) \ n; \text{ack } (\text{Suc } m) \ n < \text{ack } m \ (\text{ack } (\text{Suc } m) \ n)] \implies \text{Suc } n < \text{ack } (\text{Suc } m) \ (\text{Suc } n)$ 

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

Unusual Recursions

Two variables in the induction!

Two variables in the recursion!

A special induction rule!

```
Primrec.thy
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 (Isar Utoks Abbrev; Scripting )-----

proof (prove): step 1

goal (3 subgoals):
1.  $\wedge n. n < \text{ack } 0 \ n$ 
2.  $\wedge m. 1 < \text{ack } m \ 1 \implies 0 < \text{ack } (\text{Suc } m) \ 0$ 
3.  $\wedge m \ n. [n < \text{ack } (\text{Suc } m) \ n; \text{ack } (\text{Suc } m) \ n < \text{ack } m \ (\text{ack } (\text{Suc } m) \ n)] \implies \text{Suc } n < \text{ack } (\text{Suc } m) \ (\text{Suc } n)$ 
- u-:%%- *goals* Top L1 (Isar Proofstate Utoks Abbrev;)-----
Wrote /Users/lp15/.emacs
```

Unusual Recursions

Two variables in the induction!

Two variables in the recursion!

A special induction rule!

The subgoals follow the recursion!

```
Primrec.thy
kermann's Func
where
fun ack :: "nat => nat => nat"
  "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.  $\wedge n. n < \text{ack } 0 \ n$ 
2.  $\wedge m. 1 < \text{ack } m \ 1 \implies 0 < \text{ack } (\text{Suc } m) \ 0$ 
3.  $\wedge m \ n. [n < \text{ack } (\text{Suc } m) \ n; \text{ack } (\text{Suc } m) \ n < \text{ack } m \ (\text{ack } (\text{Suc } m) \ n)] \implies \text{Suc } n < \text{ack } (\text{Suc } m) \ (\text{Suc } n)$ 
- u-:%%- *goals* Top L1 (Isar Proofstate Utoks Abbrev;)-----
Wrote /Users/lp15/.emacs
```

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.  $\forall x \ xs \ y \ ys.$ 
    $[x \leq y \implies \text{set } (\text{merge } xs \ (y \ # \ ys)) = \text{set } xs \ \cup \ \text{set } (y \ # \ ys);$ 
    $\neg x \leq y \implies \text{set } (\text{merge } (x \ # \ xs) \ ys) = \text{set } (x \ # \ xs) \ \cup \ \text{set } ys]$ 
    $\implies \text{set } (\text{merge } (x \ # \ xs) \ (y \ # \ ys)) = \text{set } (x \ # \ xs) \ \cup \ \text{set } (y \ # \ ys)$ 
2.  $\forall xs. \text{set } (\text{merge } xs \ []) = \text{set } xs \ \cup \ \text{set } []$ 
3.  $\forall v \ va. \text{set } (\text{merge } [] \ (v \ # \ va)) = \text{set } [] \ \cup \ \text{set } (v \ # \ va)$ 

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

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
```

recursive calls are guarded by conditions

```
-u-:--- MergeSort.thy 19% L24 (Isar Utoks Abbrev; Scripting )-----

proof (prove): step 1

goal (3 subgoals):
1.  $\forall x \ xs \ y \ ys.$ 
    $[x \leq y \implies \text{set } (\text{merge } xs \ (y \ # \ ys)) = \text{set } xs \ \cup \ \text{set } (y \ # \ ys);$ 
    $\neg x \leq y \implies \text{set } (\text{merge } (x \ # \ xs) \ ys) = \text{set } (x \ # \ xs) \ \cup \ \text{set } ys]$ 
 $\implies \text{set } (\text{merge } (x \ # \ xs) \ (y \ # \ ys)) = \text{set } (x \ # \ xs) \ \cup \ \text{set } (y \ # \ ys)$ 
2.  $\forall xs. \text{set } (\text{merge } xs \ []) = \text{set } xs \ \cup \ \text{set } []$ 
3.  $\forall v \ va. \text{set } (\text{merge } [] \ (v \ # \ va)) = \text{set } [] \ \cup \ \text{set } (v \ # \ va)$ 

