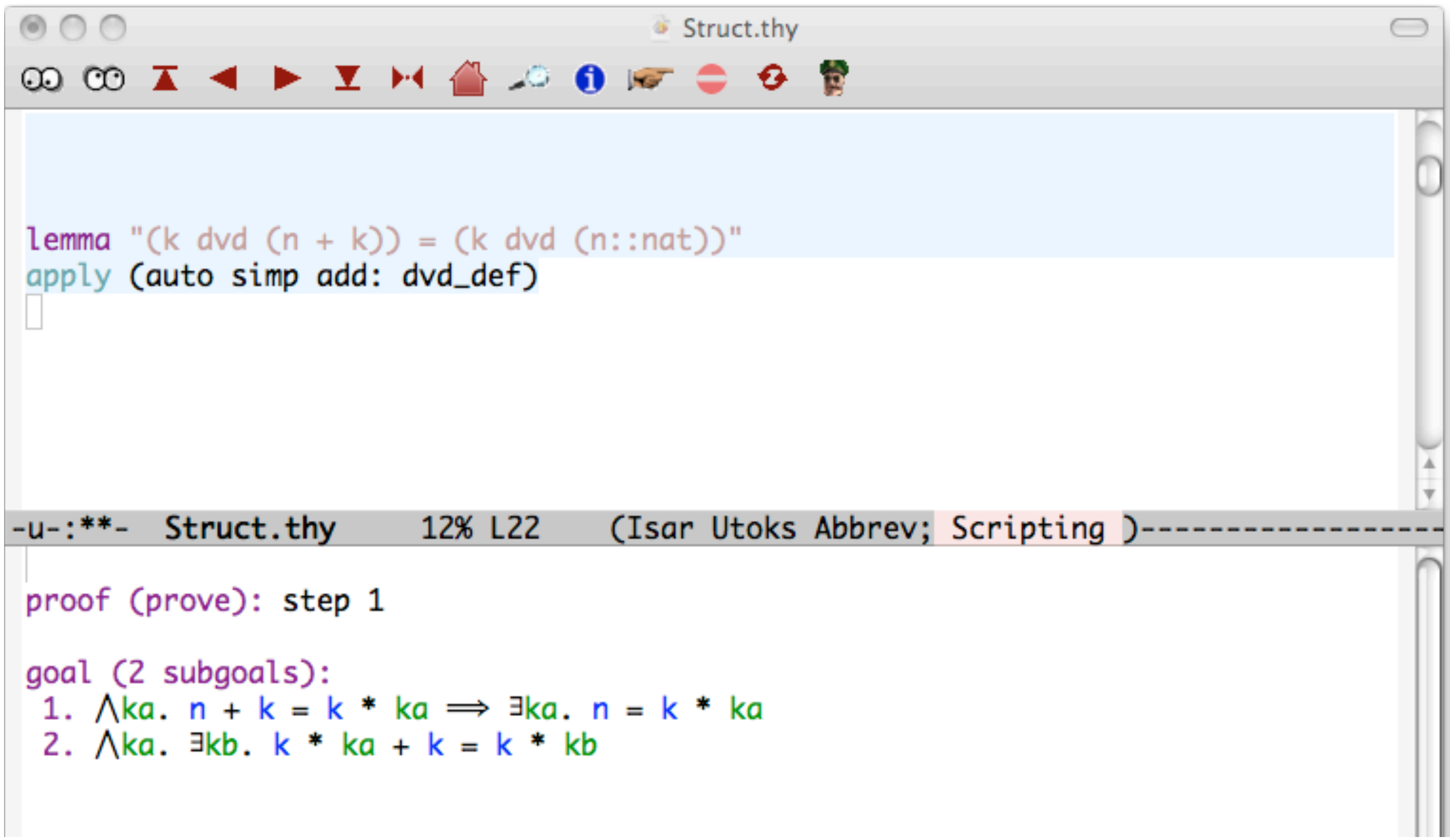
Interactive Formal Verification
10: Structured Proofs

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

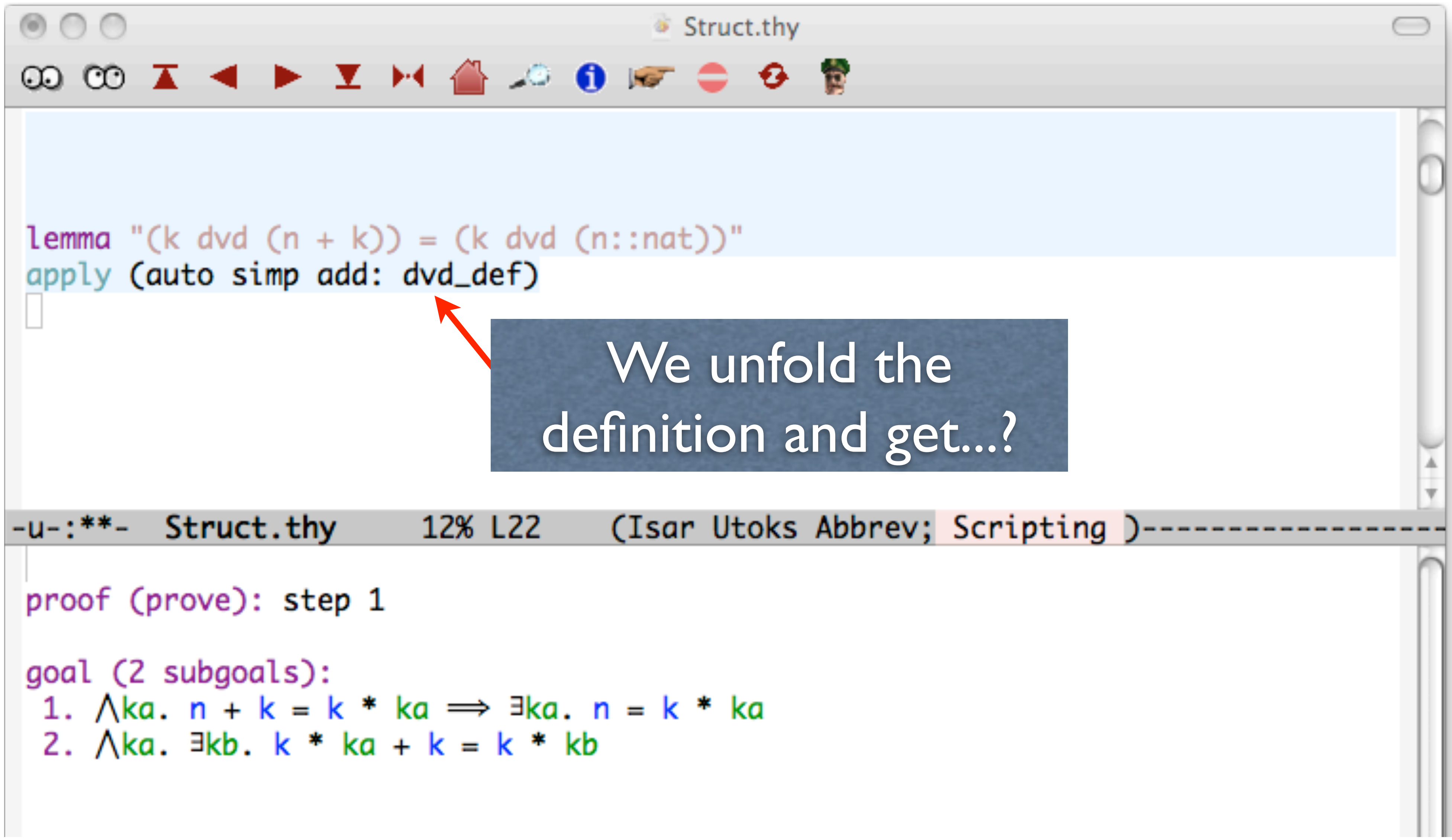
A Proof about "Divides"

$$b \text{ dvd } a \leftrightarrow (\exists k.\ a = b \times k)$$



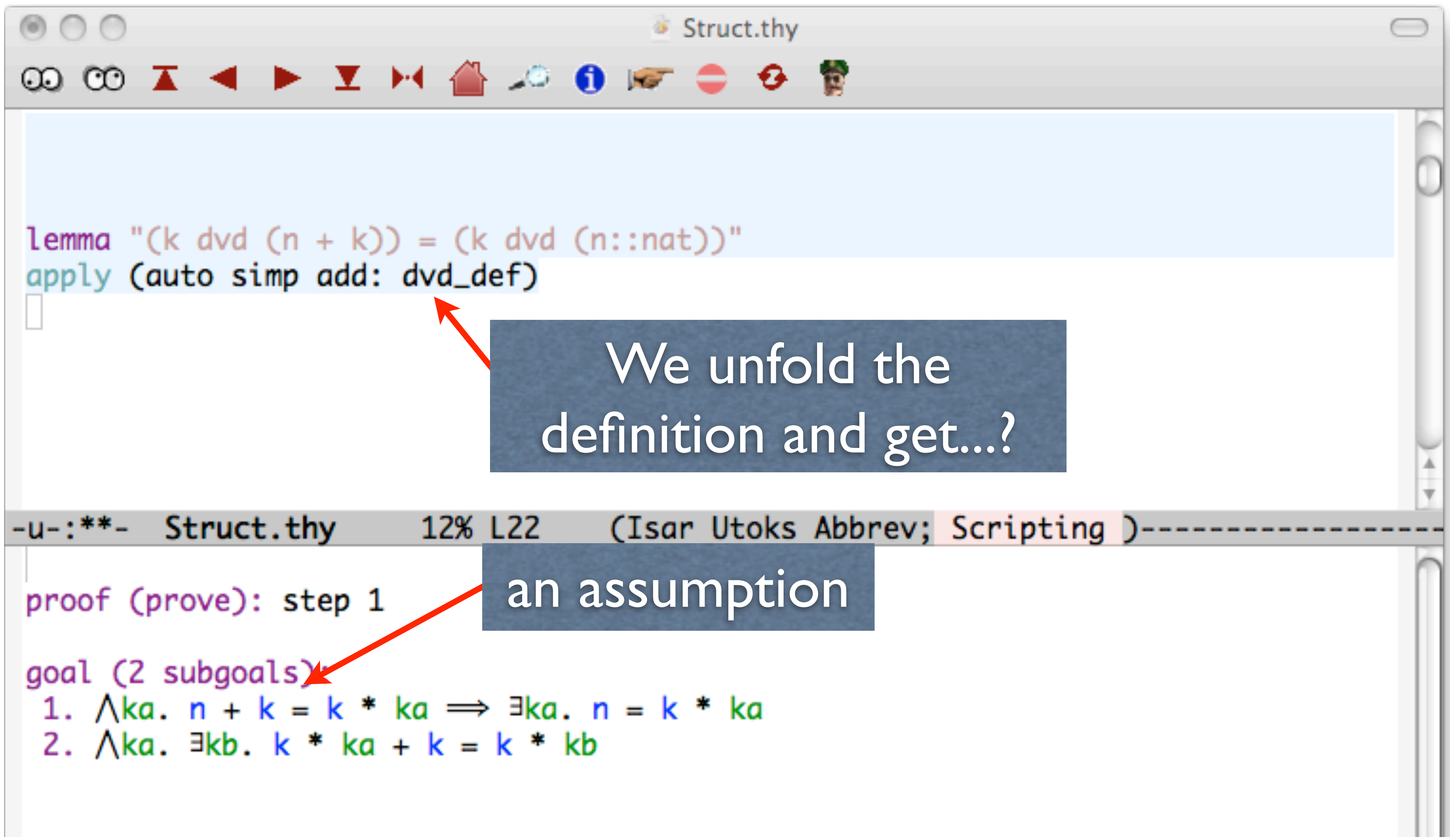
A Proof about "Divides"

b dvd a $\leftrightarrow$ ($\exists$k. a = b × k)



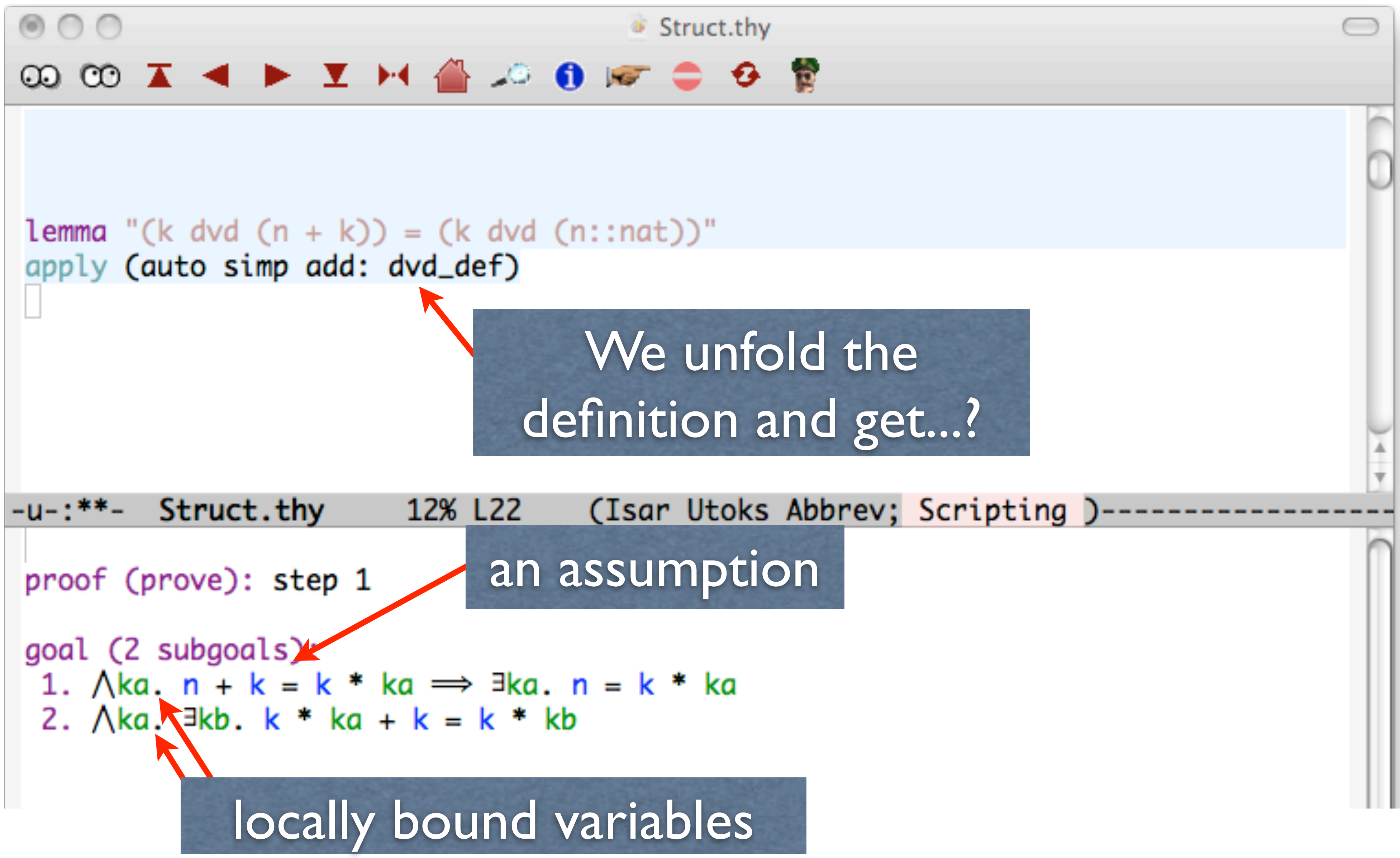
A Proof about "Divides"

$$b \text{ dvd } a \;\leftrightarrow\; (\exists k.\ a = b \times k)$$



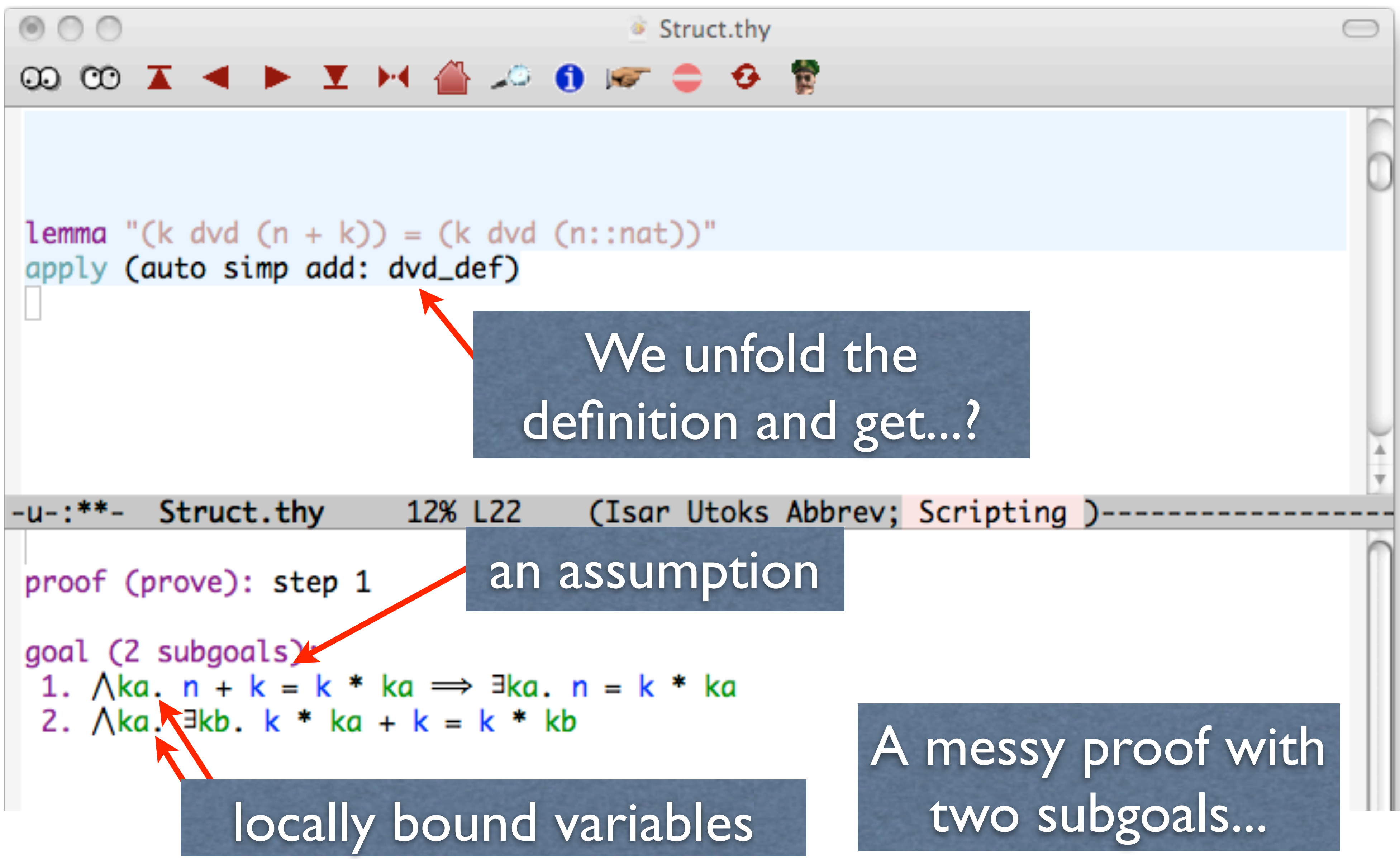
A Proof about "Divides"

$$b \text{ dvd } a \leftrightarrow (\exists k.\ a = b \times k)$$



A Proof about "Divides"

$$b \text{ dvd } a \leftrightarrow (\exists k.\ a = b \times k)$$



Complex Subgoals

Complex Subgoals

- Isabelle provides many tactics that refer to bound variables and assumptions.
  - Assumptions are often found by matching.
  - Bound variables can be referred to by name, but these names are fragile.

