Interactive Formal Verification

Lawrence C Paulson
Computer Laboratory
University of Cambridge

This lecture course introduces interactive formal proof using Isabelle. The lecture notes consist of copies of the slides, some of which have brief remarks attached. Isabelle documentation can be found on the Internet at the URL http://www.cl.cam.ac.uk/research/hvg/Isabelle/documentation.html. The most important single manual is the *Tutorial on Isabelle/HOL*. Reading the *Tutorial* is an excellent way of learning Isabelle in depth. However, the *Tutorial* is a little outdated; although its details remain correct, it presents a style of proof that has become increasingly obsolete with the advent of structured proofs and ever greater automation. These lecture notes take a very different approach and refer you to specific sections of the *Tutorial* that are particularly appropriate.

The other tutorials listed on the documentation page are mainly for advanced users.

Interactive Formal Verification
1: Introduction

Lawrence C Paulson
Computer Laboratory
University of Cambridge

What is Interactive Proof?

- Work in a logical formalism
  - precise definitions of concepts
  - formal reasoning system
- Construct hierarchies of definitions and proofs
  - libraries of formal mathematics
  - specifications of components and properties

Interactive Theorem Provers

- Based on higher-order logic
  - Isabelle, HOL (many versions), PVS
- Based on constructive type theory
  - Coq, Twelf, Agda, ...
- Based on first-order logic with recursion
  - ACL2

Here are some useful web links:

```
Isabelle: http://www.cl.cam.ac.uk/research/hvg/Isabelle/
HOL4: http://hol.sourceforge.net/
HOL Light: http://www.cl.cam.ac.uk/~jrh13/hol-light/
PVS: http://pvs.csl.sri.com/
Coq: http://coq.inria.fr/
ACL2: http://www.cs.utexas.edu/users/moore/acl2/
```

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

$\leq \; \geq \; \neq \; \wedge \; \vee \; \rightarrow \; \leftrightarrow \; \forall \; \exists$

See the *Tutorial*, section 1.3: "Types, terms and formulae"

Some Syntactic Conventions

In $\forall x.\ \mathrm{A} \wedge B$, the quantifier spans the entire formula

Parentheses are **required** in $\mathrm{A} \wedge (\forall \mathrm{x}\ \mathrm{y}.\ B)$

Binary logical connectives associate to the right: $A \rightarrow B \rightarrow C$ is the same as $A \rightarrow (B \rightarrow C)$

$\neg A \wedge B = C \vee D$ is the same as $((\neg A) \wedge (B = C)) \vee D$

See the *Tutorial*, section 1.3: “Types, terms and formulae”

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.

See the *Tutorial*, section 1.3: "Types, terms and formulae"

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`).

Product Types for Pairs

- (x_1, x_2) has type $\tau_1 * \tau_2$ provided $x_i :: \tau_i$
- $(x_1, \ldots, x_{n\text{-}1}, x_n)$ abbreviates $(x_1, \ldots, (x_{n\text{-}1}, x_n))$
- Extensible record types can also be defined.

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

Only integer constants are available. Note *that* traditional notation for floating point numbers would be inappropriate, but rational numbers can be expressed.

HOL as a Functional Language

recursive data type of lists

```
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)"

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

recursive functions
(types can be inferred)

Recursive data types can be defined as in ML, although with somewhat less generality. Recursive functions can also be declared, provided Isabelle can establish their termination; all functions in higher-order logic are total. Naturally terminating recursive definitions pose no difficulties for Isabelle. In complicated situations, it is possible to give a hint.

Proof by Induction

declaring a lemma

use it to simplify other formulas

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

two steps: *induction* followed by *automation*

end of proof

Example of a *Structured Proof*

- base case and inductive step can be proved explicitly

- Invaluable for proofs that need intricate manipulation of facts