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

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
```

recursive calls are guarded by conditions

```
proof (prove): step 1
goal (3 subgoals):
1.  $\wedge x \ xs \ y \ ys.$ 
    $[x \leq y \implies \text{set } (\text{merge } xs \ (y \ # \ ys)) = \text{set } xs \ \cup \ \text{set } (y \ # \ ys);$ 
    $\neg x \leq y \implies \text{set } (\text{merge } (x \ # \ xs) \ ys) = \text{set } (x \ # \ xs) \ \cup \ \text{set } ys]$ 
    $\implies \text{set } (\text{merge } (x \ # \ xs) \ (y \ # \ ys)) = \text{set } (x \ # \ xs) \ \cup \ \text{set } (y \ # \ ys)$ 
2.  $\wedge xs. \text{set } (\text{merge } xs \ []) = \text{set } xs \ \cup \ \text{set } []$ 
3.  $\wedge v \ va. \text{set } (\text{merge } [] \ (v \ # \ va)) = \text{set } [] \ \cup \ \text{set } (v \ # \ va)$ 
```

2 induction hypotheses, guarded by conditions!

Wrote /Users/lp15/Dropbox/ACS/4 - Advanced Recursion/MergeSort.thy

Proof Outline

$$\text{set } (\text{merge } (x\#xs) (y\#ys)) = \text{set } (x \# xs) \cup \text{set } (y \# ys)$$

$$\begin{aligned} & \text{set } (\text{if } x \leq y \text{ then } x \# \text{merge } xs (y\#ys) \\ & \quad \text{else } y \# \text{merge } (x\#xs) ys) = \dots \end{aligned}$$

=

$$\begin{aligned} & (x \leq y \rightarrow \text{set}(x \# \text{merge } xs (y\#ys)) = \dots) \& \\ & (\neg x \leq y \rightarrow \text{set}(y \# \text{merge } (x\#xs) ys) = \dots) \end{aligned}$$

=

$$\begin{aligned} & (x \leq y \rightarrow \{x\} \cup \text{set}(\text{merge } xs (y\#ys)) = \dots) \& \\ & (\neg x \leq y \rightarrow \{y\} \cup \text{set}(\text{merge } (x\#xs) ys) = \dots) \end{aligned}$$

=

$$\begin{aligned} & (x \leq y \rightarrow \{x\} \cup \text{set } xs \cup \text{set } (y \# ys) = \dots) \& \\ & (\neg x \leq y \rightarrow \{y\} \cup \text{set } (x \# xs) \cup \text{set } ys = \dots) \end{aligned}$$

Proof Outline

set (merge (x#xs) (y#ys)) = set (x # xs) U 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} U set(merge xs (y#ys)) = ...) &
(¬ x ≤ y → {y} U set(merge (x#xs) ys) = ...)

=

(x ≤ y → {x} U set xs U set (y # ys) = ...) &
(¬ x ≤ y → {y} U set (x # xs) U set ys = ...)

Proof Outline

`set (merge (x#xs) (y#ys)) = set (x # xs) U 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} U set(merge xs (y#ys)) = ...) &
(¬ x ≤ y → {y} U set(merge (x#xs) ys) = ...)`

`=`

`(x ≤ y → {x} U set xs U set (y # ys) = ...) &
(¬ x ≤ y → {y} U set (x # xs) U set ys = ...)`

Proof Outline

$$\text{set } (\text{merge } (x\#xs) (y\#ys)) = \text{set } (x \# xs) \cup \text{set } (y \# ys)$$

$$\text{set } (\text{if } x \leq y \text{ then } x \# \text{merge } xs (y\#ys) \\ \text{else } y \# \text{merge } (x\#xs) ys) = \dots$$

=

$$(x \leq y \rightarrow \text{set}(x \# \text{merge } xs (y\#ys)) = \dots) \& \\ (\neg x \leq y \rightarrow \text{set}(y \# \text{merge } (x\#xs) ys) = \dots)$$

=

$$(x \leq y \rightarrow \{x\} \cup \text{set}(\text{merge } xs (y\#ys)) = \dots) \& \\ (\neg x \leq y \rightarrow \{y\} \cup \text{set}(\text{merge } (x\#xs) ys) = \dots)$$

=

$$(x \leq y \rightarrow \{x\} \cup \text{set } xs \cup \text{set } (y \# ys) = \dots) \& \\ (\neg x \leq y \rightarrow \{y\} \cup \text{set } (x \# xs) \cup \text{set } ys = \dots)$$

Proof Outline

$$\text{set } (\text{merge } (x\#xs) (y\#ys)) = \text{set } (x \# xs) \cup \text{set } (y \# ys)$$

$$\text{set } (\text{if } x \leq y \text{ then } x \# \text{merge } xs (y\#ys) \\ \text{else } y \# \text{merge } (x\#xs) ys) = \dots$$

=

$$(x \leq y \rightarrow \text{set}(x \# \text{merge } xs (y\#ys)) = \dots) \& \\ (\neg x \leq y \rightarrow \text{set}(y \# \text{merge } (x\#xs) ys) = \dots)$$

=

$$(x \leq y \rightarrow \{x\} \cup \text{set}(\text{merge } xs (y\#ys)) = \dots) \& \\ (\neg x \leq y \rightarrow \{y\} \cup \text{set}(\text{merge } (x\#xs) ys) = \dots)$$

=

$$(x \leq y \rightarrow \{x\} \cup \text{set } xs \cup \text{set } (y \# ys) = \dots) \& \\ (\neg x \leq y \rightarrow \{y\} \cup \text{set } (x \# xs) \cup \text{set } ys = \dots)$$

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))"
```