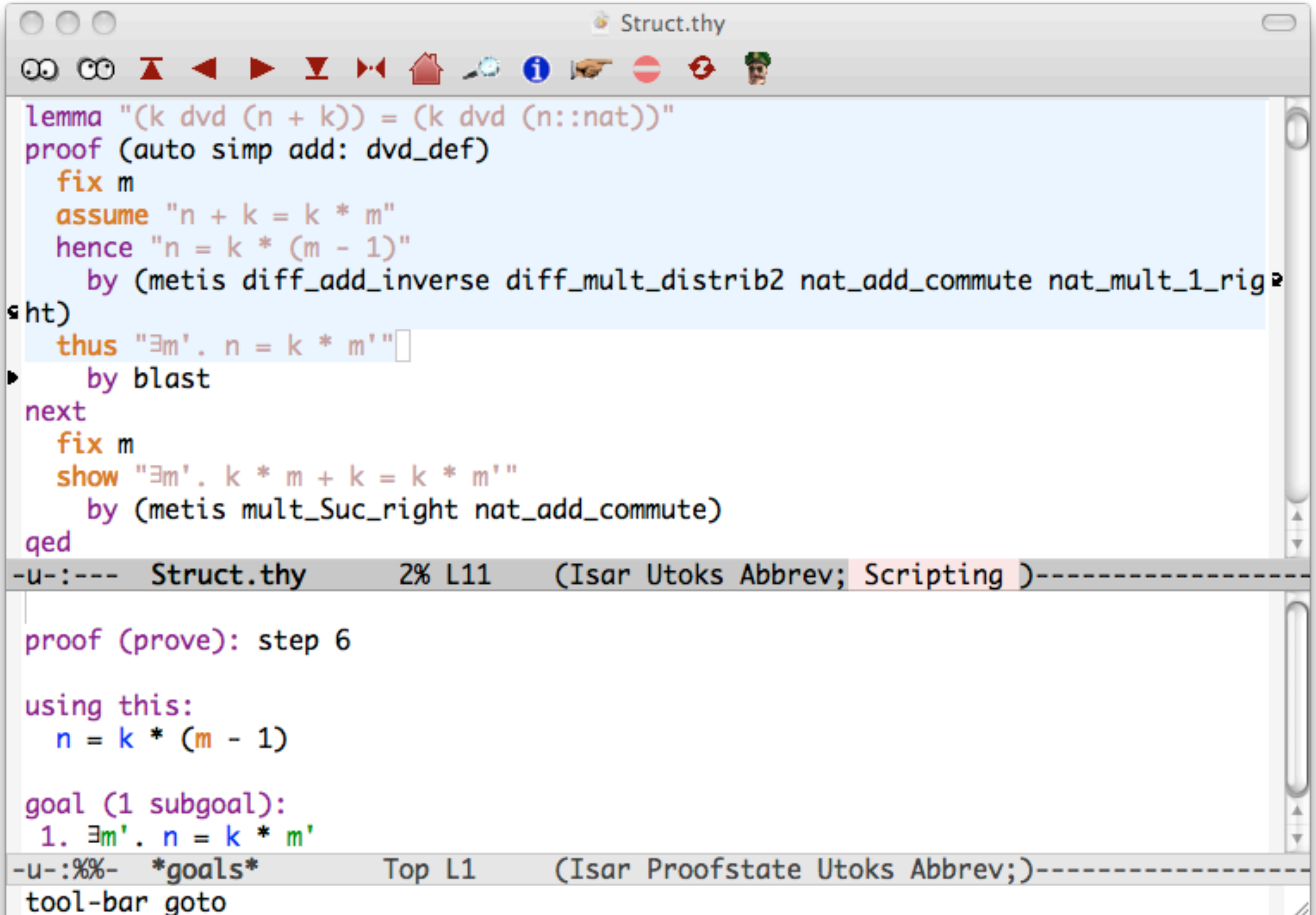
Complex Subgoals

- Isabelle provides many tactics that refer to bound variables and assumptions.
    - Assumptions are often found by matching.
    - Bound variables can be referred to by name, but these names are fragile.
- *Structured proofs* provide a robust means of referring to these elements by name.

Complex Subgoals

- Isabelle provides many tactics that refer to bound variables and assumptions.
  - Assumptions are often found by matching.
  - Bound variables can be referred to by name, but these names are fragile.
- *Structured proofs* provide a robust means of referring to these elements by name.
- Structured proofs are typically verbose but much more readable than linear apply-proofs.

A Structured Proof



But how do you write them?

The Elements of Isar

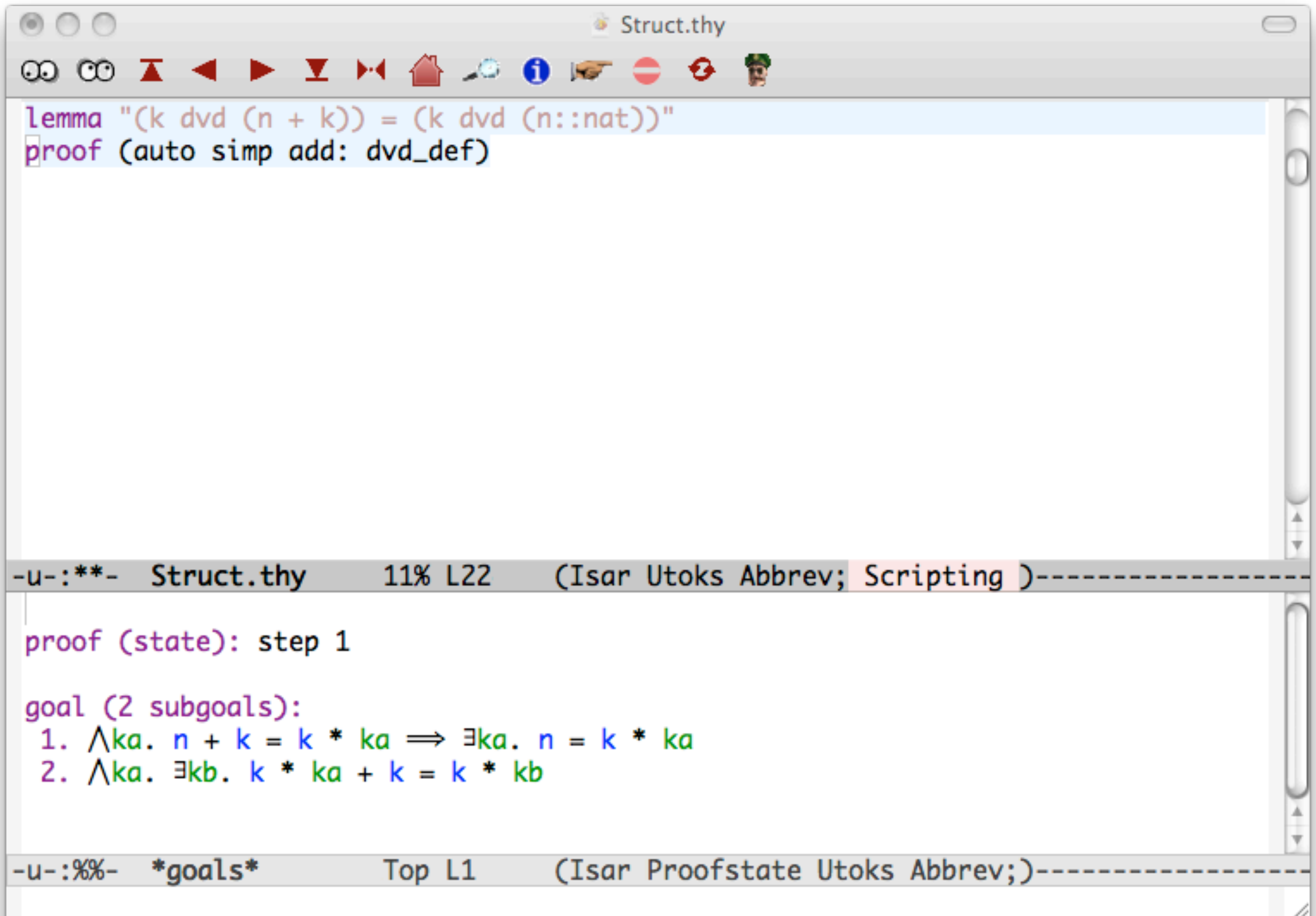
The Elements of Isar

- A *proof context* holds local variables and assumptions of a subgoal.
  - In a context, the variables are free and the assumptions are simply theorems.
  - Closing a context yields a theorem having the structure of a subgoal.

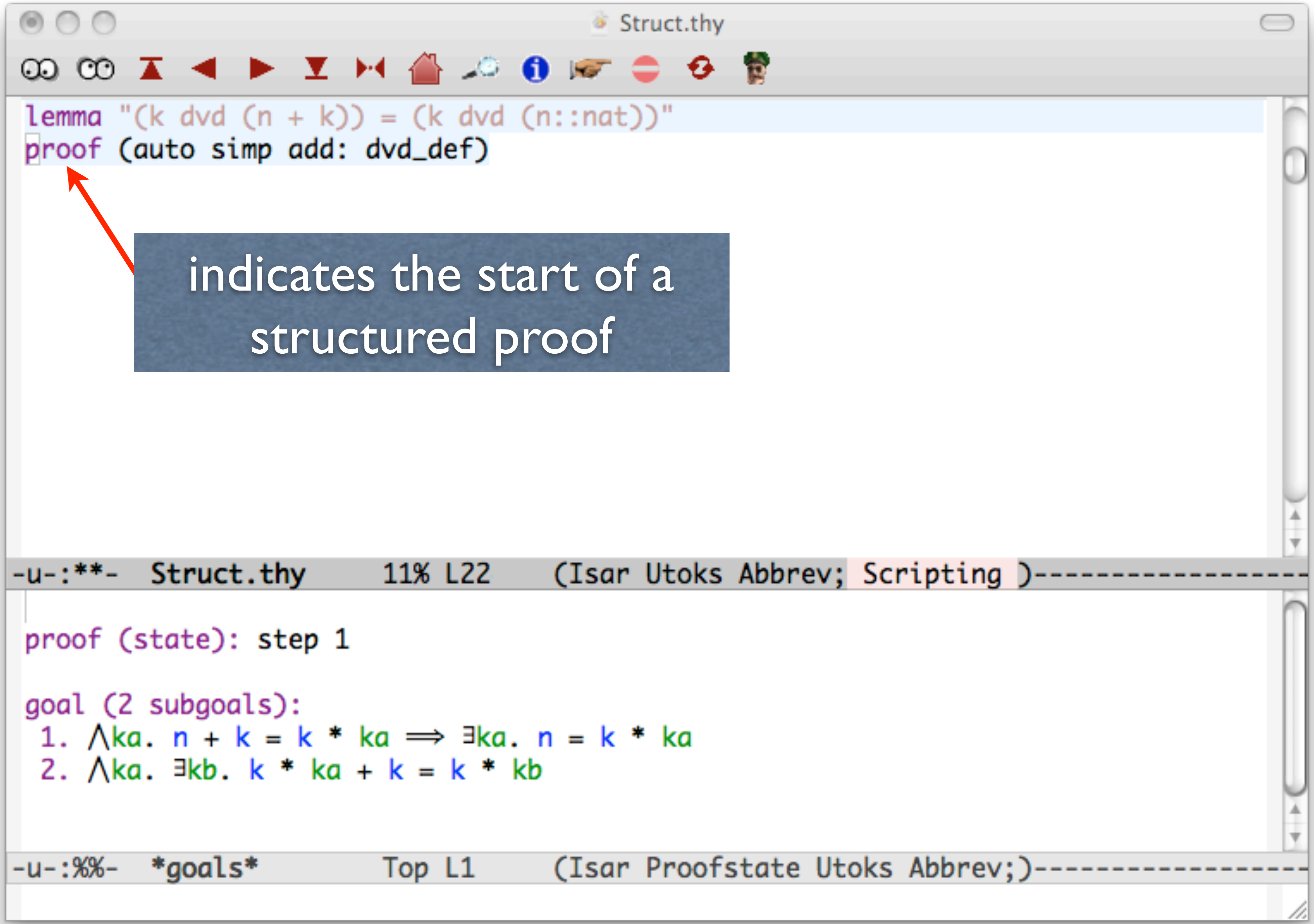
The Elements of Isar

- A *proof context* holds local variables and assumptions of a subgoal.
    - In a context, the variables are free and the assumptions are simply theorems.
    - Closing a context yields a theorem having the structure of a subgoal.
- The Isar language lets us state and prove intermediate results, express inductions, etc.