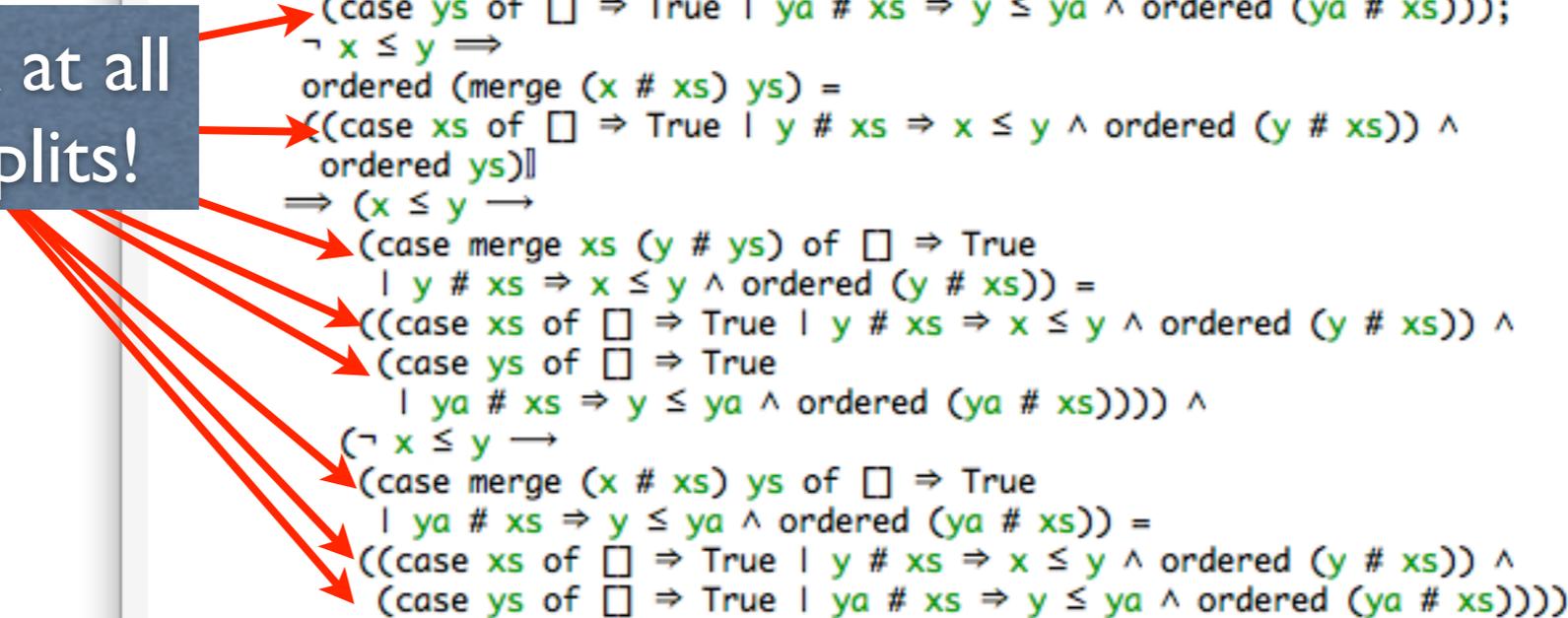
Case-Splits for Lists

```
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.  $\forall x \ xs \ y \ ys.$ 
    $\llbracket x \leq y \Rightarrow$ 
   ordered (merge xs (y # ys)) =
   (ordered xs  $\wedge$ 
   (case ys of  $\square \Rightarrow$  True |  $ya \ # \ xs \Rightarrow y \leq ya \wedge$  ordered (ya # xs)));
    $\neg x \leq y \Rightarrow$ 
   ordered (merge (x # xs) ys) =
   ((case xs of  $\square \Rightarrow$  True |  $y \ # \ xs \Rightarrow x \leq y \wedge$  ordered (y # xs))  $\wedge$ 
   ordered ys)
 $\Rightarrow (x \leq y \rightarrow$ 
   (case merge xs (y # ys) of  $\square \Rightarrow$  True
   |  $y \ # \ xs \Rightarrow x \leq y \wedge$  ordered (y # xs)) =
   ((case xs of  $\square \Rightarrow$  True |  $y \ # \ xs \Rightarrow x \leq y \wedge$  ordered (y # xs))  $\wedge$ 
   (case ys of  $\square \Rightarrow$  True
   |  $ya \ # \ xs \Rightarrow y \leq ya \wedge$  ordered (ya # xs))))  $\wedge$ 
   ( $\neg x \leq y \rightarrow$ 
   (case merge (x # xs) ys of  $\square \Rightarrow$  True
   |  $ya \ # \ xs \Rightarrow y \leq ya \wedge$  ordered (ya # xs)) =
   ((case xs of  $\square \Rightarrow$  True |  $y \ # \ xs \Rightarrow x \leq y \wedge$  ordered (y # xs))  $\wedge$ 
   (case ys of  $\square \Rightarrow$  True |  $ya \ # \ xs \Rightarrow y \leq ya \wedge$  ordered (ya # xs))))
-u:%%- *goals* 2% L4 (Isar Proofstate Utoks Abbrev;)-----
Write /Users/lp15/Dropbox/ACS/4 - Advanced Recursion/MergeSort.thy
```