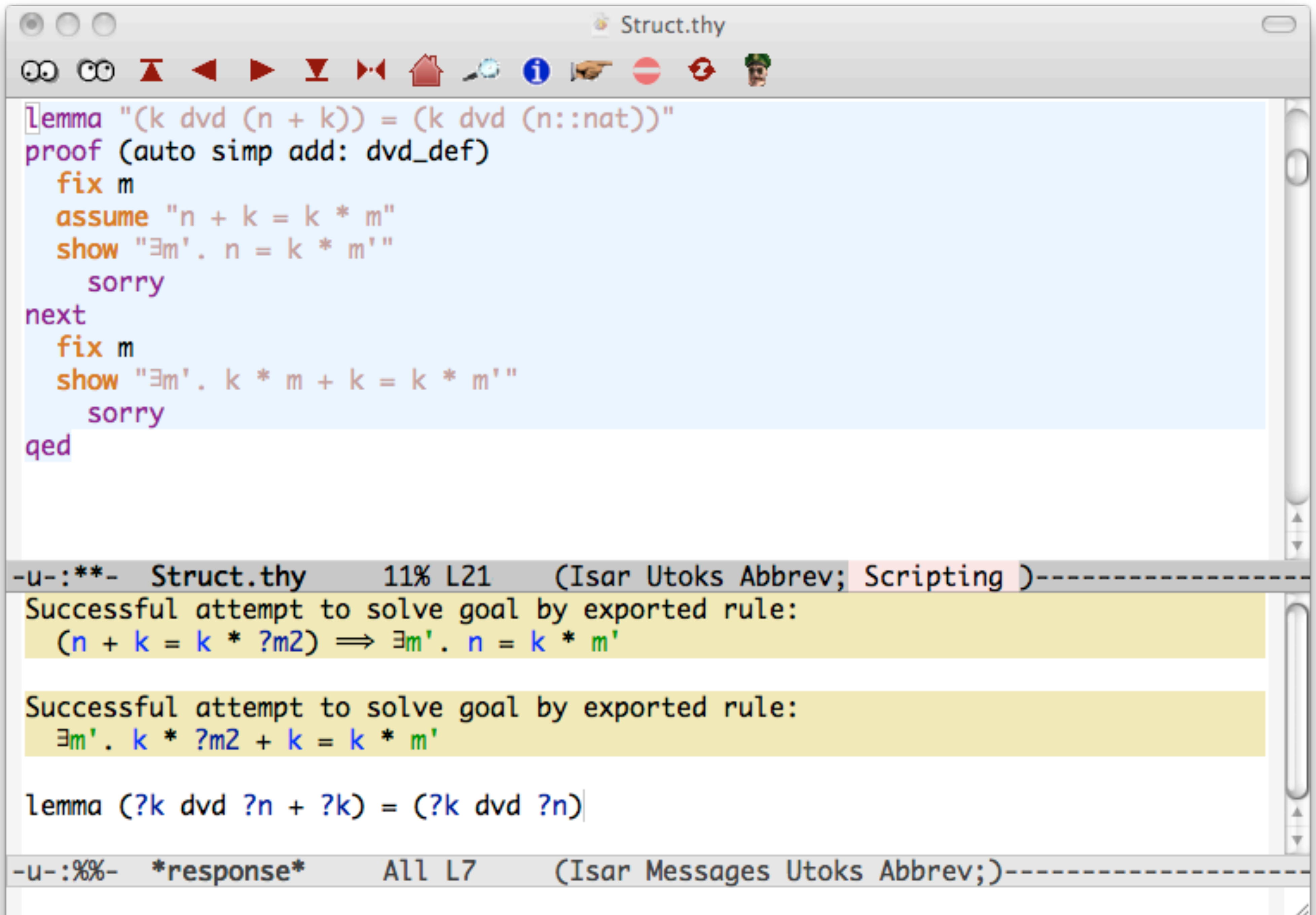
Getting Started



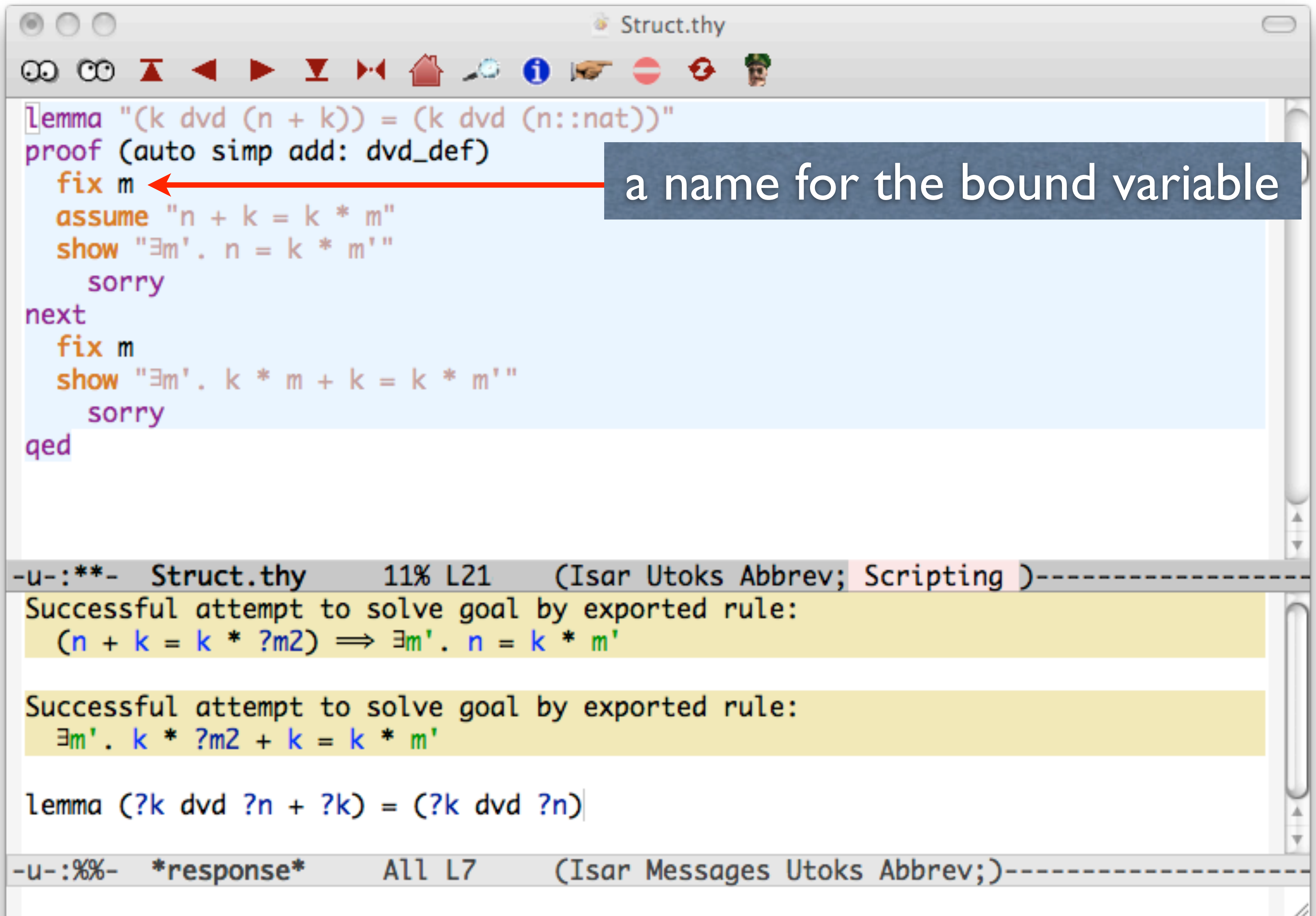
Getting Started



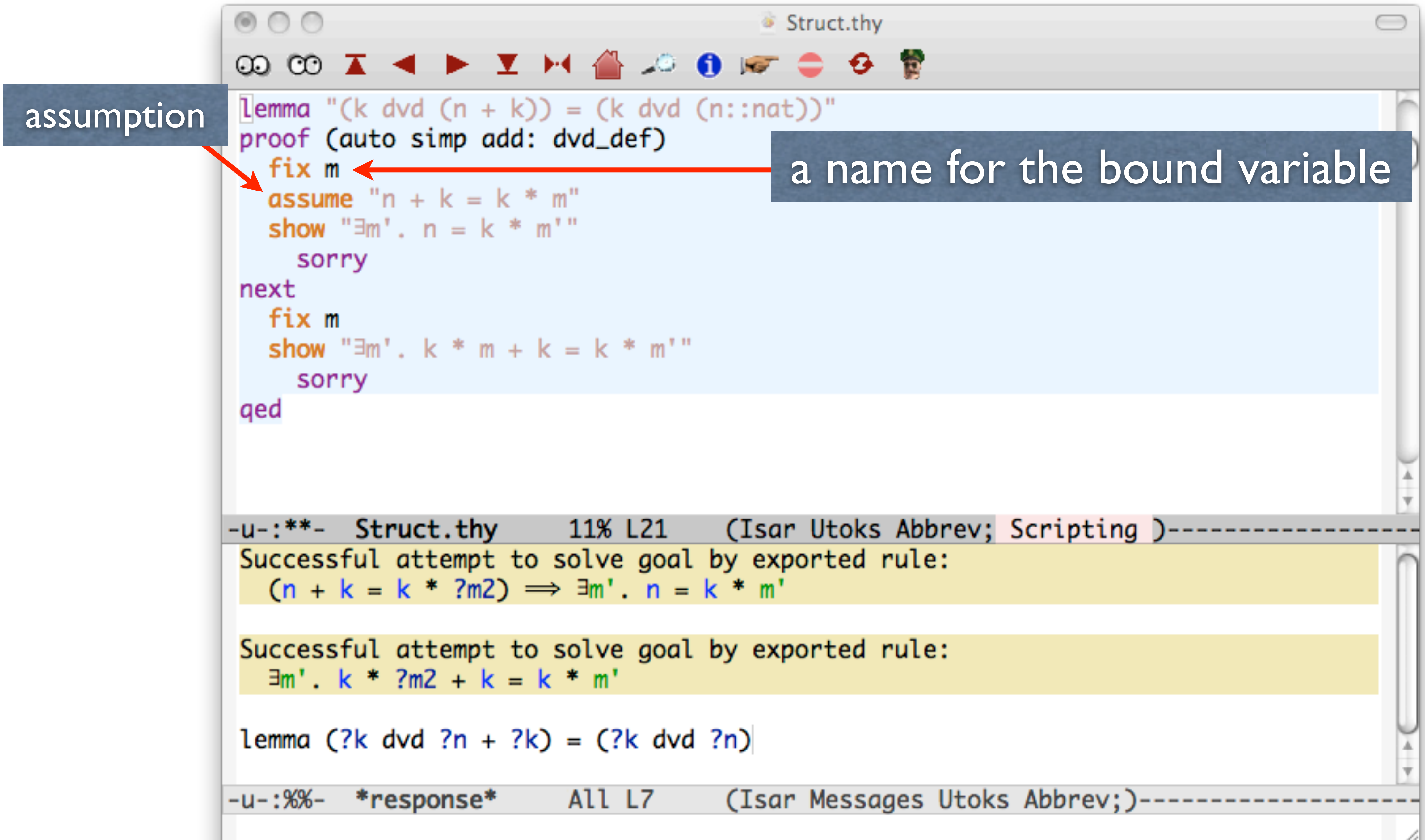
The Proof Skeleton



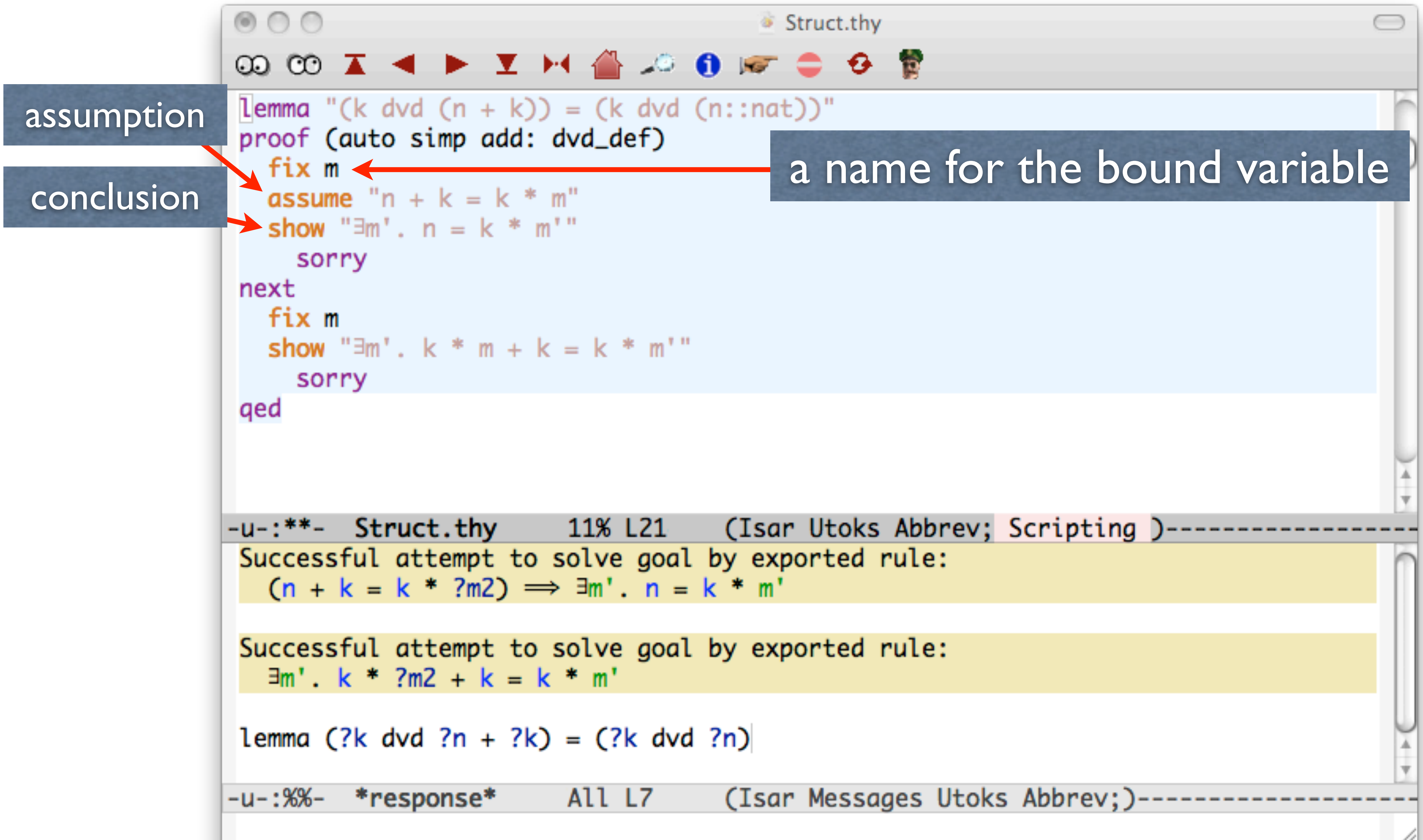
The Proof Skeleton



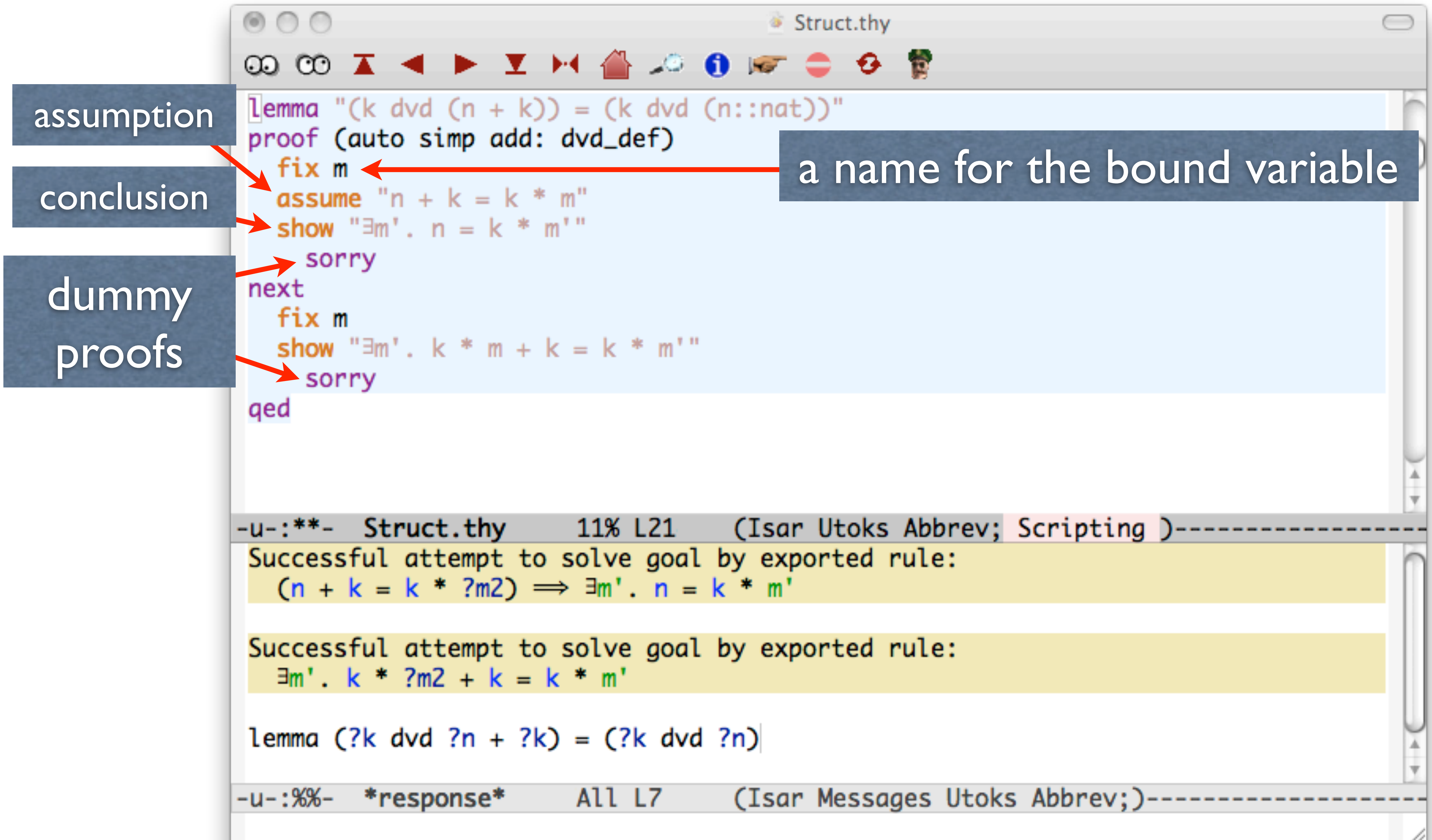
The Proof Skeleton



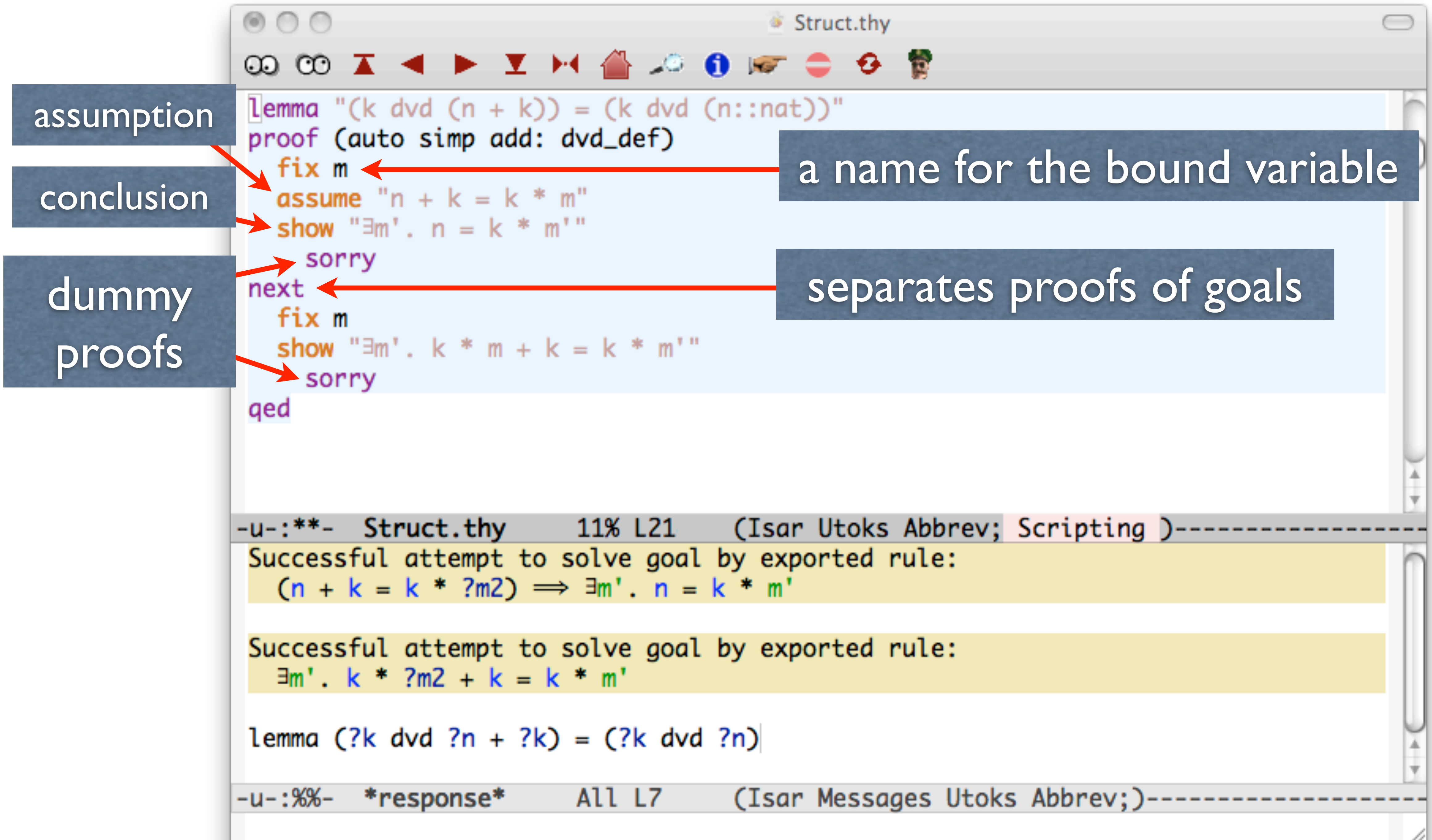
The Proof Skeleton



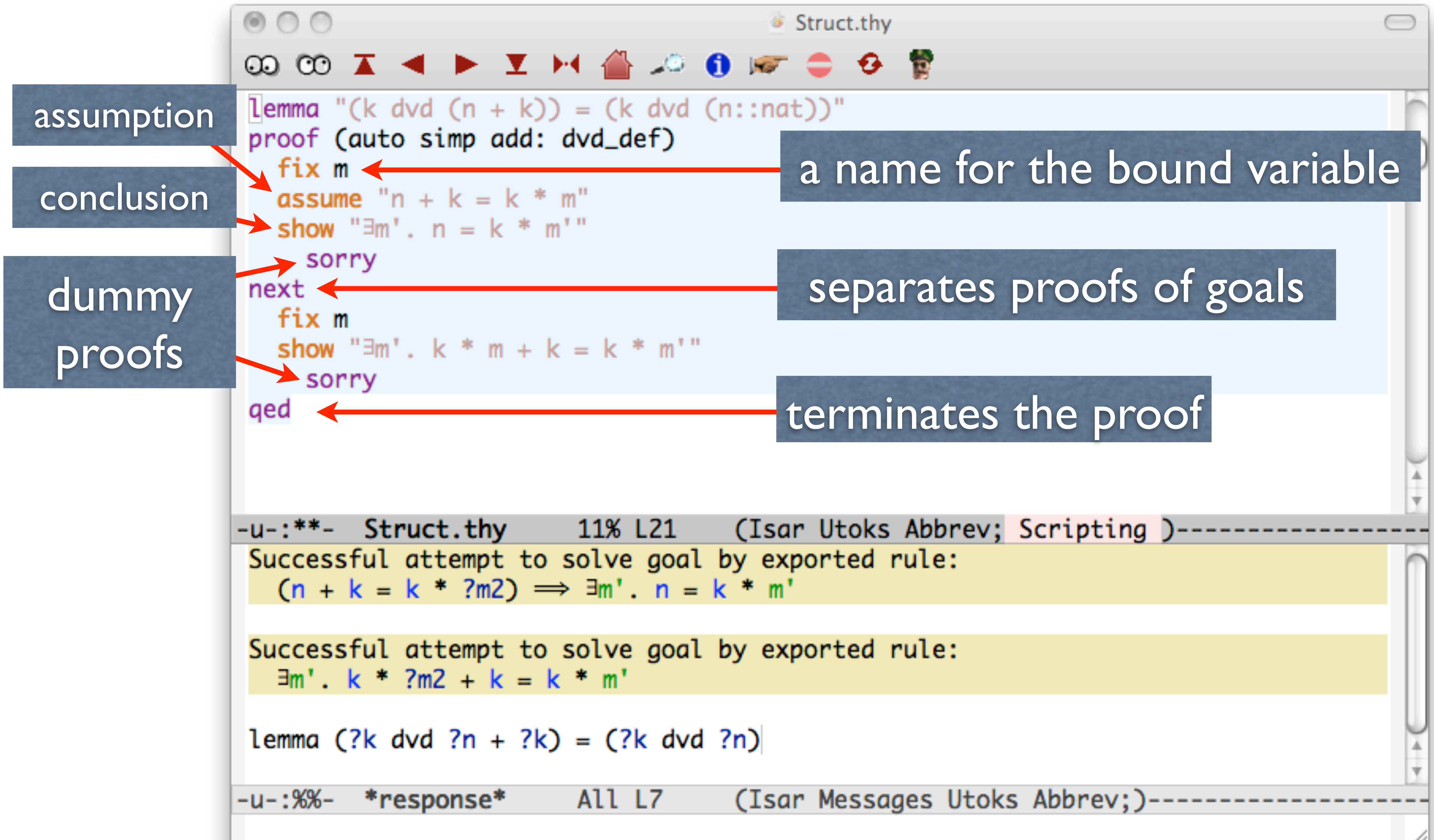
The Proof Skeleton



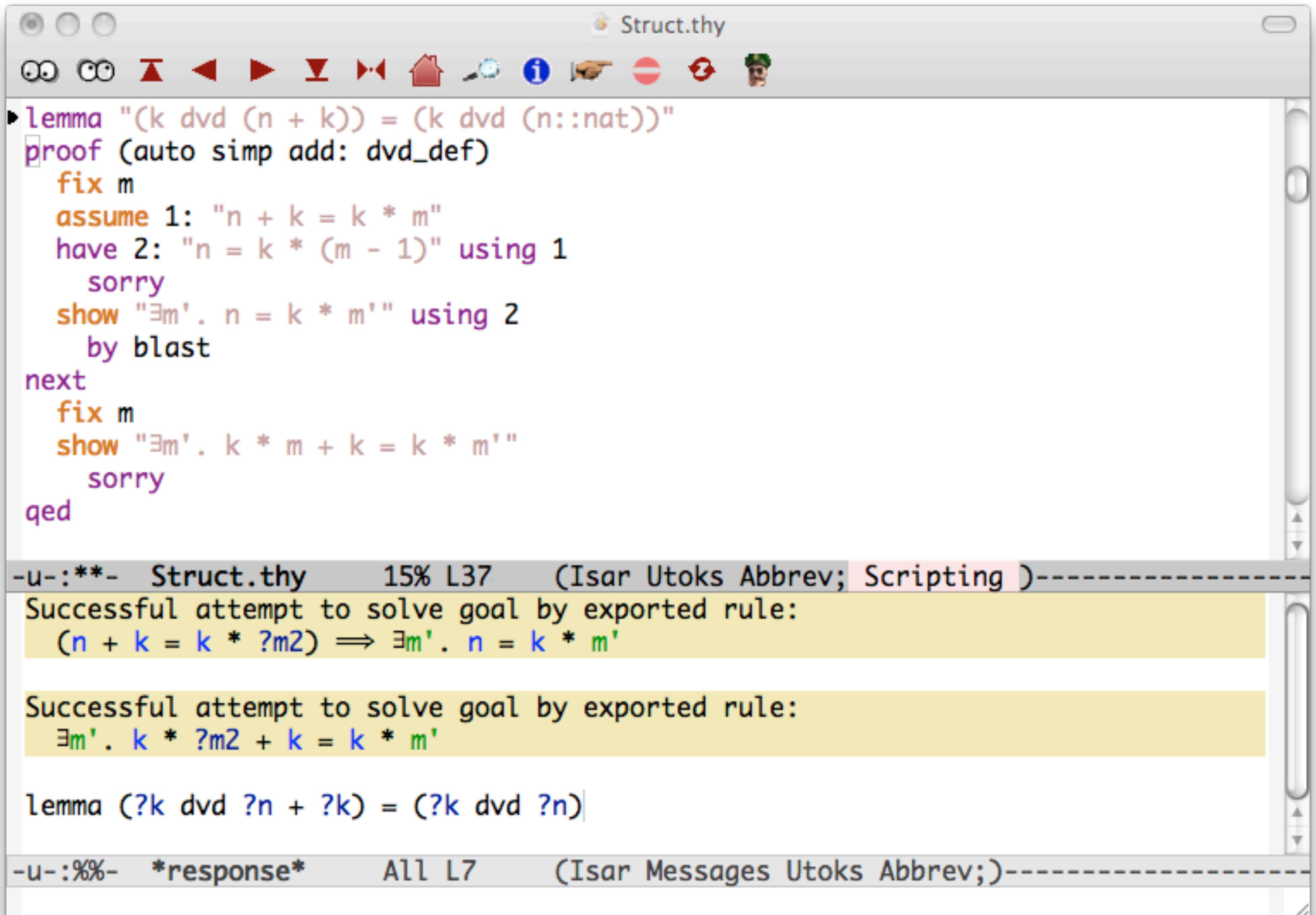
The Proof Skeleton



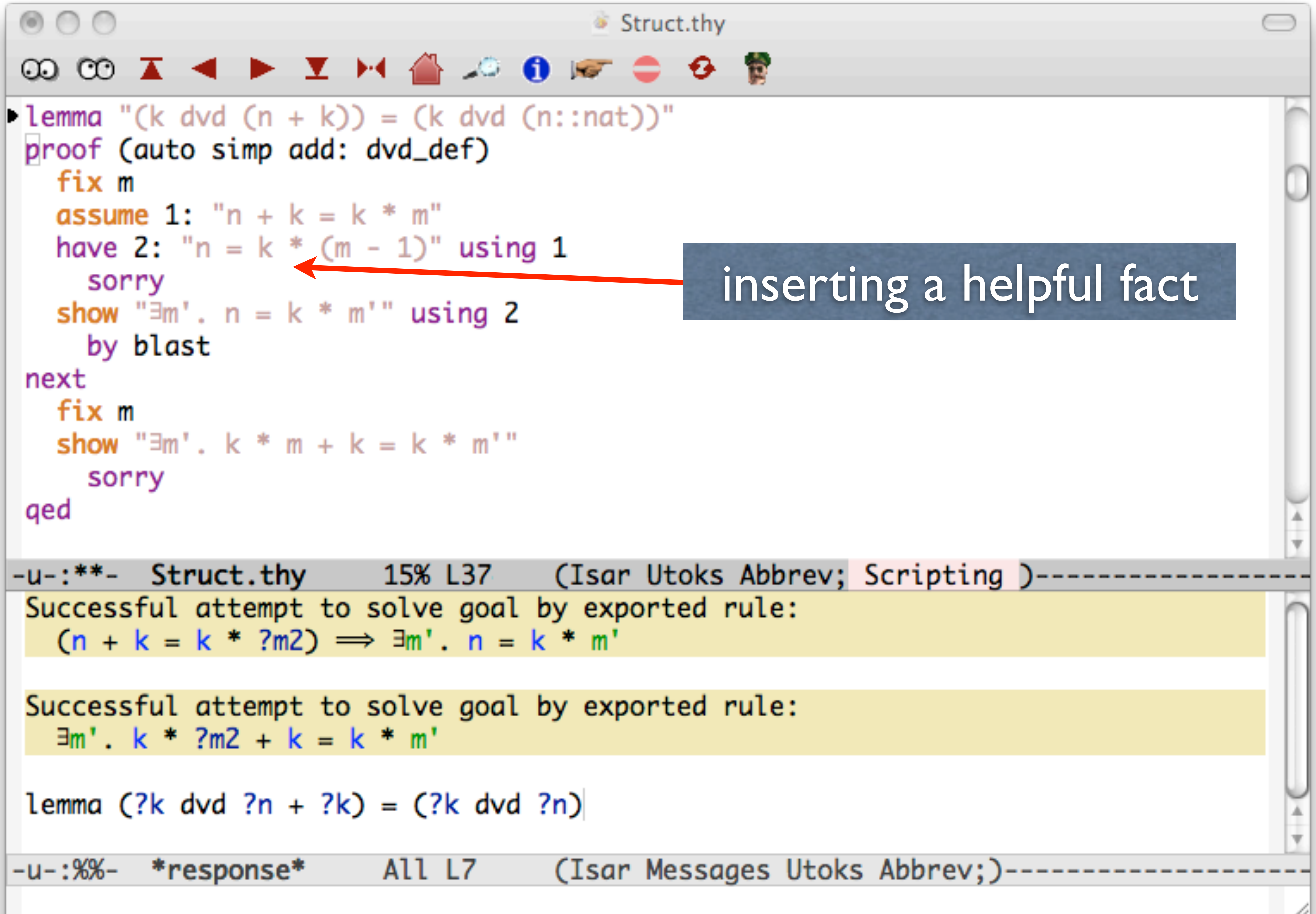
The Proof Skeleton



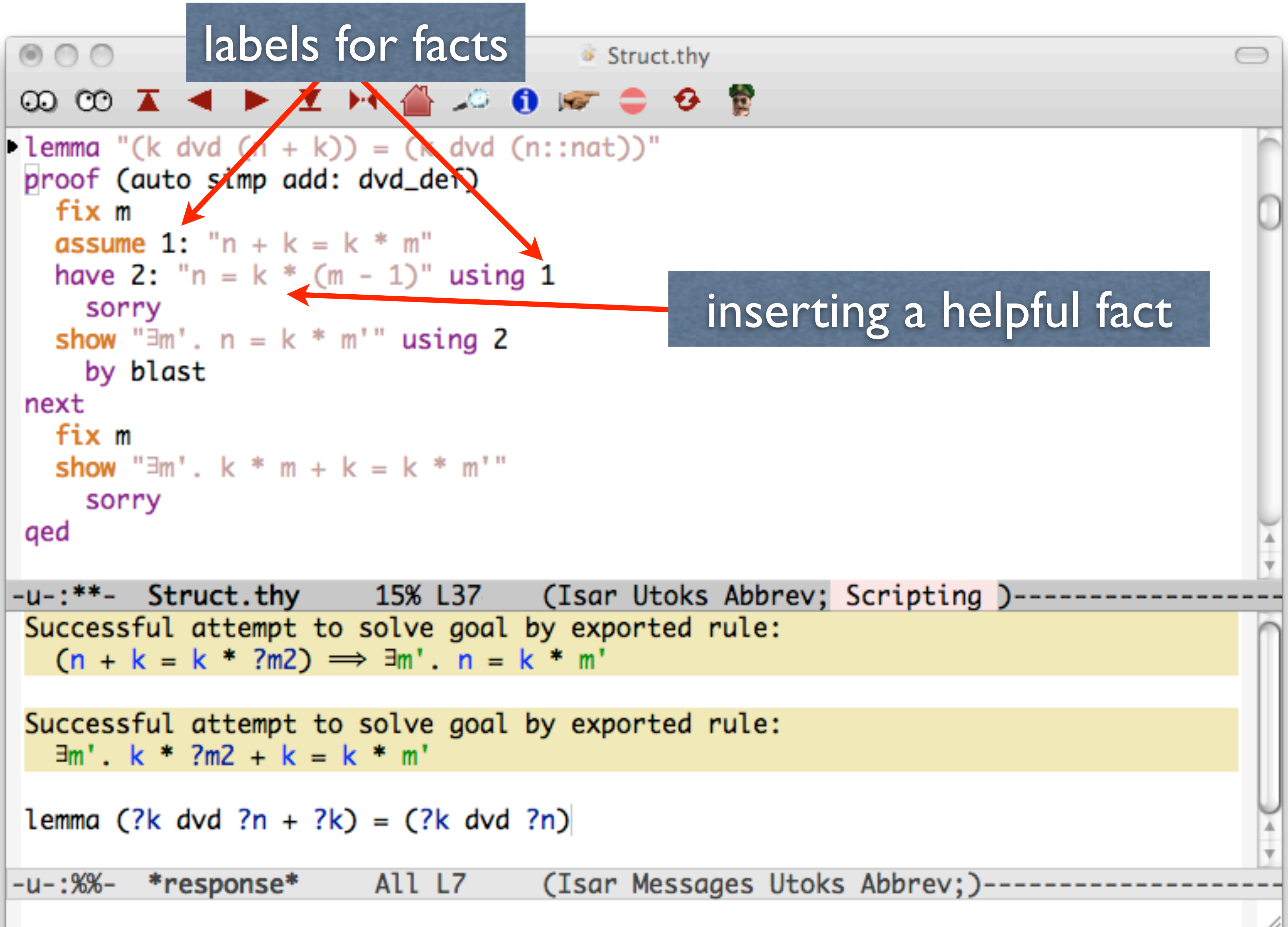
Fleshing Out that Skeleton



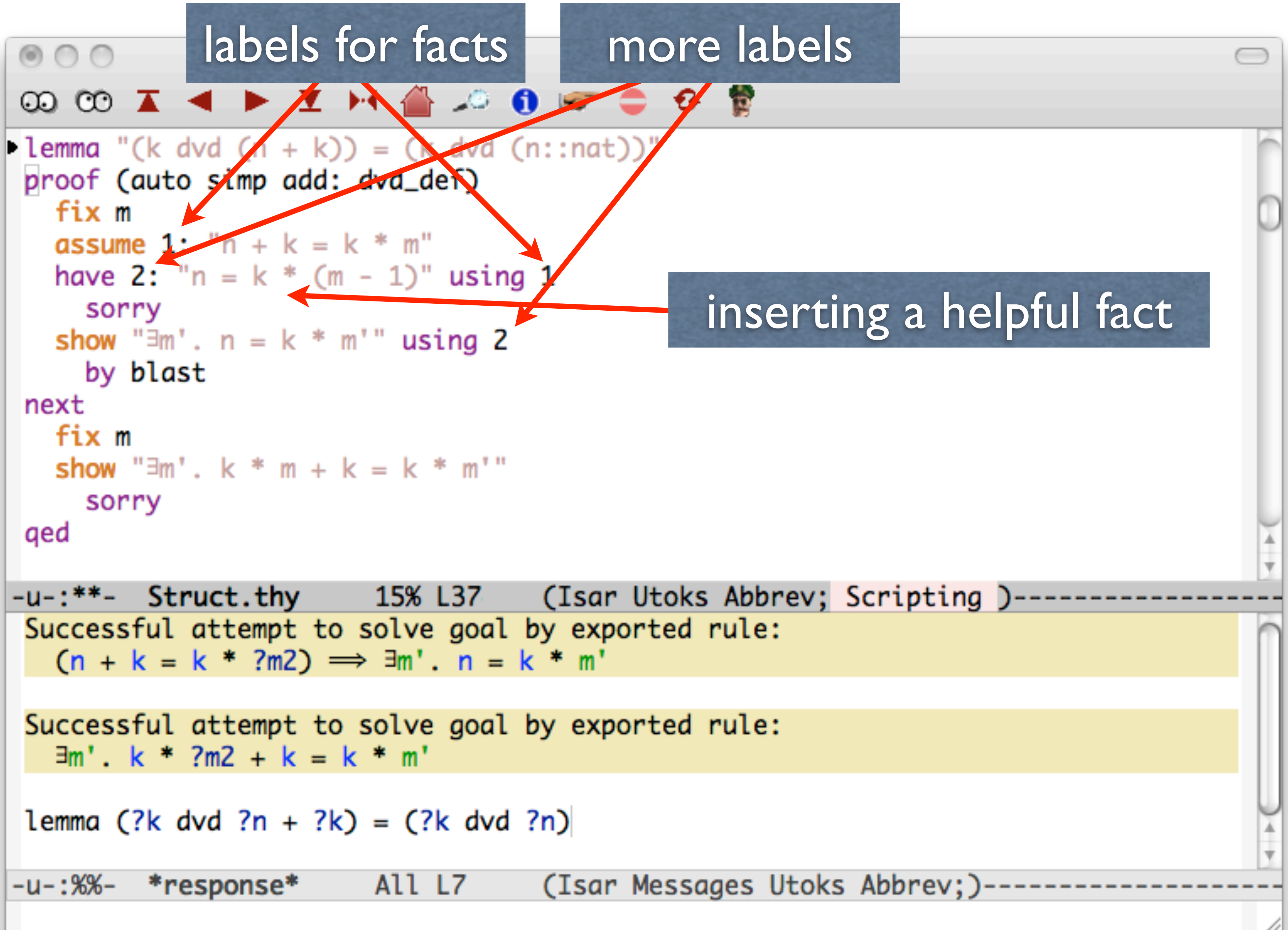
Fleshing Out that Skeleton



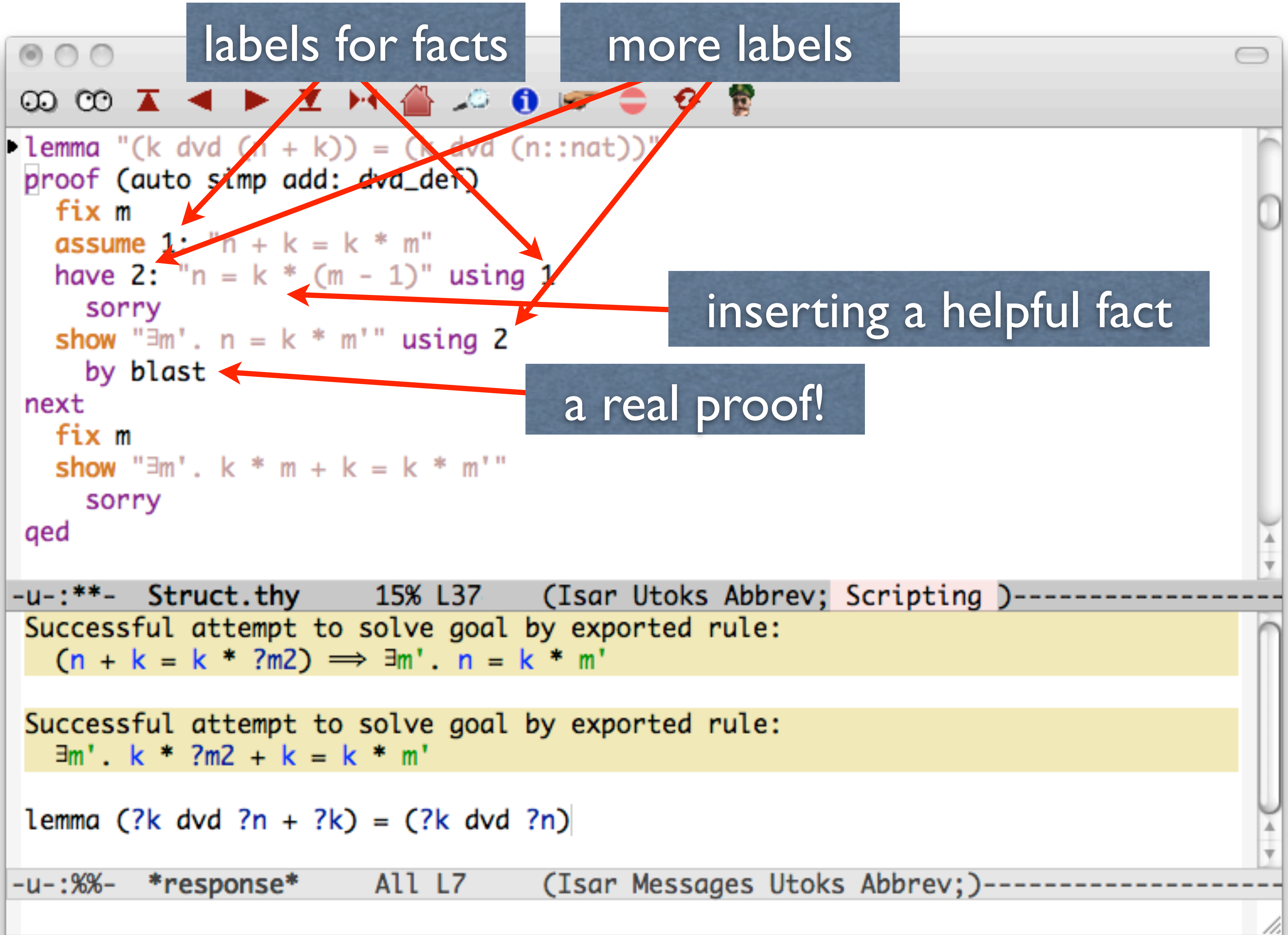
Fleshing Out that Skeleton



Fleshing Out that Skeleton



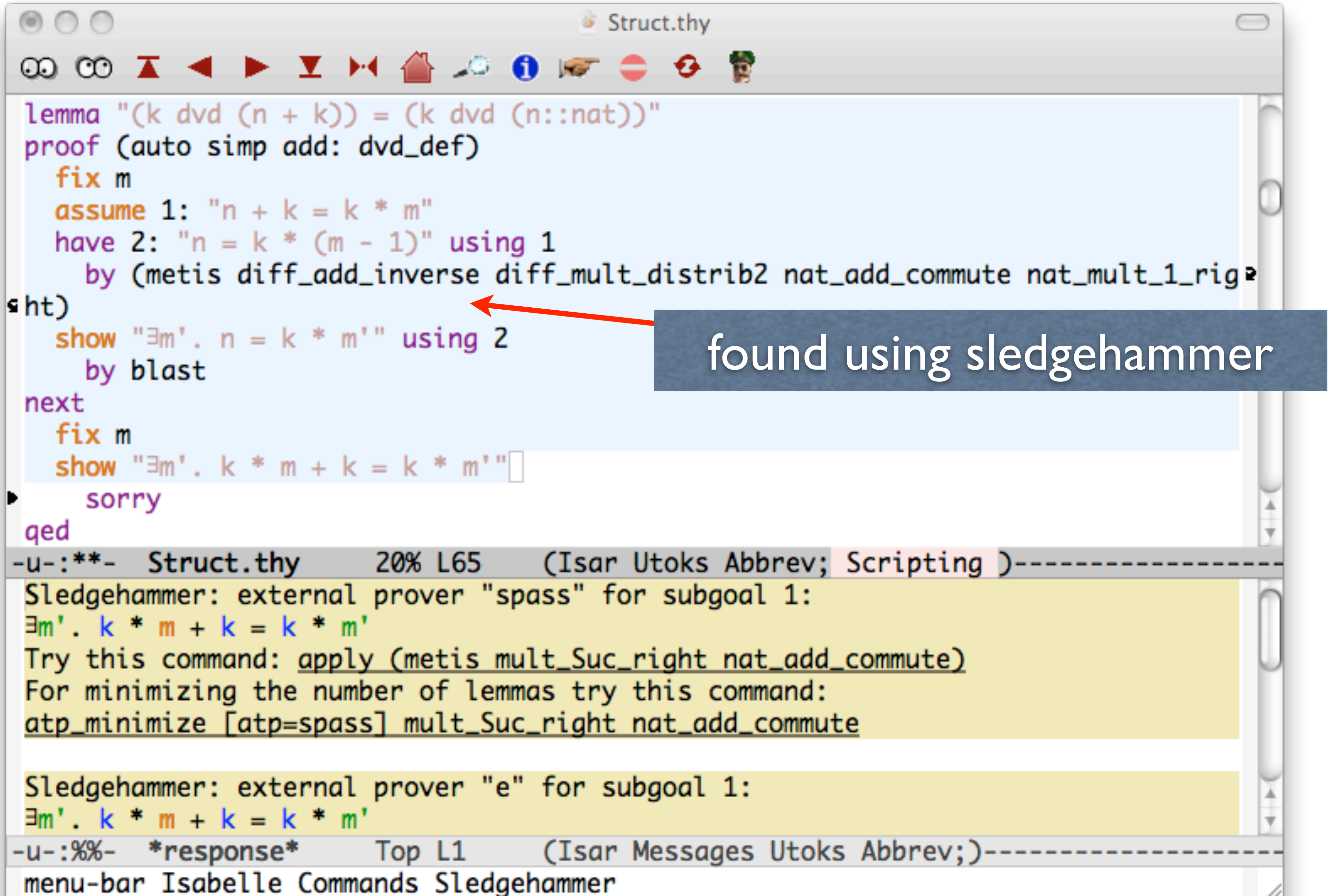
Fleshing Out that Skeleton



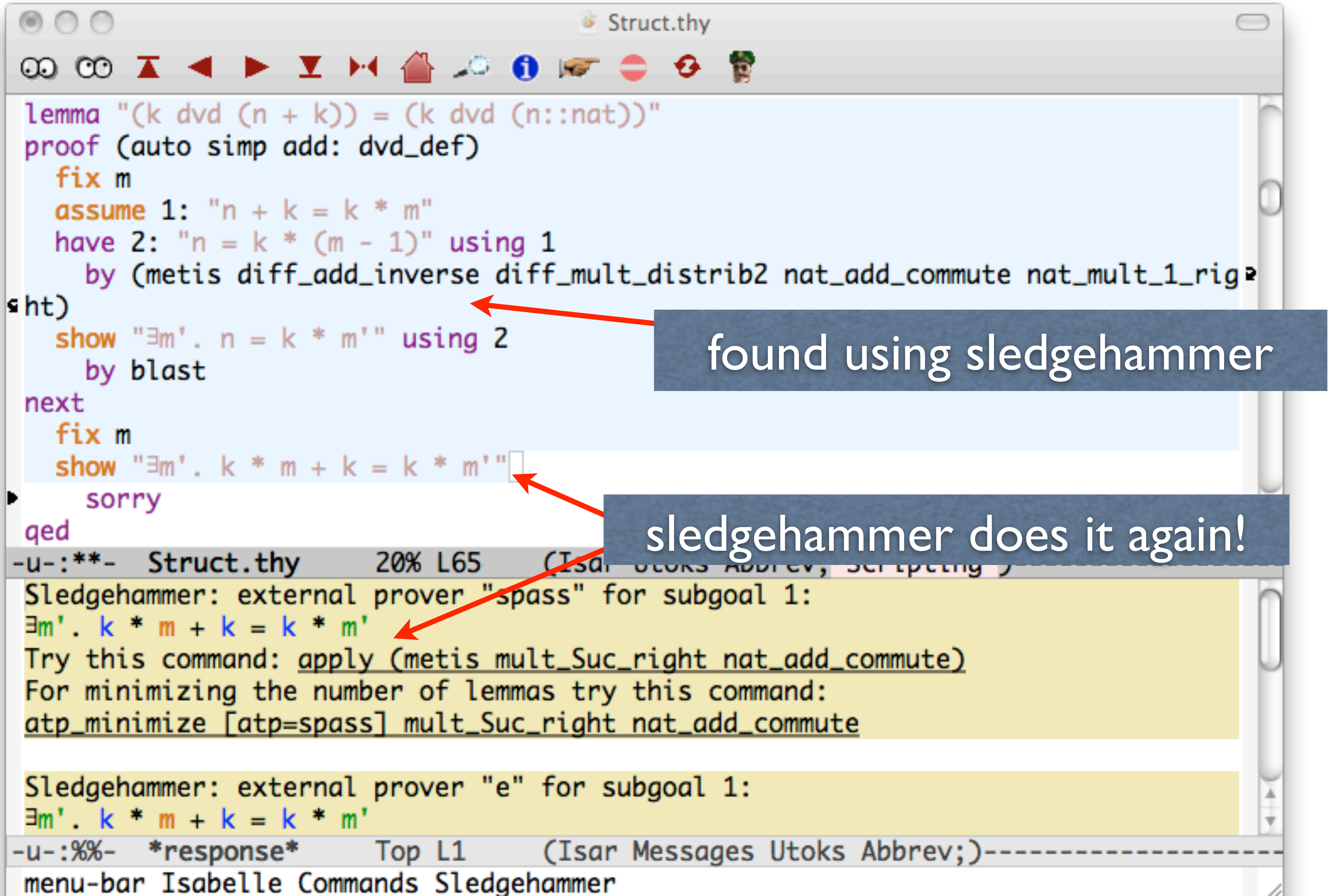
Completing the Proof

```
Struct.thy

lemma "(k dvd (n + k)) = (k dvd (n::nat))"
proof (auto simp add: dvd_def)
  fix m
  assume 1: "n + k = k * m"
  have 2: "n = k * (m - 1)" using 1
    by (metis diff_add_inverse diff_mult_distrib2 nat_add_commute nat_mult_1_rig
ht)
  show "∃m'. n = k * m'" using 2
    by blast
next
  fix m
  show "∃m'. k * m + k = k * m'"
    sorry
qed
-u-:**-  Struct.thy     20% L65     (Isar Utoks Abbrev; Scripting )-----------------
Sledgehammer: external prover "spass" for subgoal 1:
∃m'. k * m + k = k * m'
Try this command: apply (metis mult_Suc_right nat_add_commute)
For minimizing the number of lemmas try this command:
atp_minimize [atp=spass] mult_Suc_right nat_add_commute

Sledgehammer: external prover "e" for subgoal 1:
∃m'. k * m + k = k * m'
-u-:%%-  *response*     Top L1      (Isar Messages Utoks Abbrev;)-------------------
menu-bar Isabelle Commands Sledgehammer
```