Help! Look at all the case-splits!



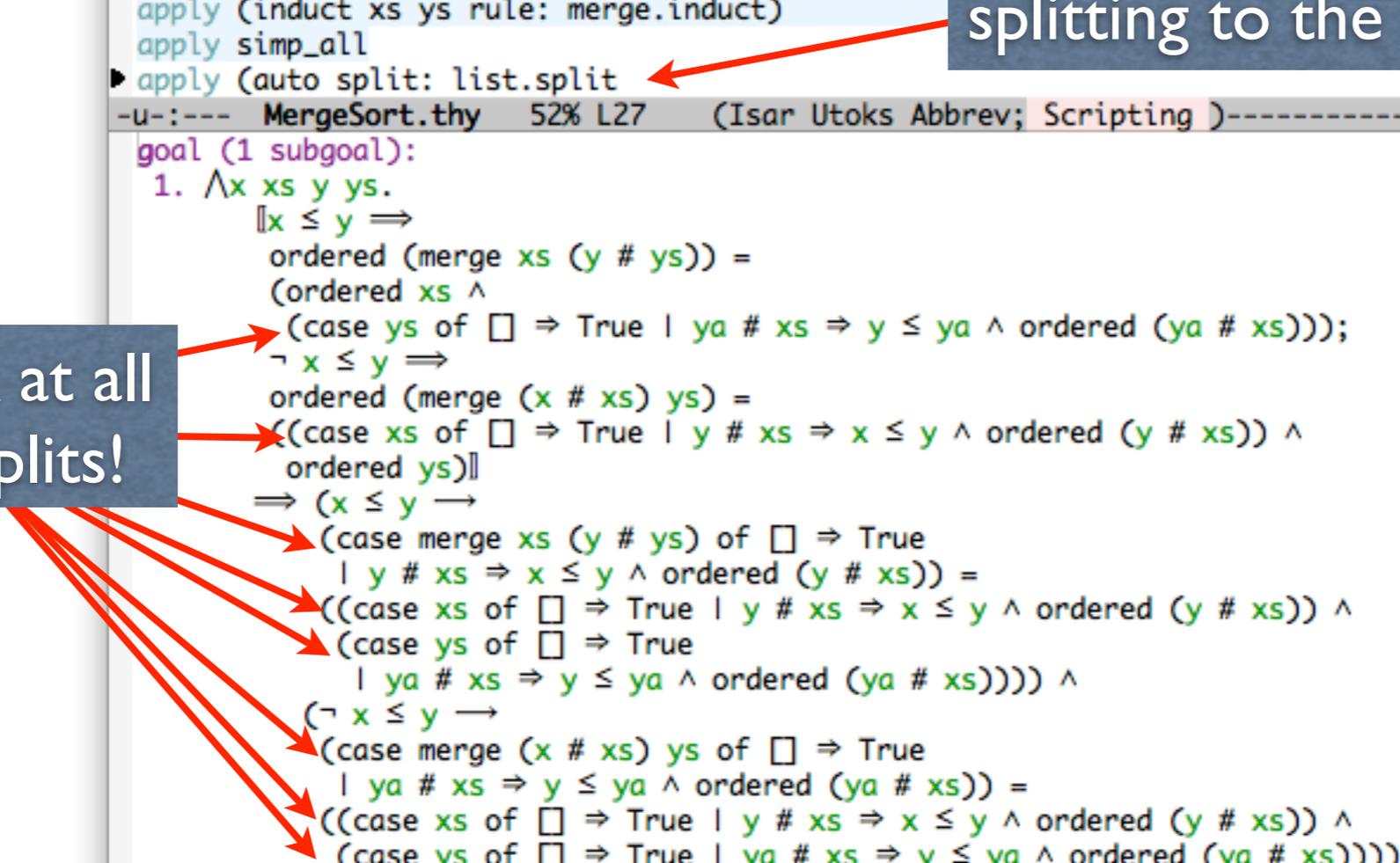
Case-Splitting in Action

```
lemma ordered_merge [simp]: "ordered (merge xs 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 )-----

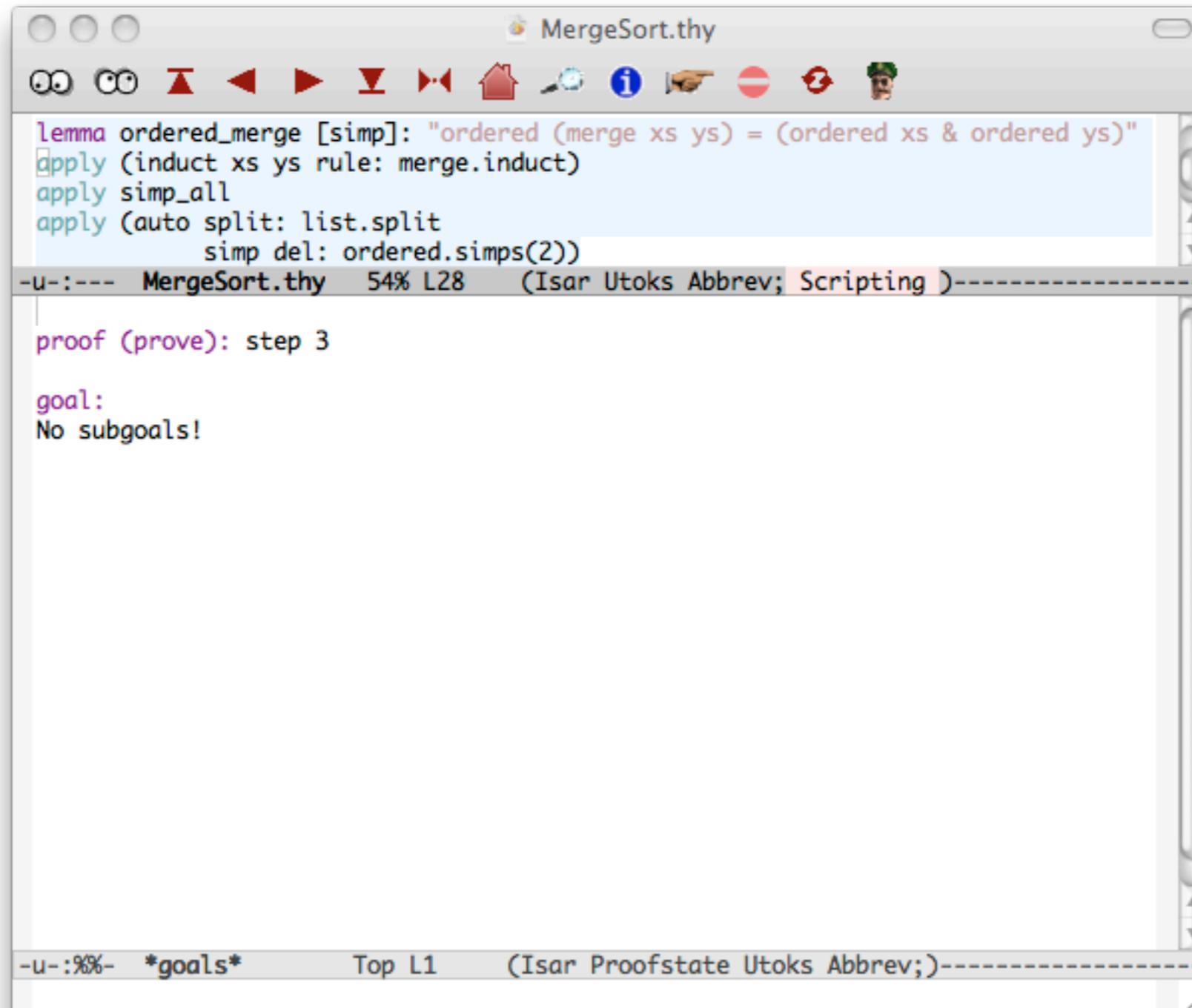
goal (1 subgoal):
  1.  $\forall x \ xs \ y \ ys.
    \quad \llbracket x \leq y \Rightarrow
      \quad \text{ordered (merge xs (y \# ys))} =
      \quad (\text{ordered xs} \wedge
      \quad (\text{case ys of } \square \Rightarrow \text{True} \mid ya \# xs \Rightarrow y \leq ya \wedge \text{ordered (ya \# xs)}));
      \neg x \leq y \Rightarrow
      \quad \text{ordered (merge (x \# xs) ys)} =