Completing the Proof



Completing the Proof



Streamlining the Proof

```
assume 1: "n + k = k * m"
have 2: "n = k * (m - 1)" using 1
  by (metis diff_add_inverse diff
show "∃m'. n = k * m'" using 2
```

Streamlining the Proof

```
assume 1: "n + k = k * m"                     →  assume "n + k = k * m"
have 2: "n = k * (m - 1)" using 1             →  hence "n = k * (m - 1)"
  by (metis diff_add_inverse diff                  by (metis diff_add_inverse diff
show "∃m'. n = k * m'" using 2                →  thus "∃m'. n = k * m'"
```

Streamlining the Proof

```
assume 1: "n + k = k * m"
have 2: "n = k * (m - 1)" using 1
  by (metis diff_add_inverse diff
show "∃m'. n = k * m'" using 2
```

```
assume "n + k = k * m"
hence "n = k * (m - 1)"
  by (metis diff_add_inverse diff
thus "∃m'. n = k * m'"
```

- `hence` means `have` — `using` the previous fact

Streamlining the Proof

```
assume 1: "n + k = k * m"
have 2: "n = k * (m - 1)" using 1
  by (metis diff_add_inverse diff
show "∃m'. n = k * m'" using 2
```

→

```
assume "n + k = k * m"
hence "n = k * (m - 1)"
  by (metis diff_add_inverse diff
thus "∃m'. n = k * m'"
```

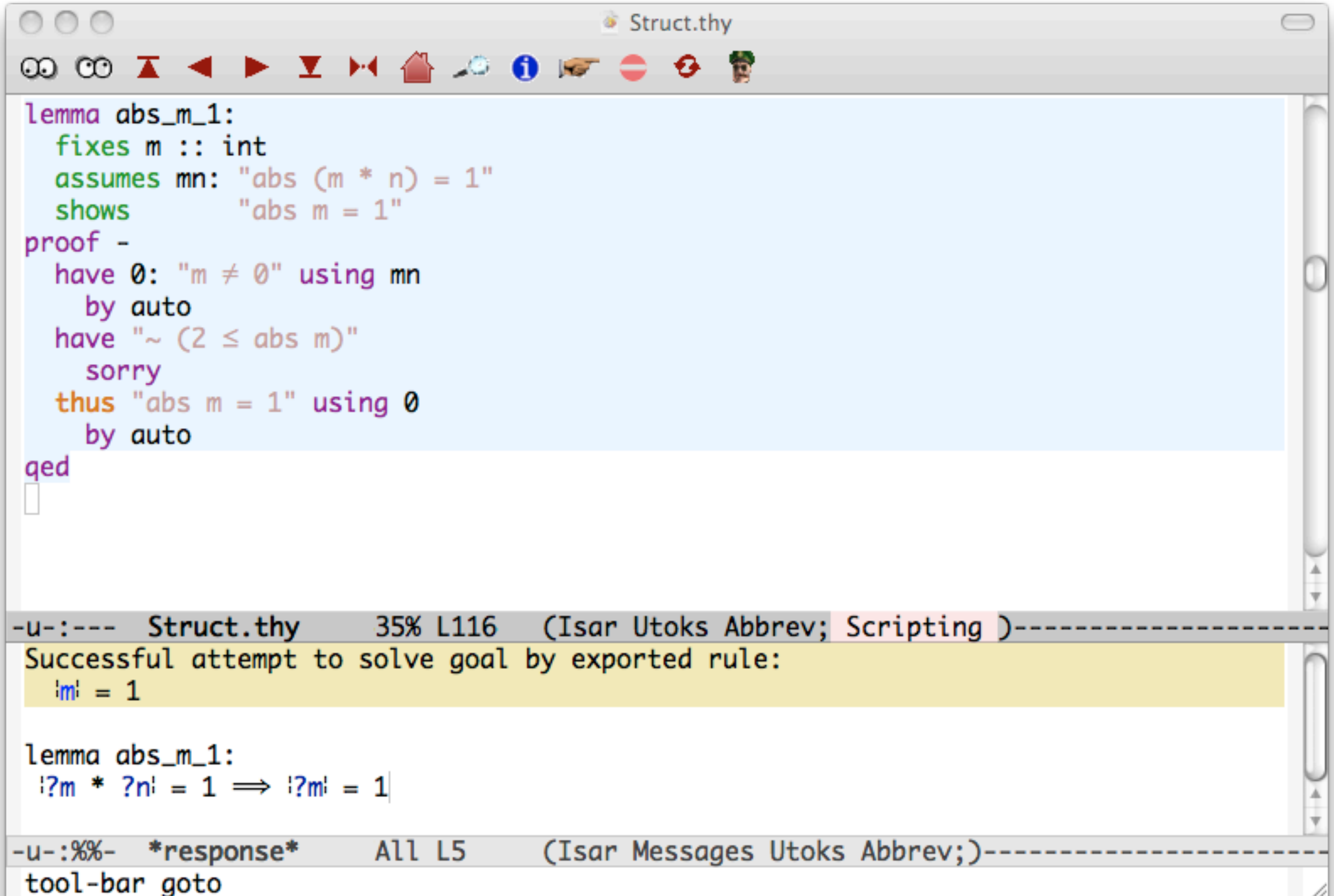
- `hence` means `have` — `using` the previous fact
- `thus` means `show` — `using` the previous fact

Streamlining the Proof

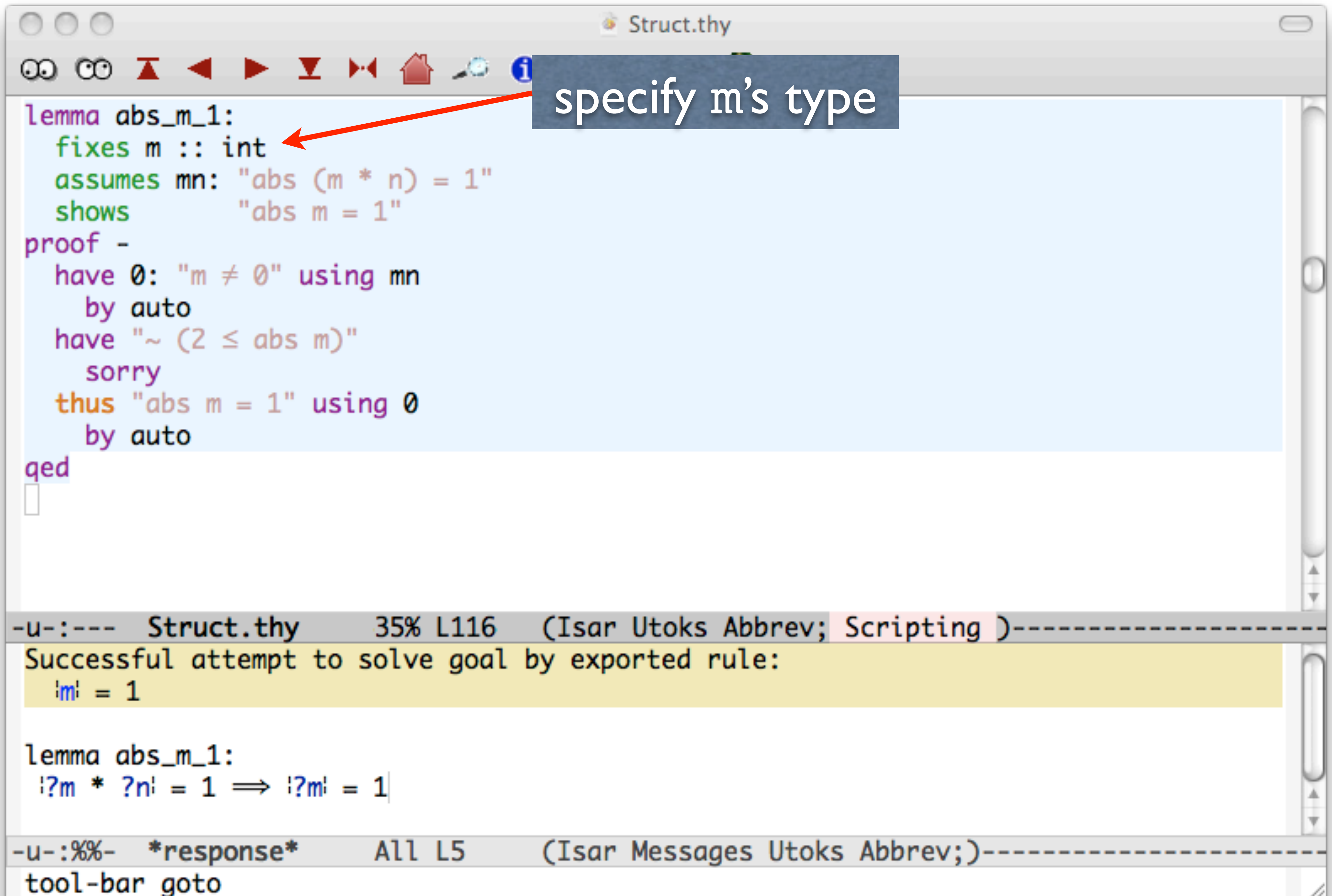
```
assume 1: "n + k = k * m"                 ⟶  assume "n + k = k * m"
have 2: "n = k * (m - 1)" using 1         ⟶  hence "n = k * (m - 1)"
  by (metis diff_add_inverse diff               by (metis diff_add_inverse diff
show "∃m'. n = k * m'" using 2            ⟶  thus "∃m'. n = k * m'"
```

- `hence` means `have` — `using` the previous fact
- `thus` means `show` — `using` the previous fact
- There are numerous other tricks of this sort!