      \quad ((\text{case xs of } \square \Rightarrow \text{True} \mid y \# xs \Rightarrow x \leq y \wedge \text{ordered (y \# xs)}) \wedge
      \quad \text{ordered ys}) \rrbracket
    \Rightarrow (x \leq y \rightarrow
      \quad (\text{case merge xs (y \# ys) of } \square \Rightarrow \text{True}
      \quad \mid y \# xs \Rightarrow x \leq y \wedge \text{ordered (y \# xs)}) =
      \quad ((\text{case xs of } \square \Rightarrow \text{True} \mid y \# xs \Rightarrow x \leq y \wedge \text{ordered (y \# xs)}) \wedge
      \quad (\text{case ys of } \square \Rightarrow \text{True}
      \quad \mid ya \# xs \Rightarrow y \leq ya \wedge \text{ordered (ya \# xs)})) \wedge
      \quad (\neg x \leq y \rightarrow
      \quad (\text{case merge (x \# xs) ys of } \square \Rightarrow \text{True}
      \quad \mid ya \# xs \Rightarrow y \leq ya \wedge \text{ordered (ya \# xs)}) =
      \quad ((\text{case xs of } \square \Rightarrow \text{True} \mid y \# xs \Rightarrow x \leq y \wedge \text{ordered (y \# xs)}) \wedge
      \quad (\text{case ys of } \square \Rightarrow \text{True} \mid ya \# xs \Rightarrow y \leq ya \wedge \text{ordered (ya \# xs)}))
    )
-u:%%- *goals* 2% L4 (Isar Proofstate Utoks Abbrev; )-----
Write /Users/lp15/Dropbox/ACS/4 - Advanced Recursion/MergeSort.thy$ 
```

Automatic case splitting to the rescue!

Help! Look at all the case-splits!



Completing the Proof



The screenshot shows a window titled "MergeSort.thy" with a toolbar containing various navigation icons. The main text area contains the following code:

```
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  
        simp del: ordered.simps(2))
```

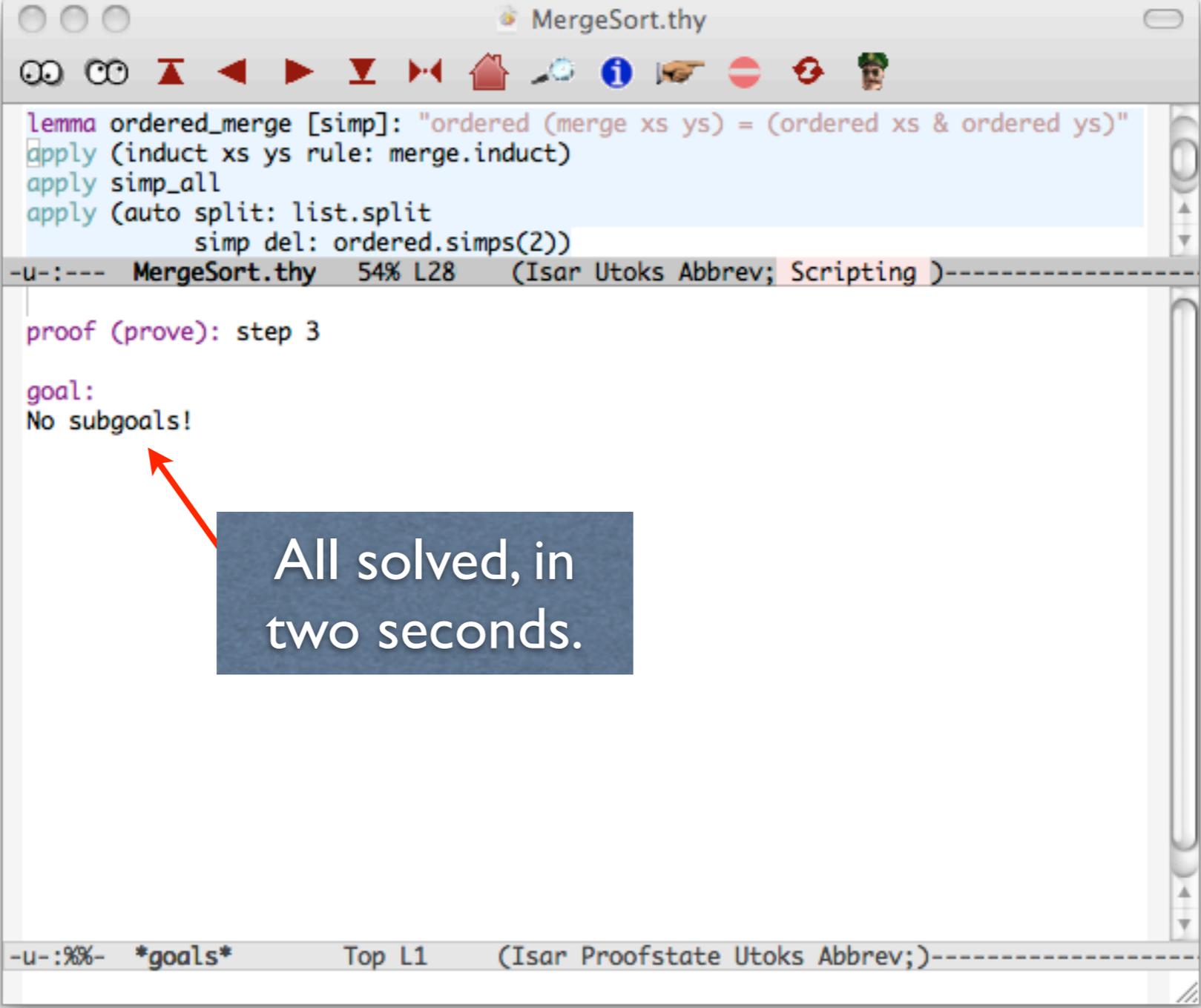
A status bar below the code indicates the current position: "-u:--- MergeSort.thy 54% L28 (Isar Utoks Abbrev; Scripting)-----".