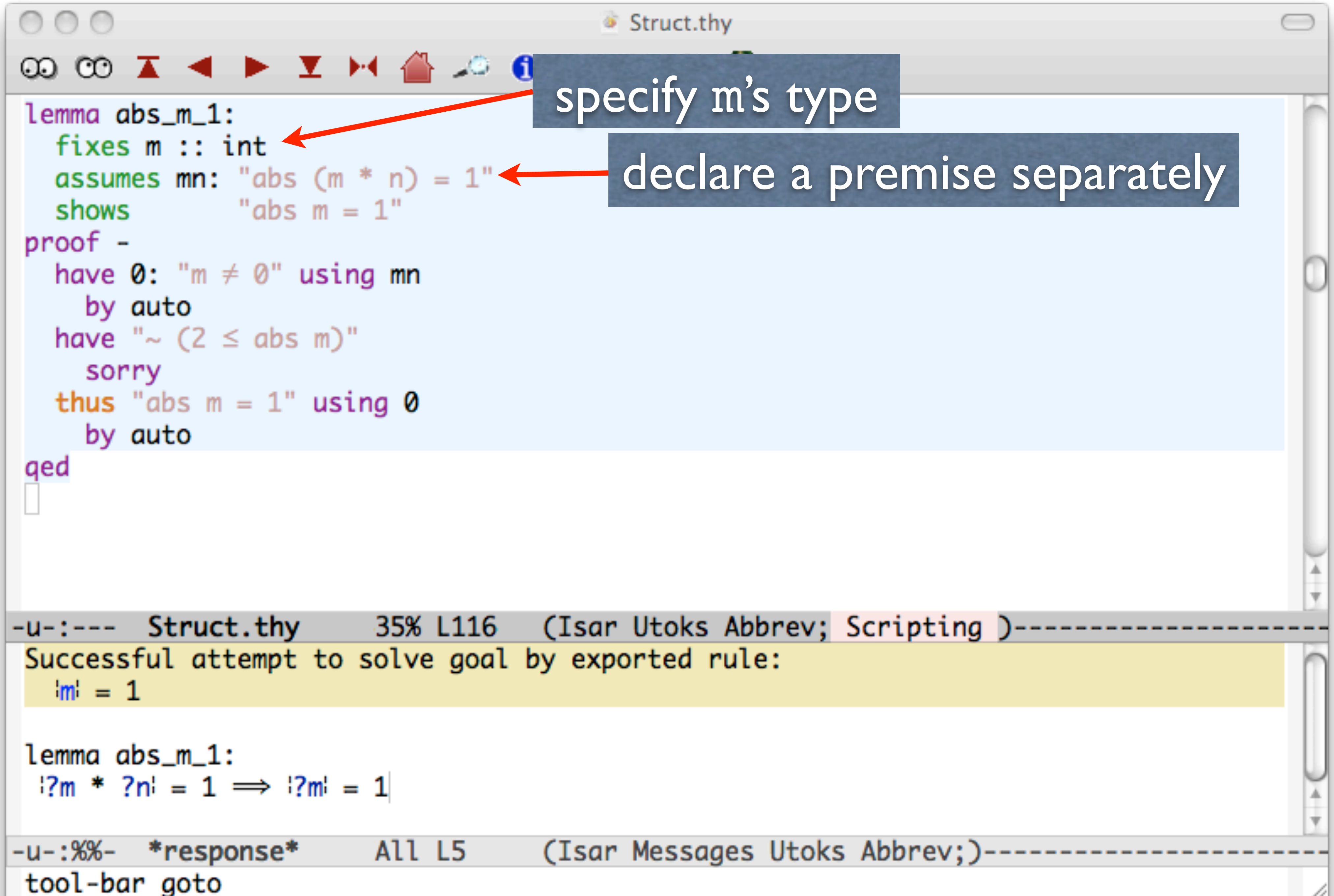
Another Proof Skeleton



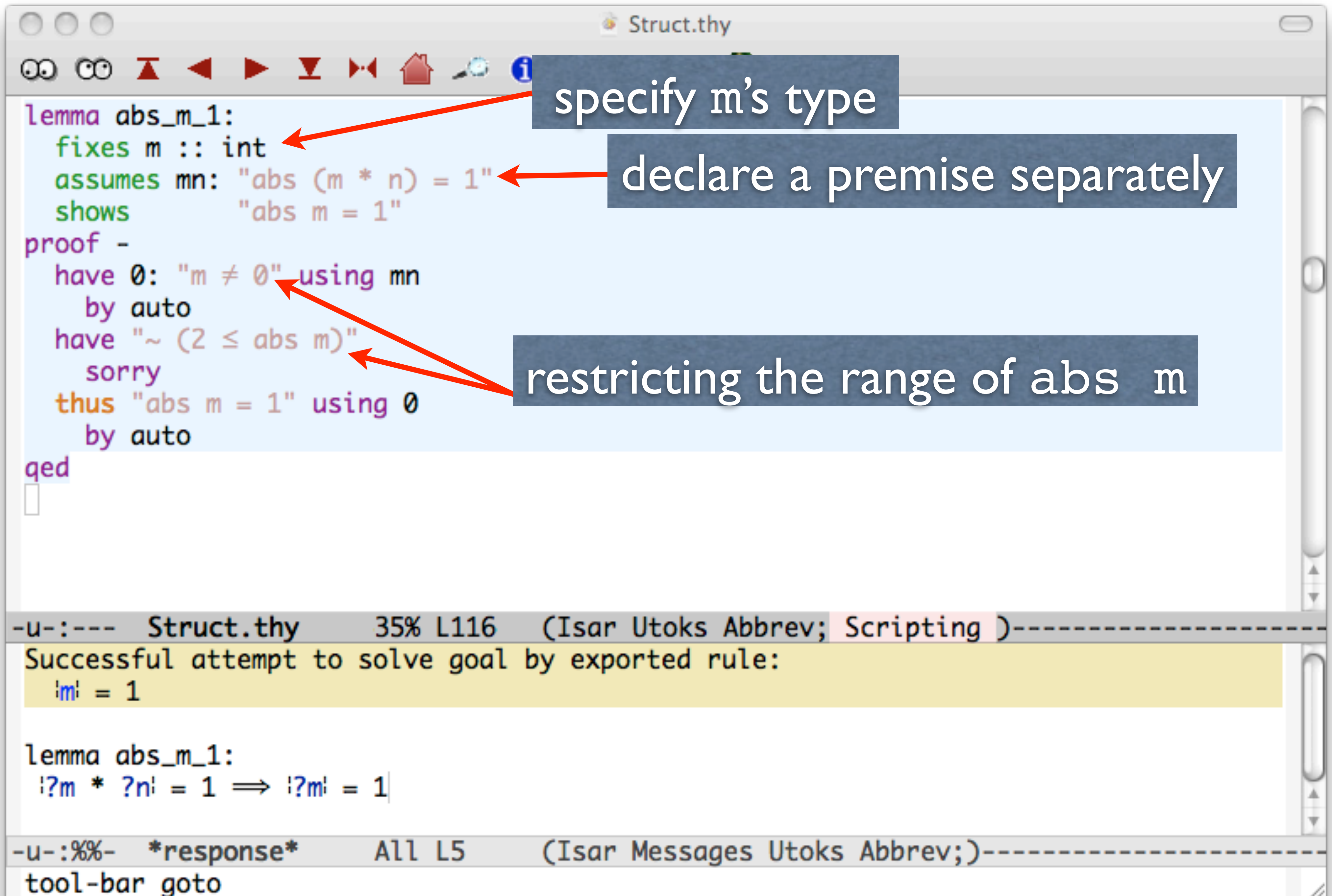
Another Proof Skeleton



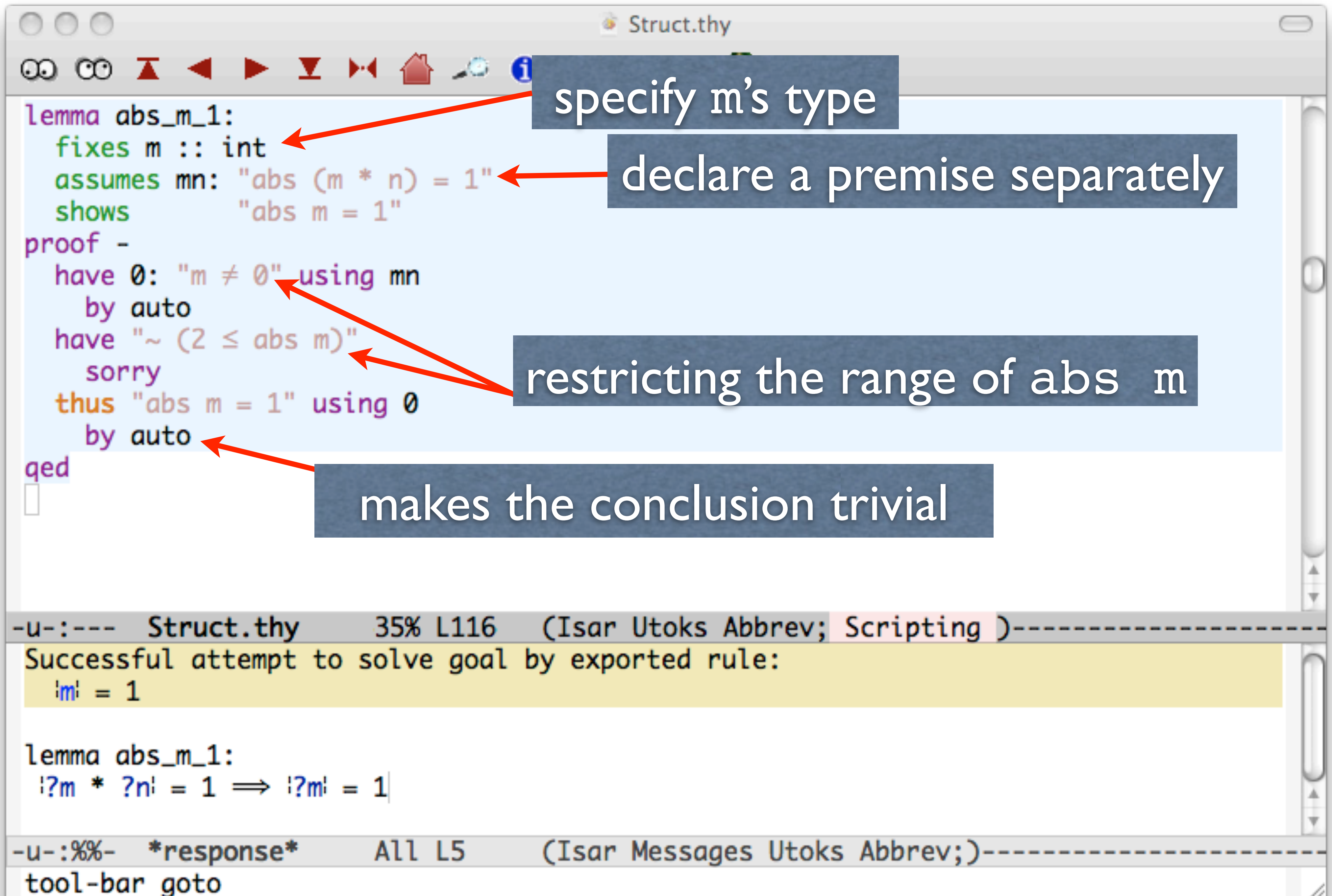
Another Proof Skeleton



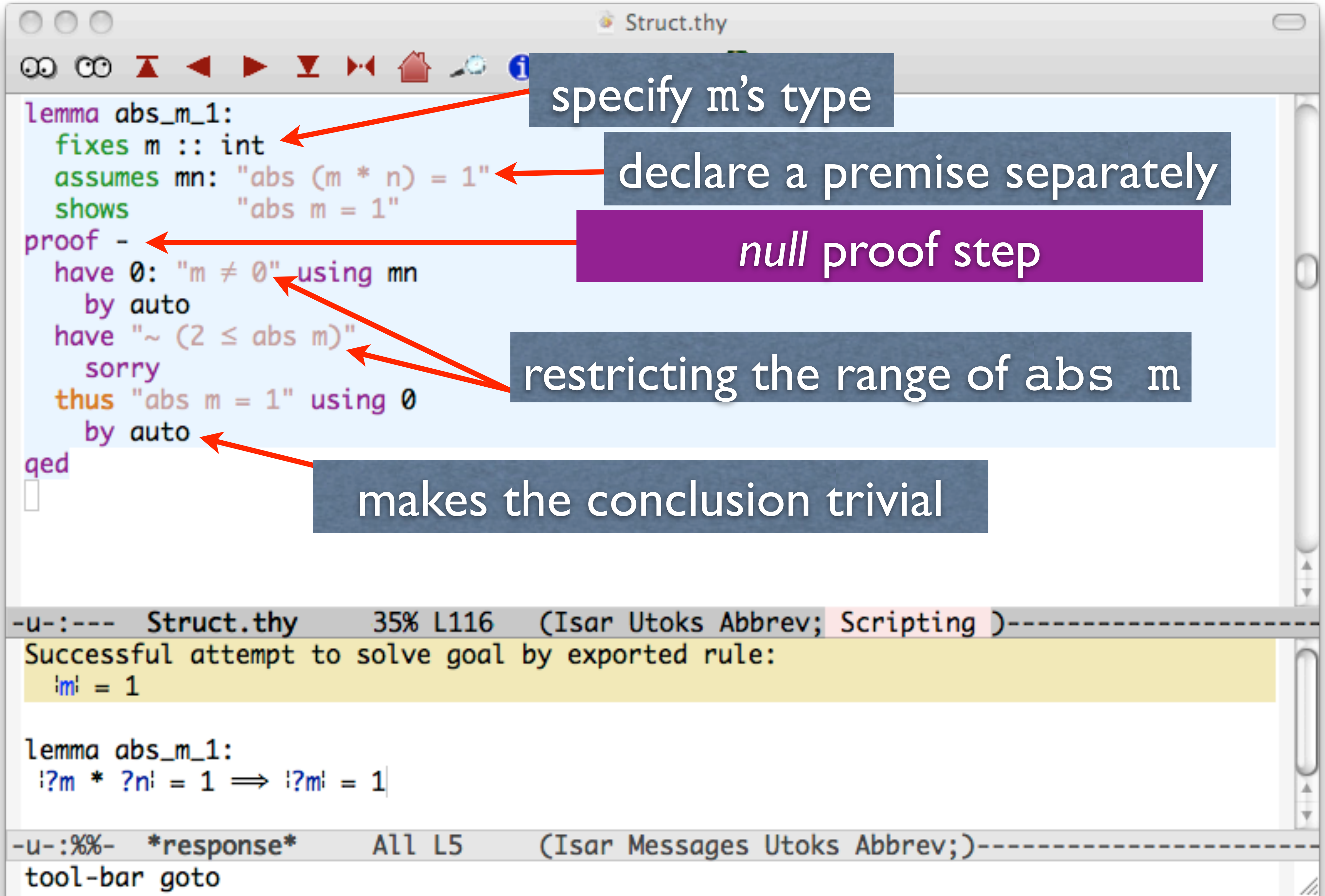
Another Proof Skeleton



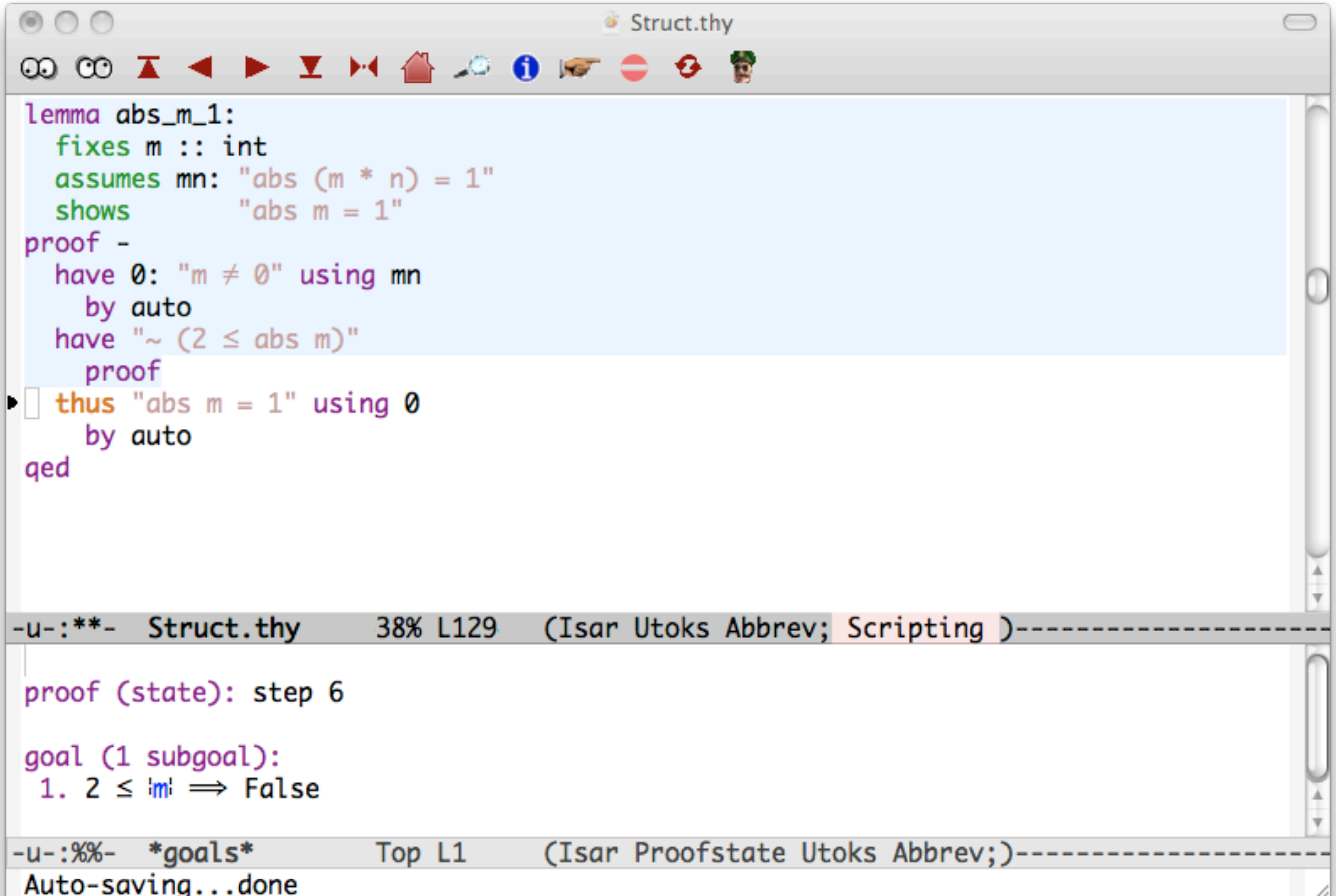
Another Proof Skeleton



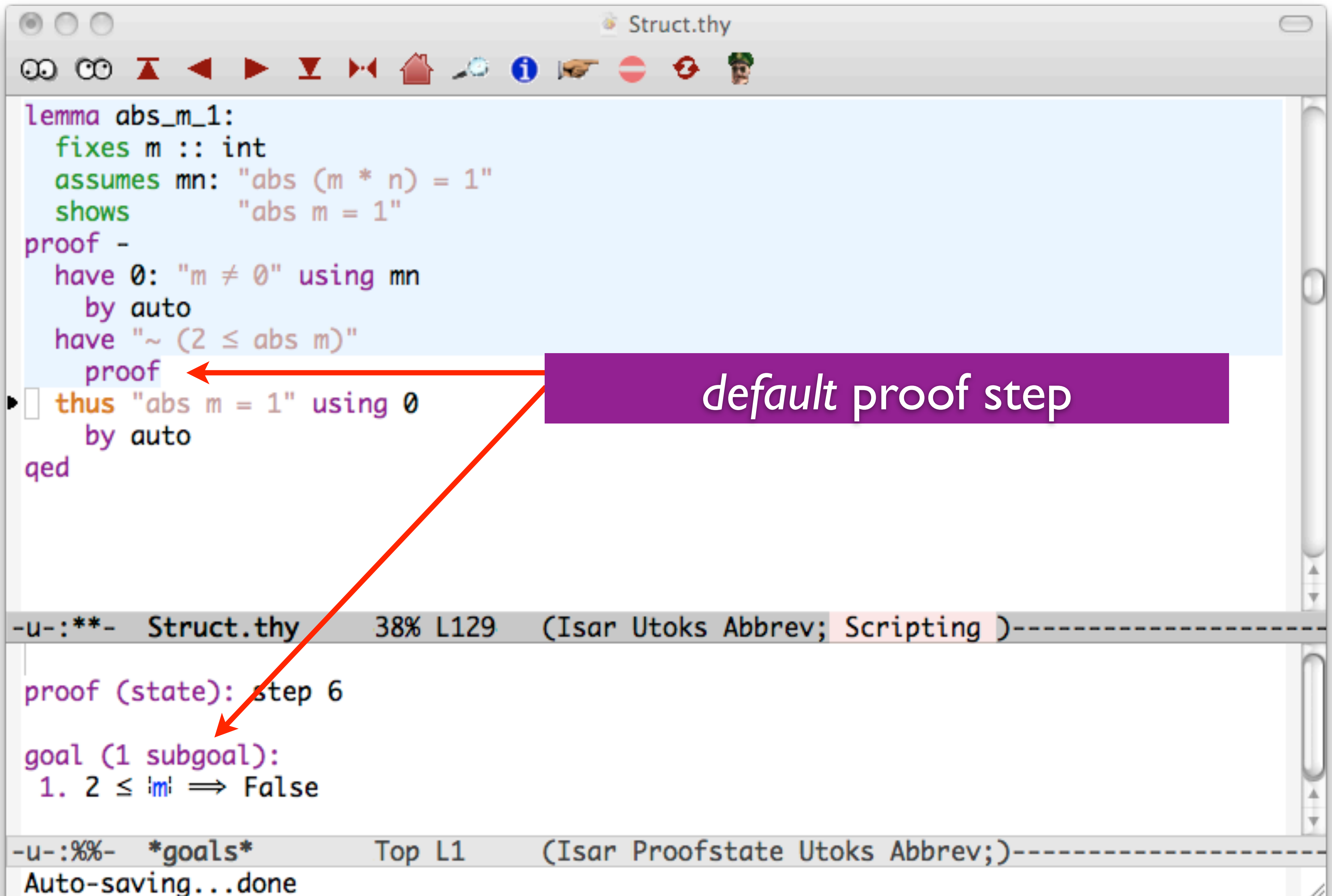
Another Proof Skeleton



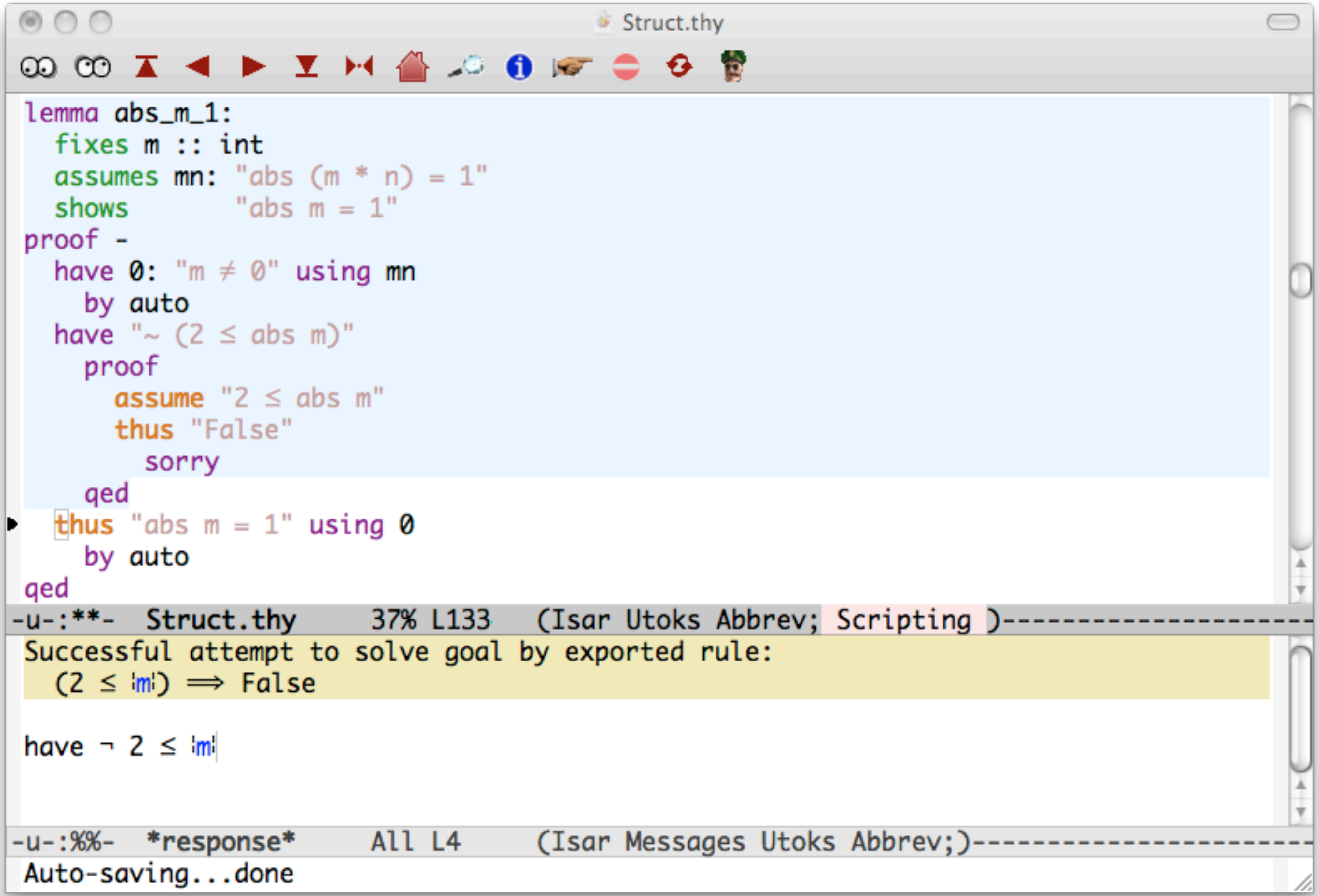
Starting a Nested Proof



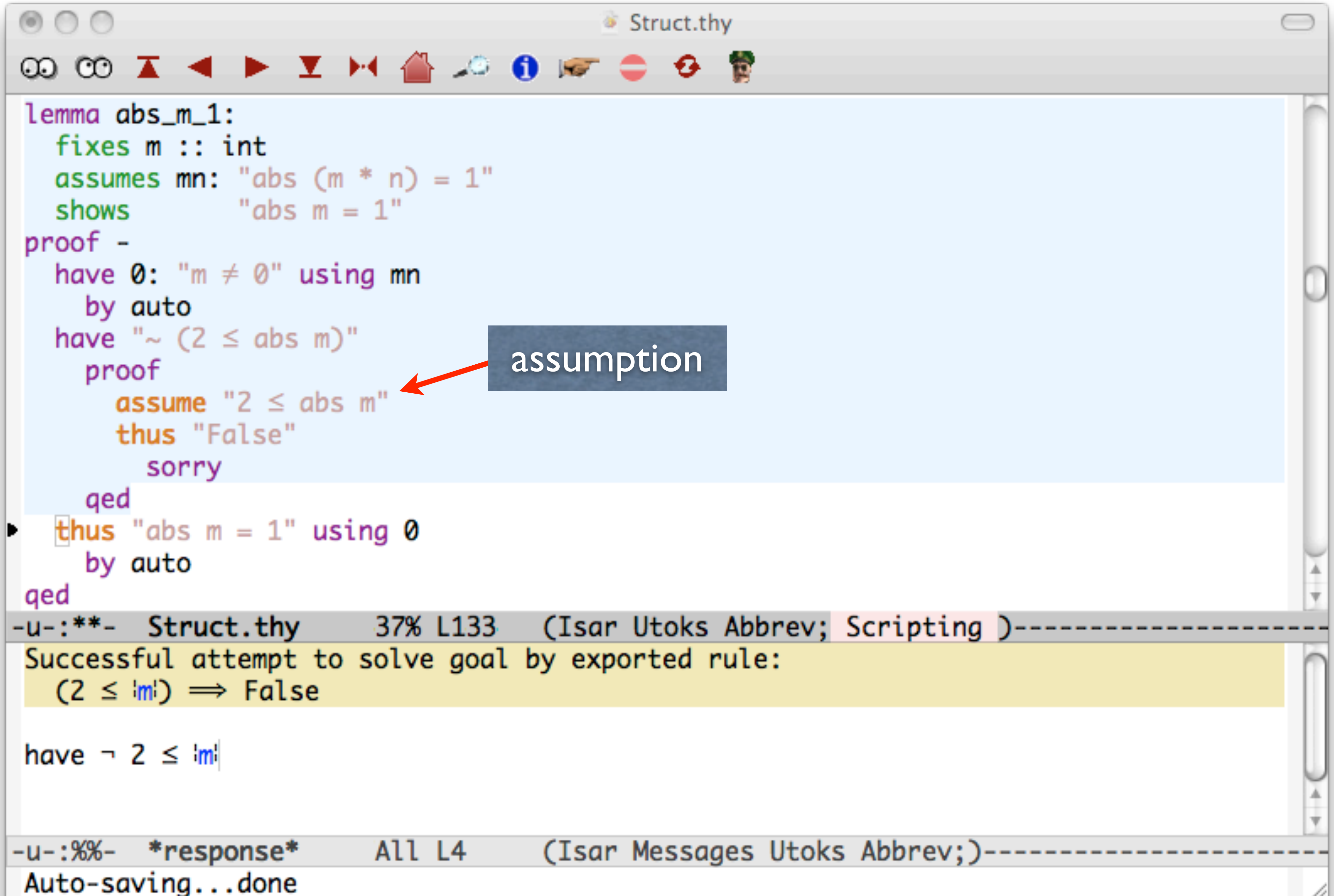
Starting a Nested Proof



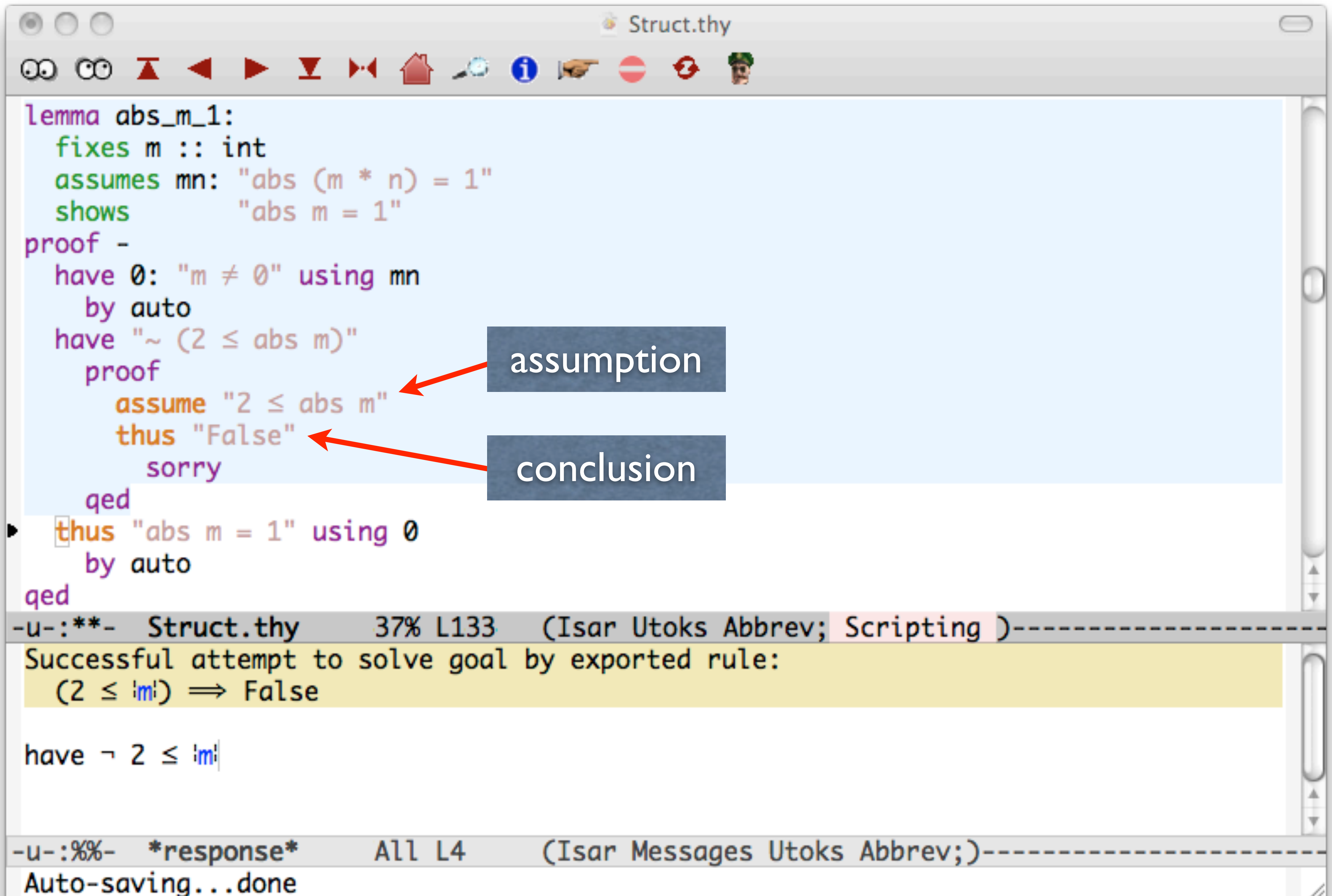
A Nested Proof Skeleton



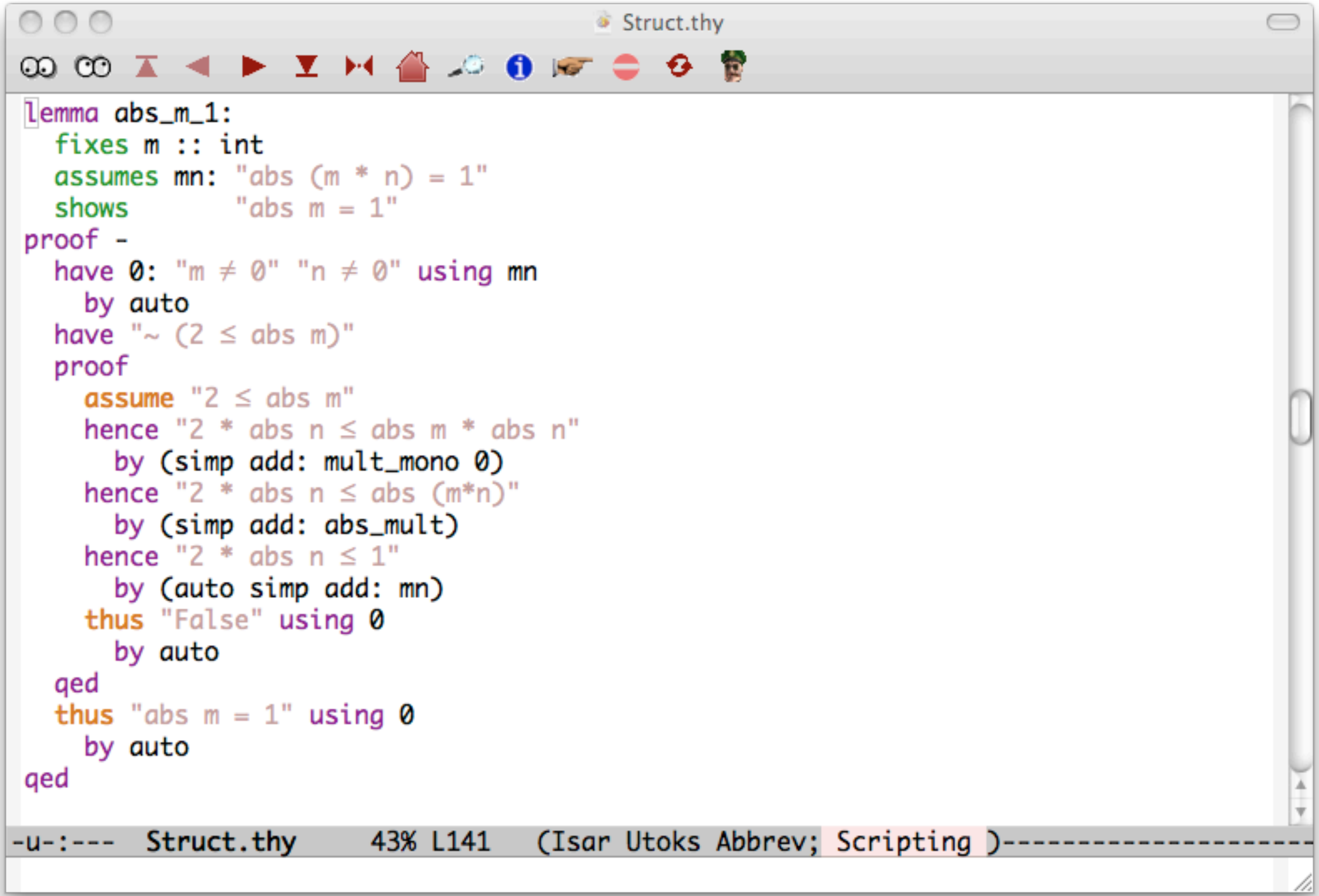
A Nested Proof Skeleton



A Nested Proof Skeleton