The proof is shown as:

```
proof (prove): step 3  
goal:  
No subgoals!
```

A second status bar at the bottom indicates the current goal state: "-u:%%- *goals* Top L1 (Isar Proofstate Utoks Abbrev;)-----".

Completing the Proof



The screenshot shows a window titled "MergeSort.thy" with a toolbar and a text editor. The text editor contains the following code:

```
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  
      simp del: ordered.simps(2))
```

A status bar below the code shows: `-u:--- MergeSort.thy 54% L28 (Isar Utoks Abbrev; Scripting)`

The main text area shows the following code:

```
proof (prove): step 3  
goal:  
No subgoals!
```

A red arrow points from a dark blue box containing the text "All solved, in two seconds." to the "No subgoals!" line.

A second status bar at the bottom shows: `-u:%%- *goals* Top L1 (Isar Proofstate Utoks Abbrev;)`

Completing the Proof

The screenshot shows a window titled "MergeSort.thy" with a toolbar and a code editor. The code editor contains the following text:

```
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  
      simp del: ordered.simps(2))
```

Below the code is a status bar: `-u:--- MergeSort.thy 54% L28 (Isar Utoks Abbrev; Scripting)`

The main area shows a proof step:

```
proof (prove): step 3  
goal:  
No subgoals!
```

Two blue callout boxes with white text and red arrows point to the code and the goal:

- A box containing "All solved, in two seconds." points to the `goal:` line.
- A box containing "But what is this? Risk of looping!" points to the `apply (auto split: list.split simp del: ordered.simps(2))` line.

At the bottom, another status bar reads: `-u:%%- *goals* Top L1 (Isar Proofstate Utoks Abbrev;)`

Case Splitting for Lists

Simplification will replace

$P (\text{case } xs \text{ of } [] \Rightarrow a \mid \text{Cons } h \ tl \Rightarrow b \ h \ tl)$

by

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

Case Splitting for Lists

Simplification will replace

$P (\text{case } xs \text{ of } [] \Rightarrow a \mid \text{Cons } h \ tl \Rightarrow b \ h \ tl)$

by

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

- It creates a case for each datatype constructor.

Case Splitting for Lists

Simplification will replace

$P (\text{case } xs \text{ of } [] \Rightarrow a \mid \text{Cons } h \ tl \Rightarrow b \ h \ tl)$

by

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

- 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