A Complete Proof



A Complete Proof

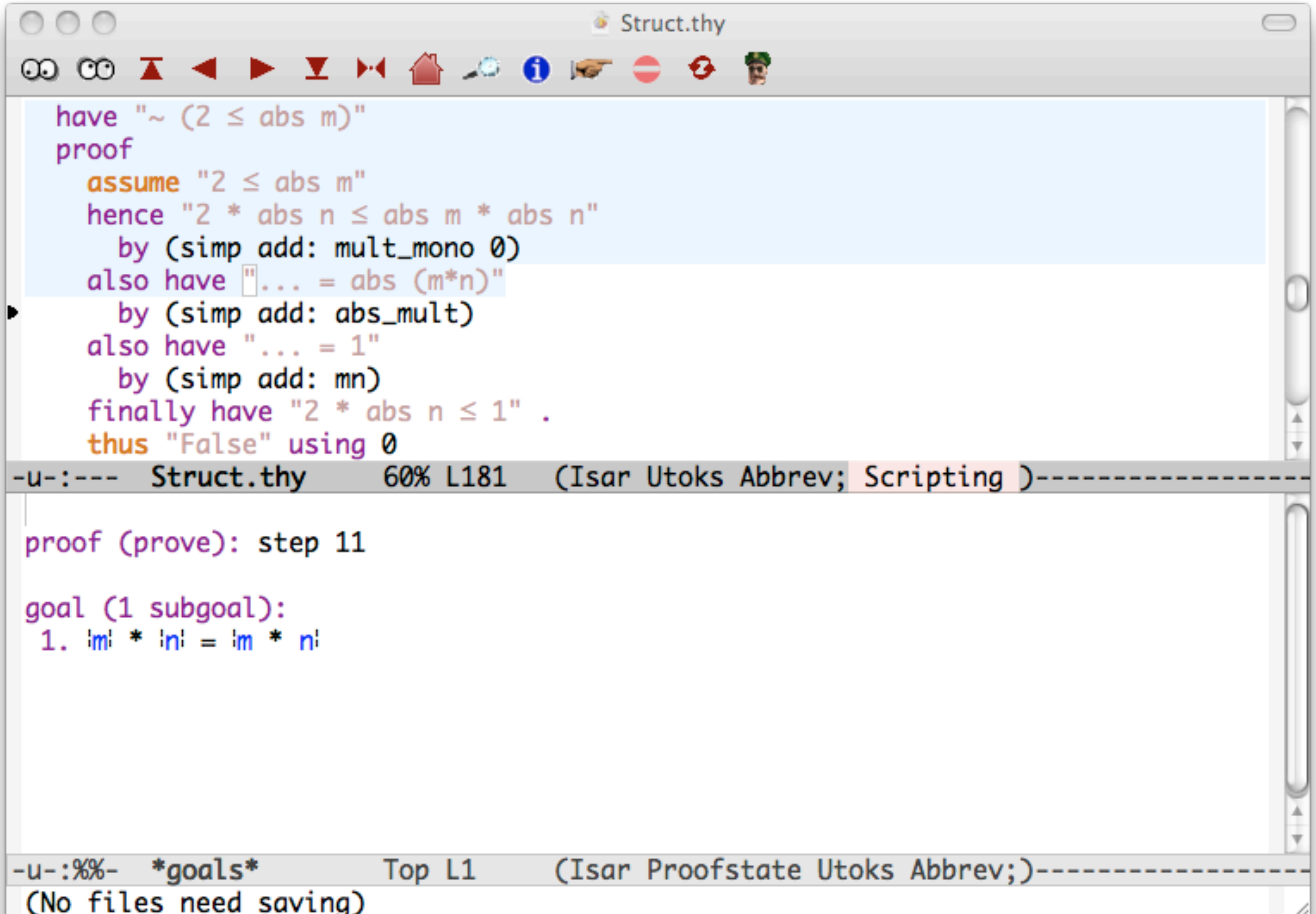
Struct.thy

```
lemma abs_m_1:
  fixes m :: int
  assumes mn: "abs (m * n) = 1"
  shows       "abs m = 1"
proof -
  have 0: "m ≠ 0" "n ≠ 0" using mn
    by auto
  have "~ (2 ≤ abs m)"
  proof
    assume "2 ≤ abs m"
    hence "2 * abs n ≤ abs m * abs n"
      by (simp add: mult_mono 0)
    hence "2 * abs n ≤ abs (m*n)"
      by (simp add: abs_mult)
    hence "2 * abs n ≤ 1"
      by (auto simp add: mn)
    thus "False" using 0
      by auto
  qed
  thus "abs m = 1" using 0
    by auto
qed
```

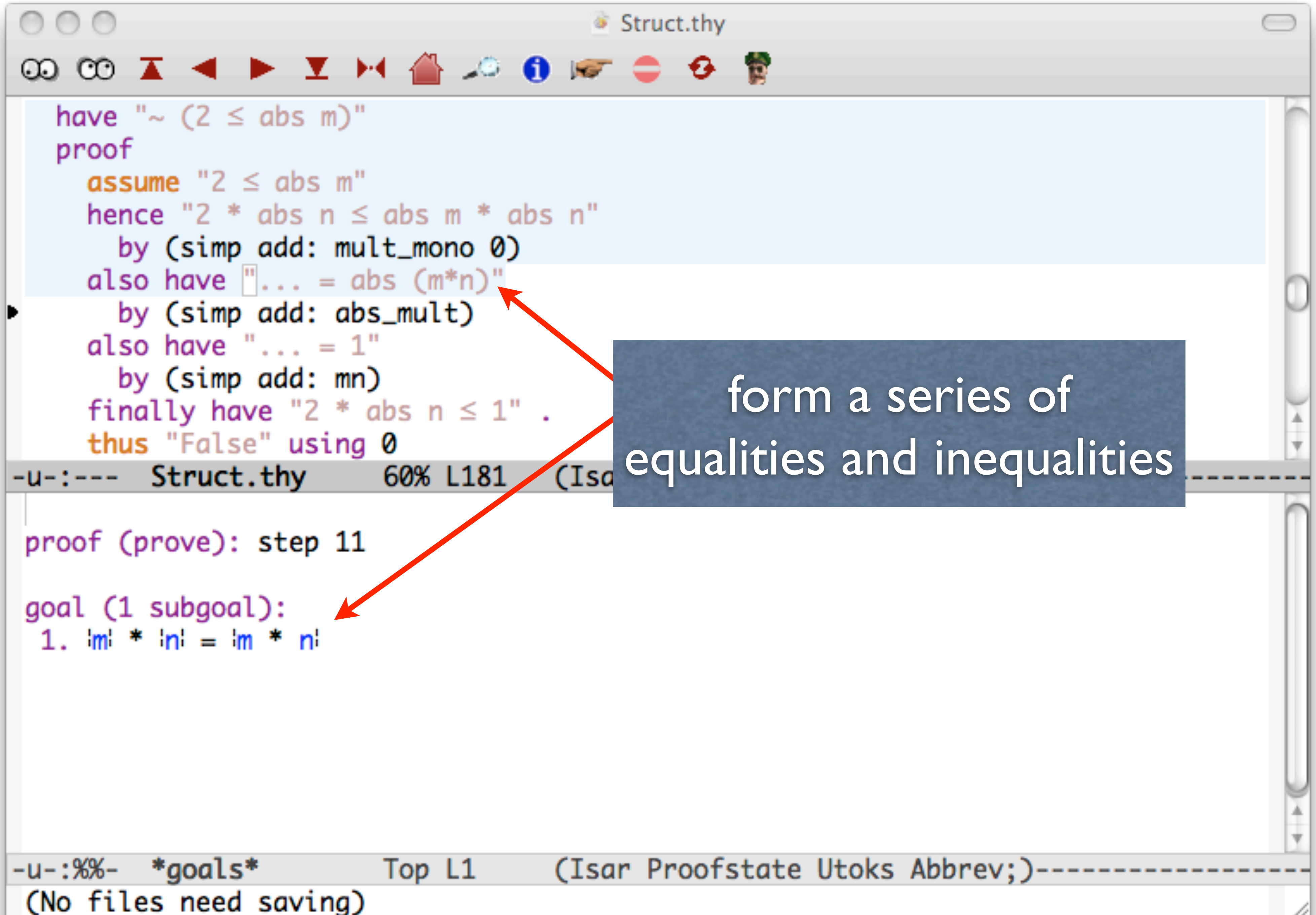
-u-:--- Struct.thy 43% L141 (Isar Utoks Abbrev; Scripting)--------------------

a chain of steps leads to contradiction

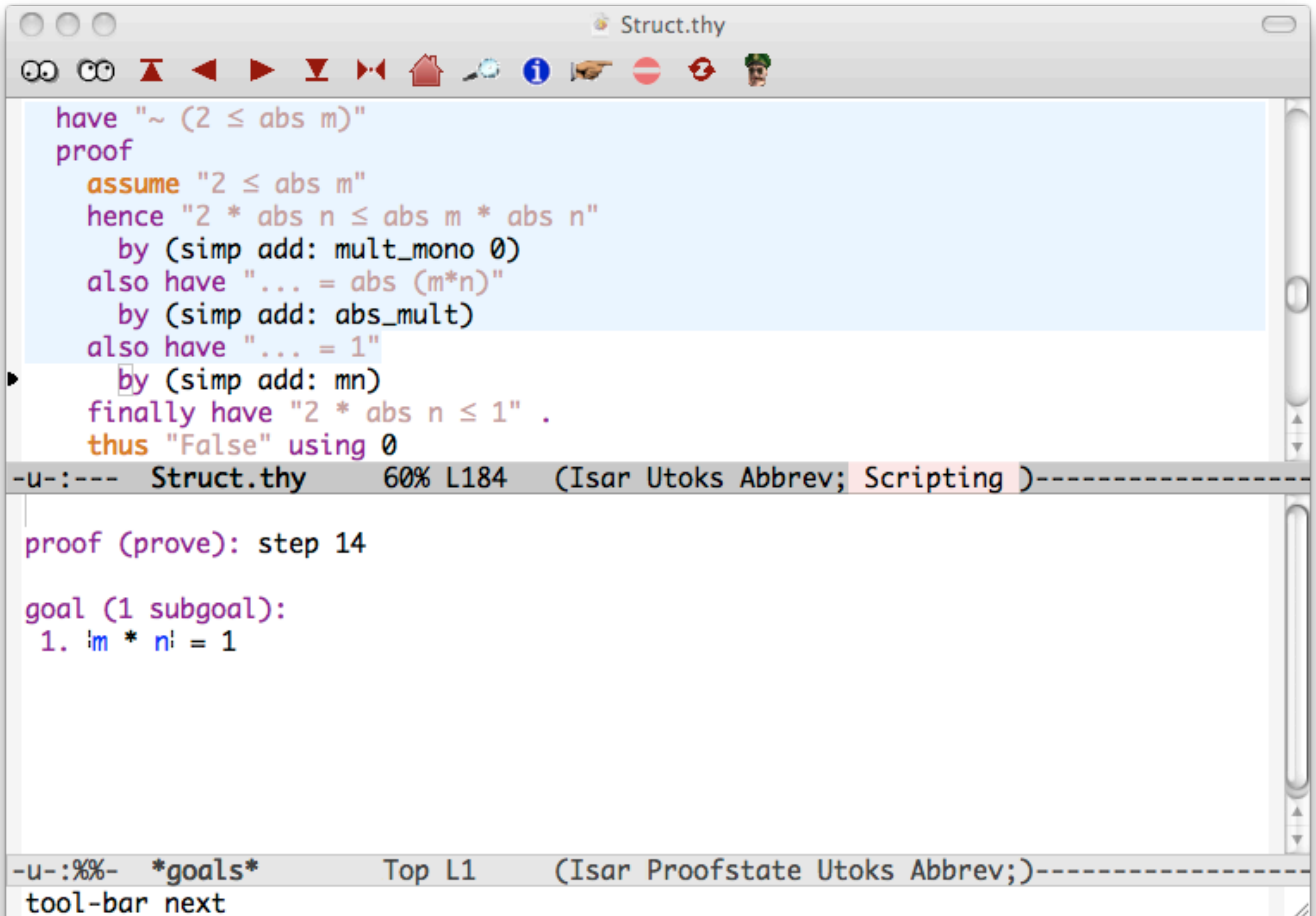
Calculational Proofs



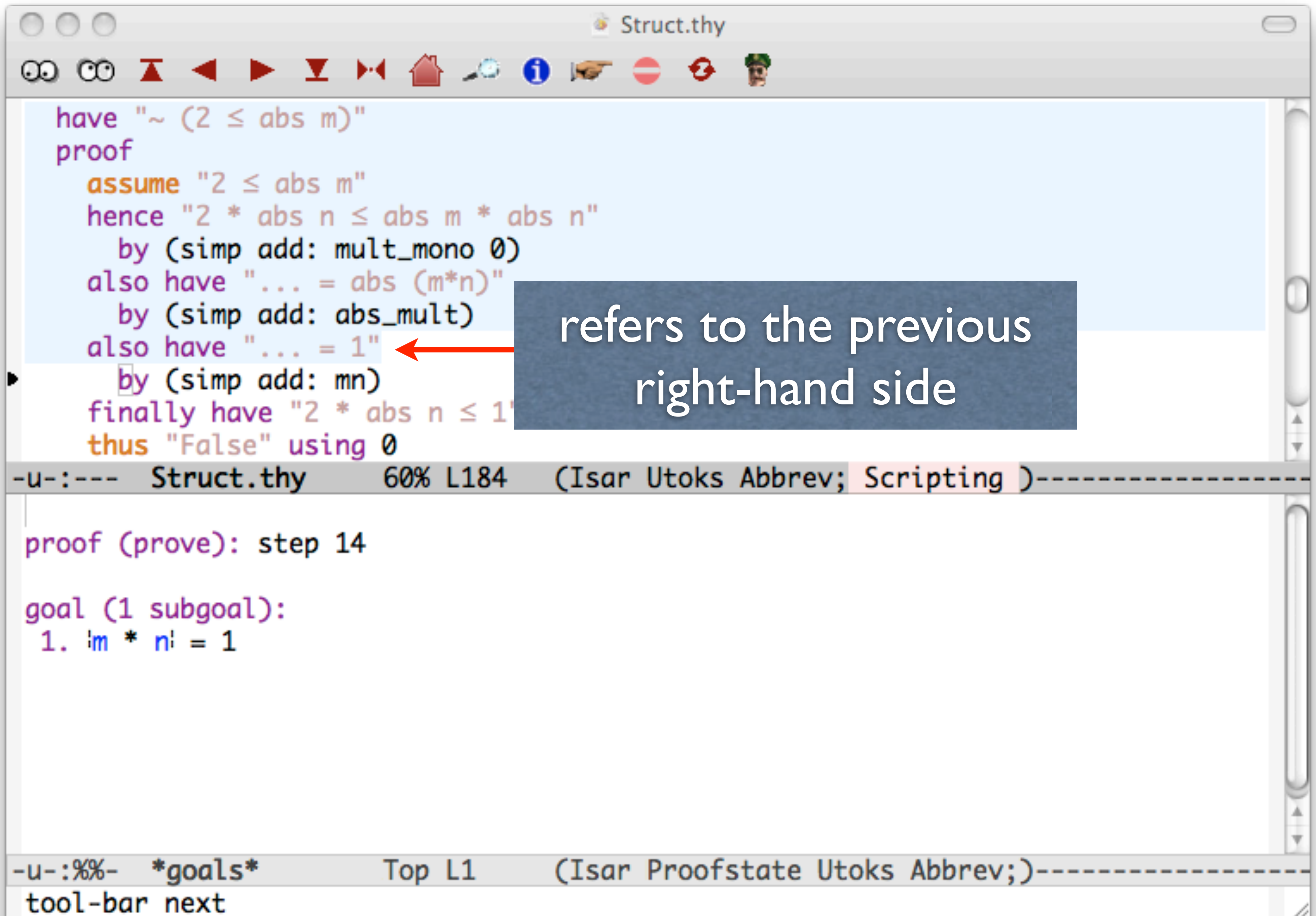
Calculational Proofs



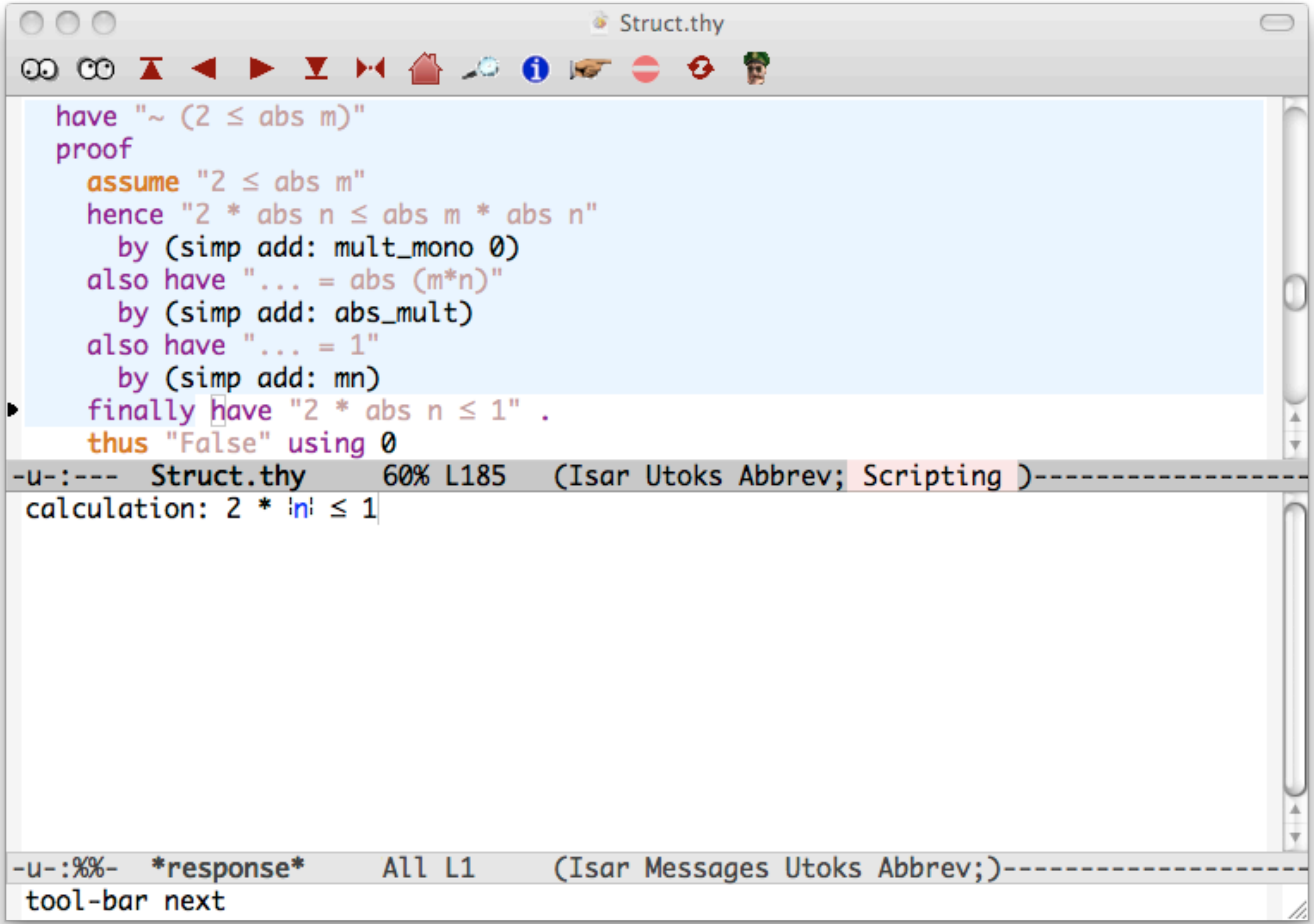
The Next Step



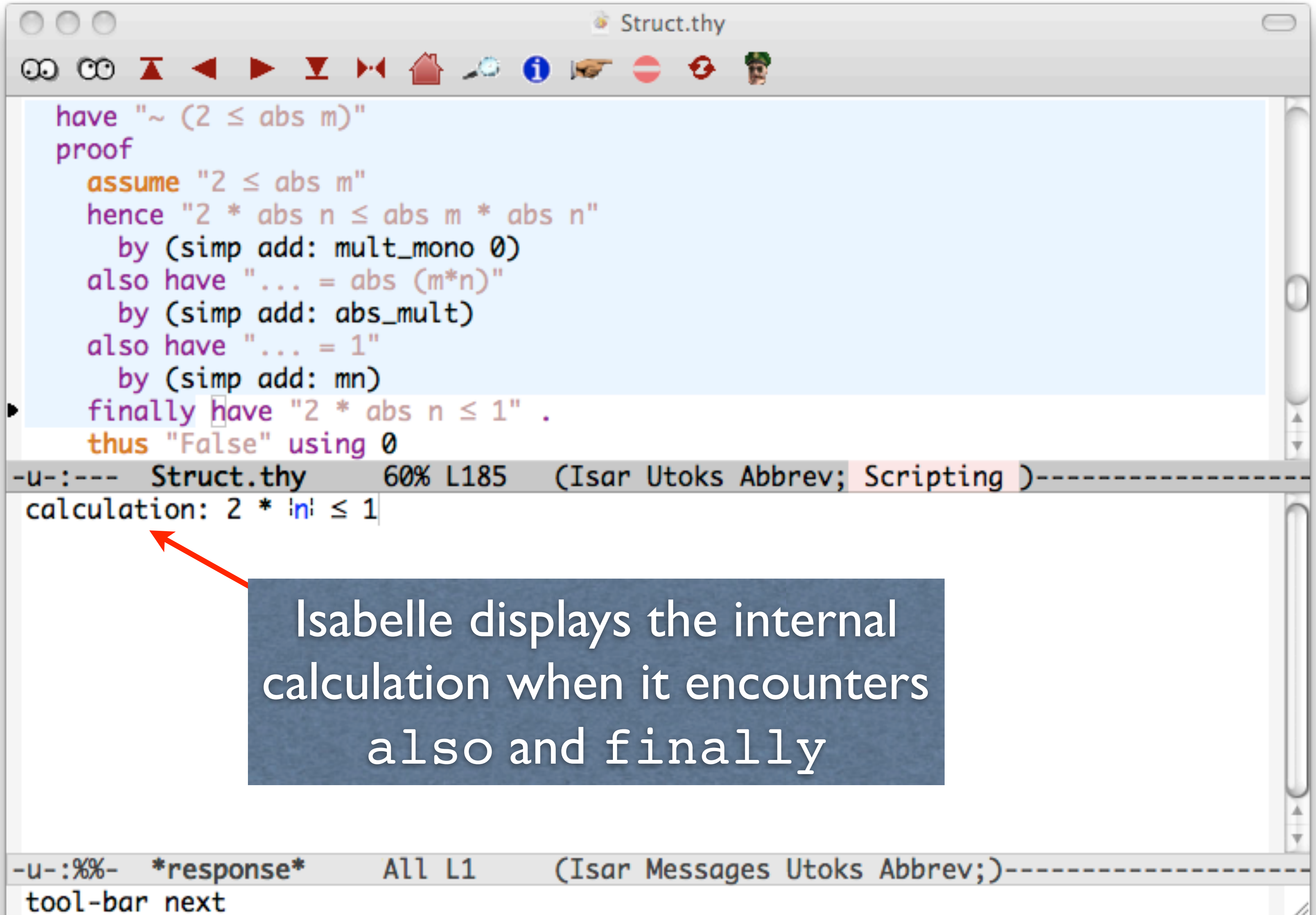
The Next Step



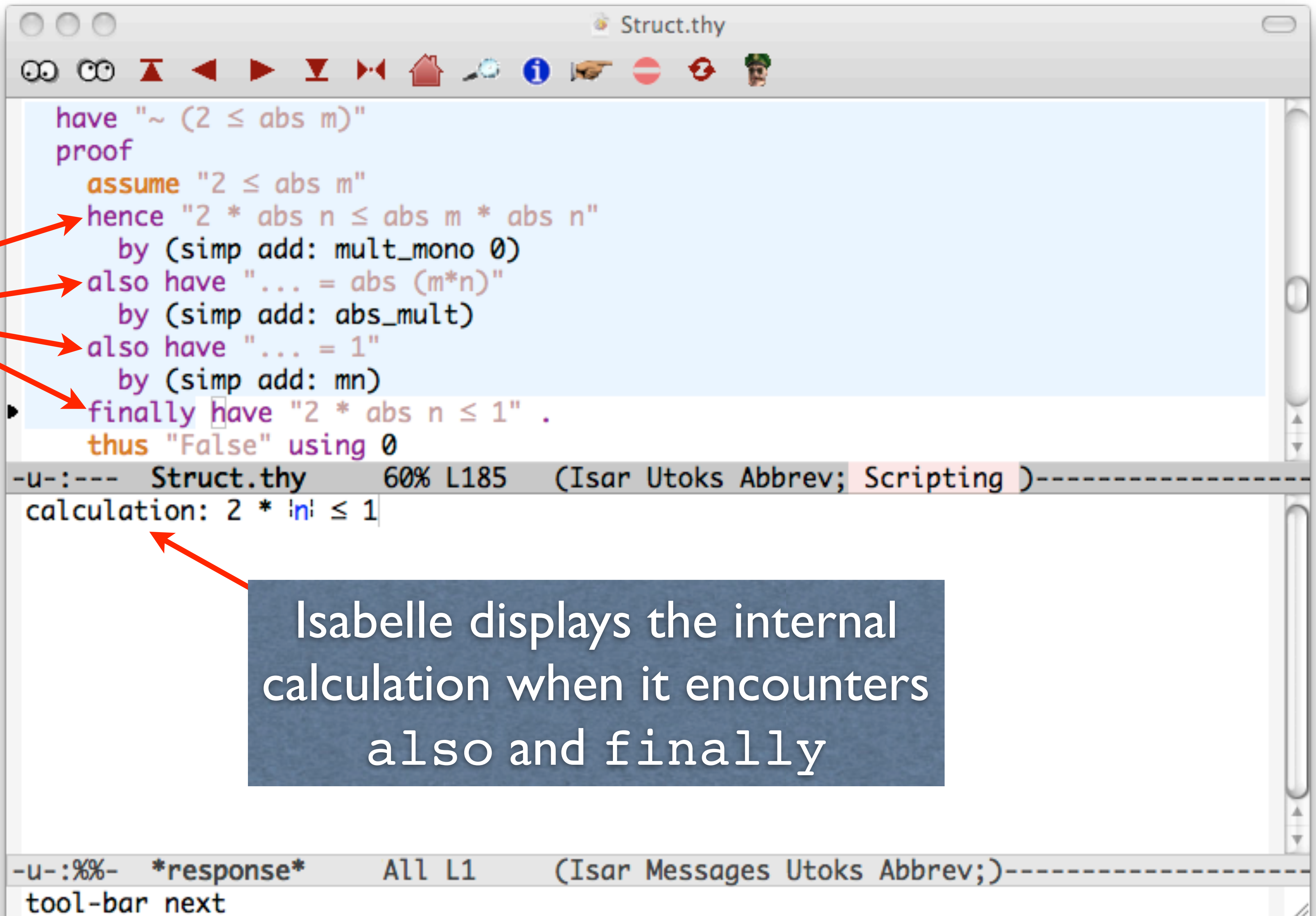
The Internal Calculation



The Internal Calculation



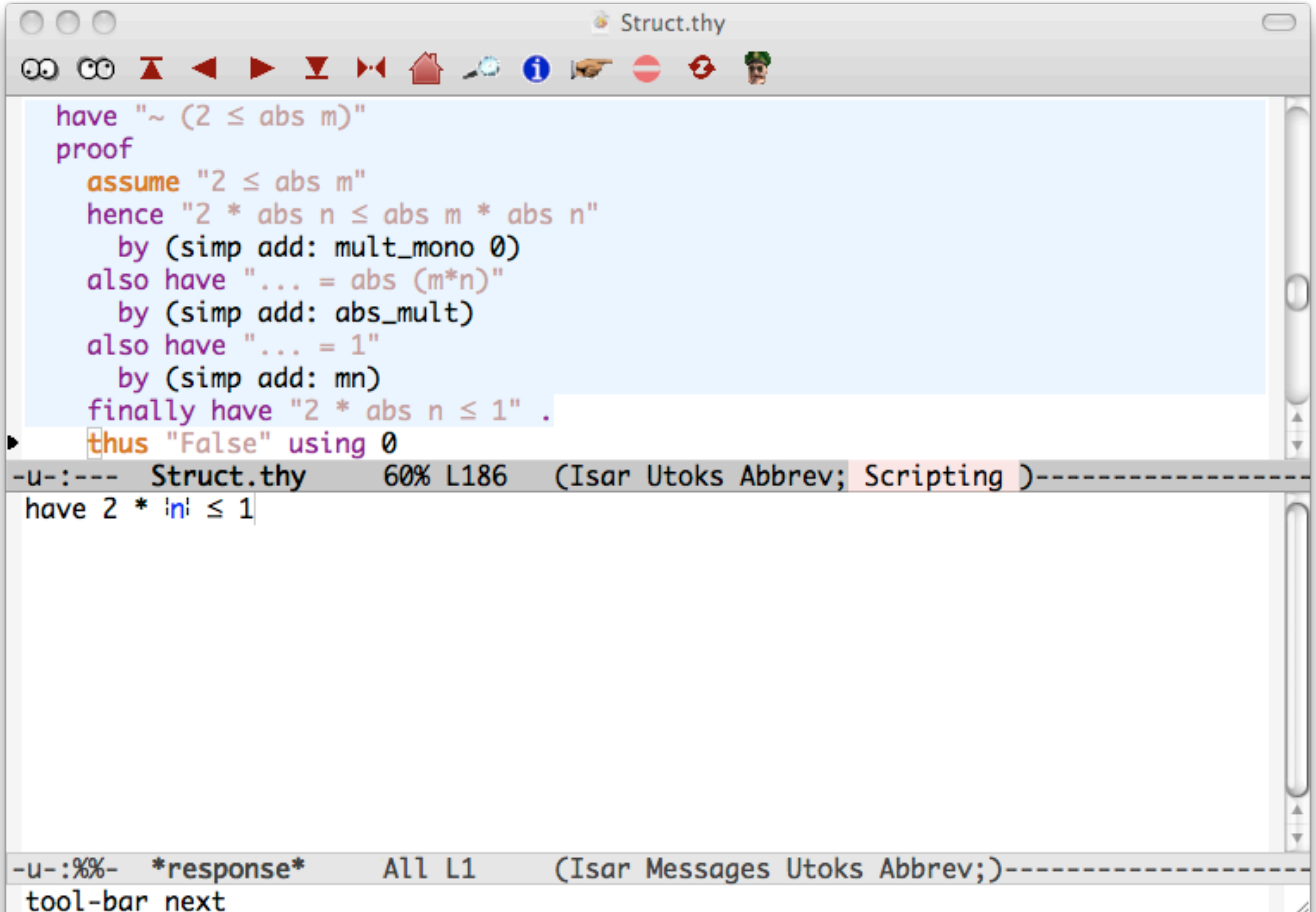
The Internal Calculation



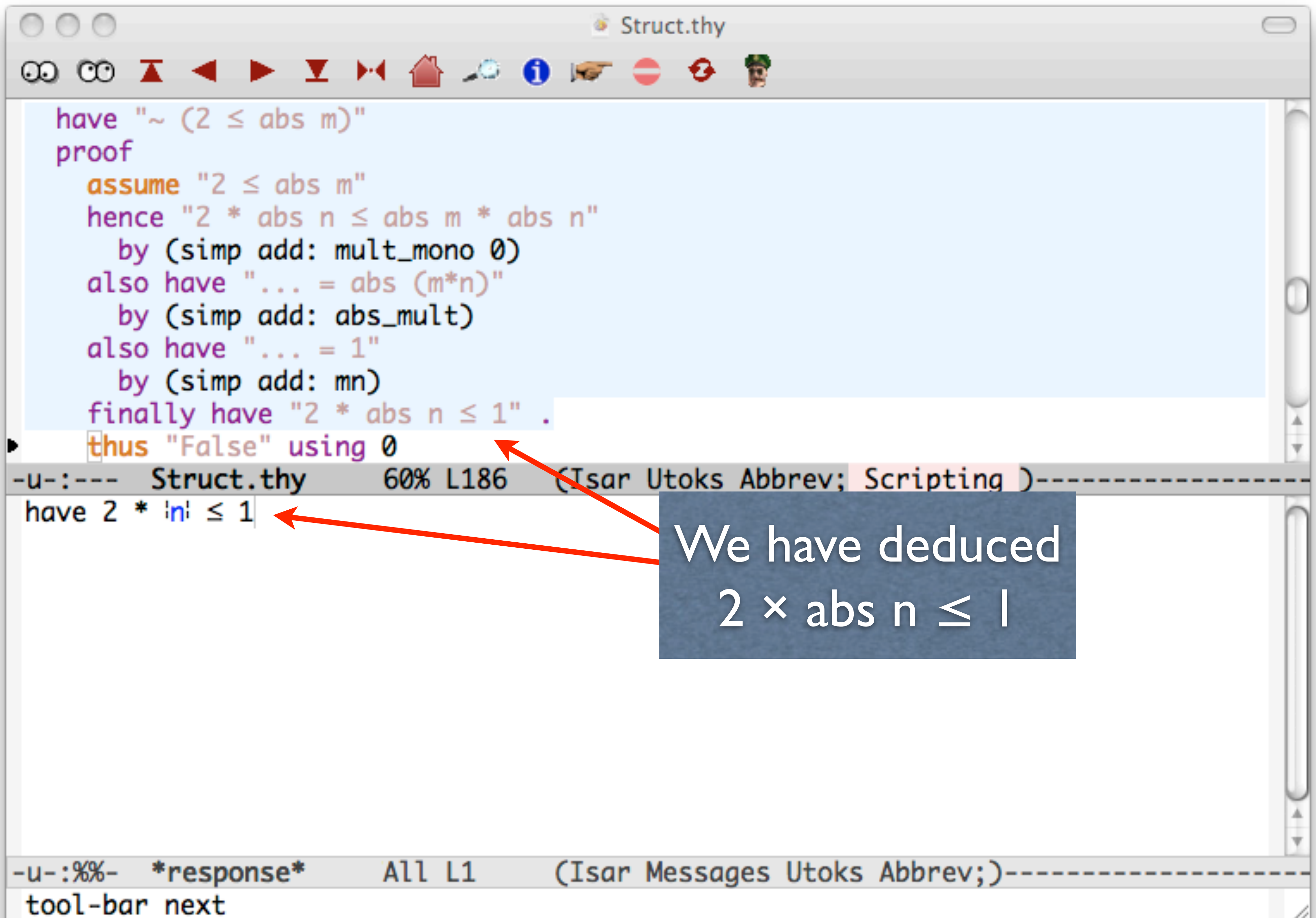
structure of a calculation

Isabelle displays the internal calculation when it encounters `also` and `finally`

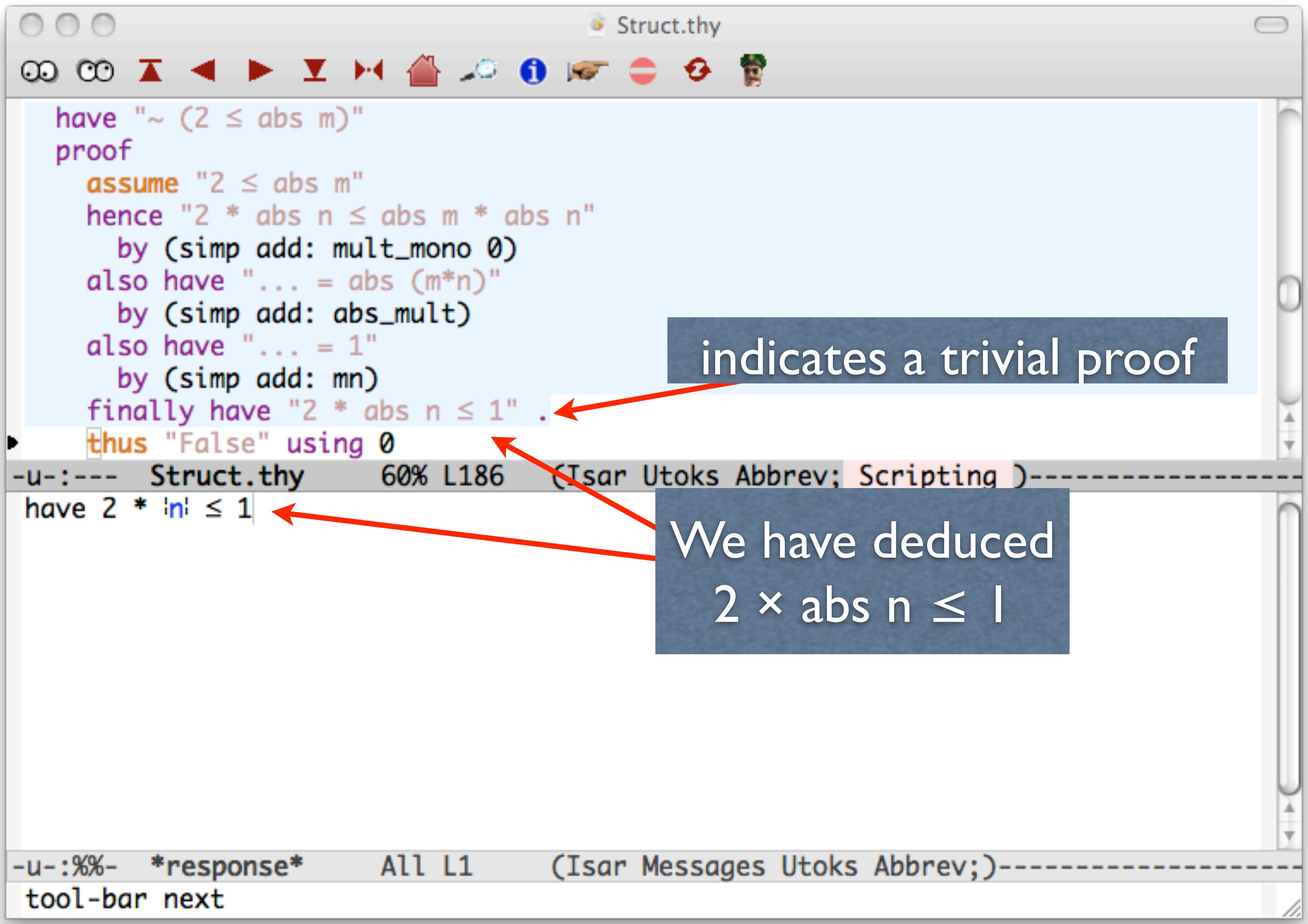
Ending the Calculation



Ending the Calculation



Ending the Calculation



Structure of a Calculation

Structure of a Calculation

- The first line is `have`/`hence`

Structure of a Calculation

- The first line is `have/hence`
- Subsequent lines begin, `also have “... = “`

Structure of a Calculation

- The first line is `have/hence`
- Subsequent lines begin, `also have “... = “`
- *Any* transitive relation may be used. New ones may be declared.

Structure of a Calculation

- The first line is `have/hence`
- Subsequent lines begin, `also have “... = “`
- *Any* transitive relation may be used. New ones may be declared.
- The concluding line begins, `finally have/show`, repeats the calculation and terminates with `(.)`