```
lemma "app xs Nil = xs"
proof (induct xs)
  case Nil
  show "app Nil Nil = Nil"
    by auto
next
  case (Cons a xs)
  show "app (Cons a xs) Nil = Cons a xs"
    by auto
qed
```

Interactive Formal Verification
2: Isabelle Theories

Lawrence C Paulson
Computer Laboratory
University of Cambridge

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

See the *Tutorial*, section 1.2 (Theories) and 2.1 (An Introductory Theory).

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"
types
  val   = nat -- "values"
  state = "loc => val"
  aexp  = "state => val"
  bexp  = "state => bool"    -- "just 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

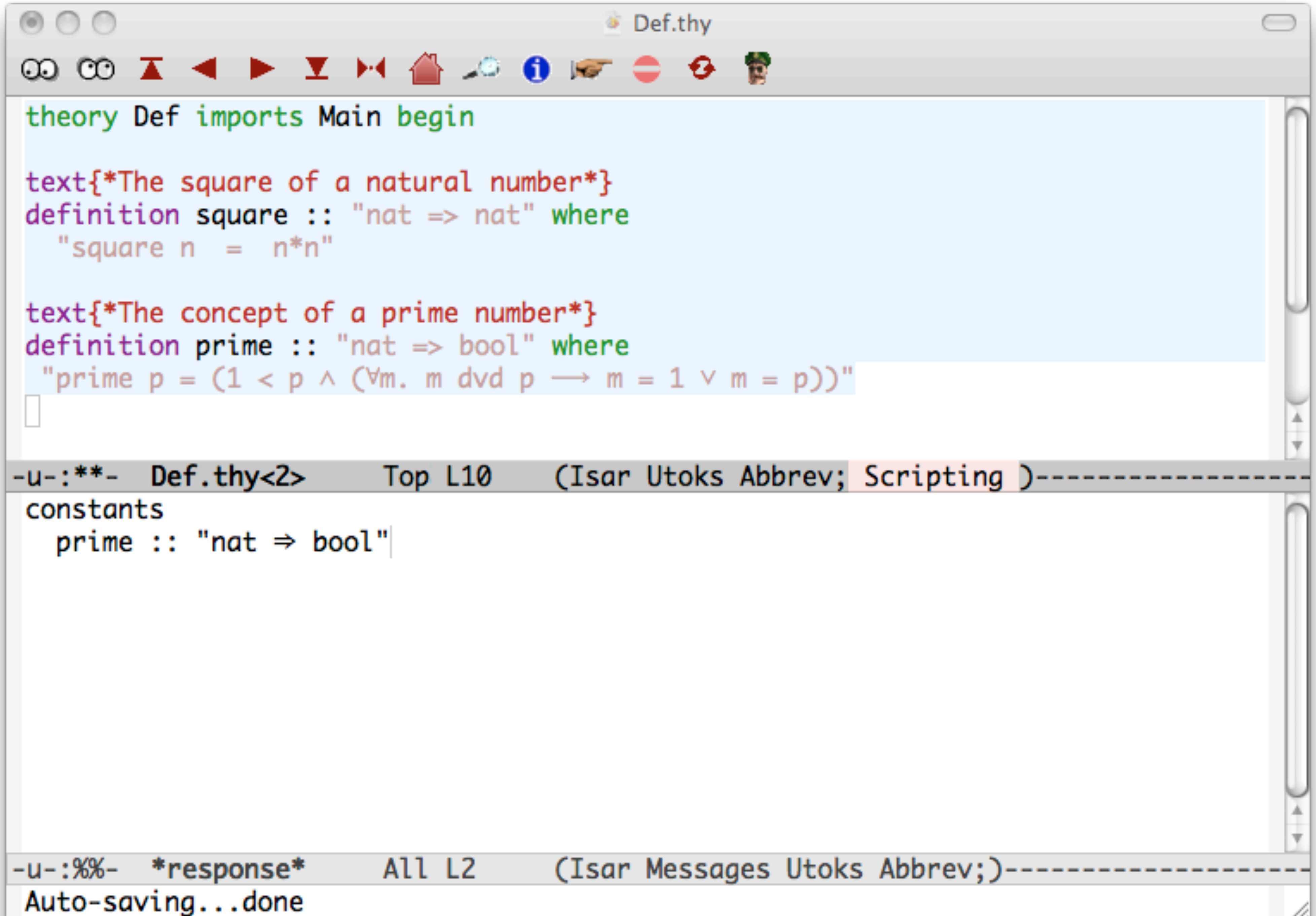
recursive type of commands

Notes on Type Declarations

- 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*.

See the tutorial, section 2.5.

Basic Constant Definitions



See the *Tutorial*, Section 2.7.2 Constant Definitions.

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

- 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!

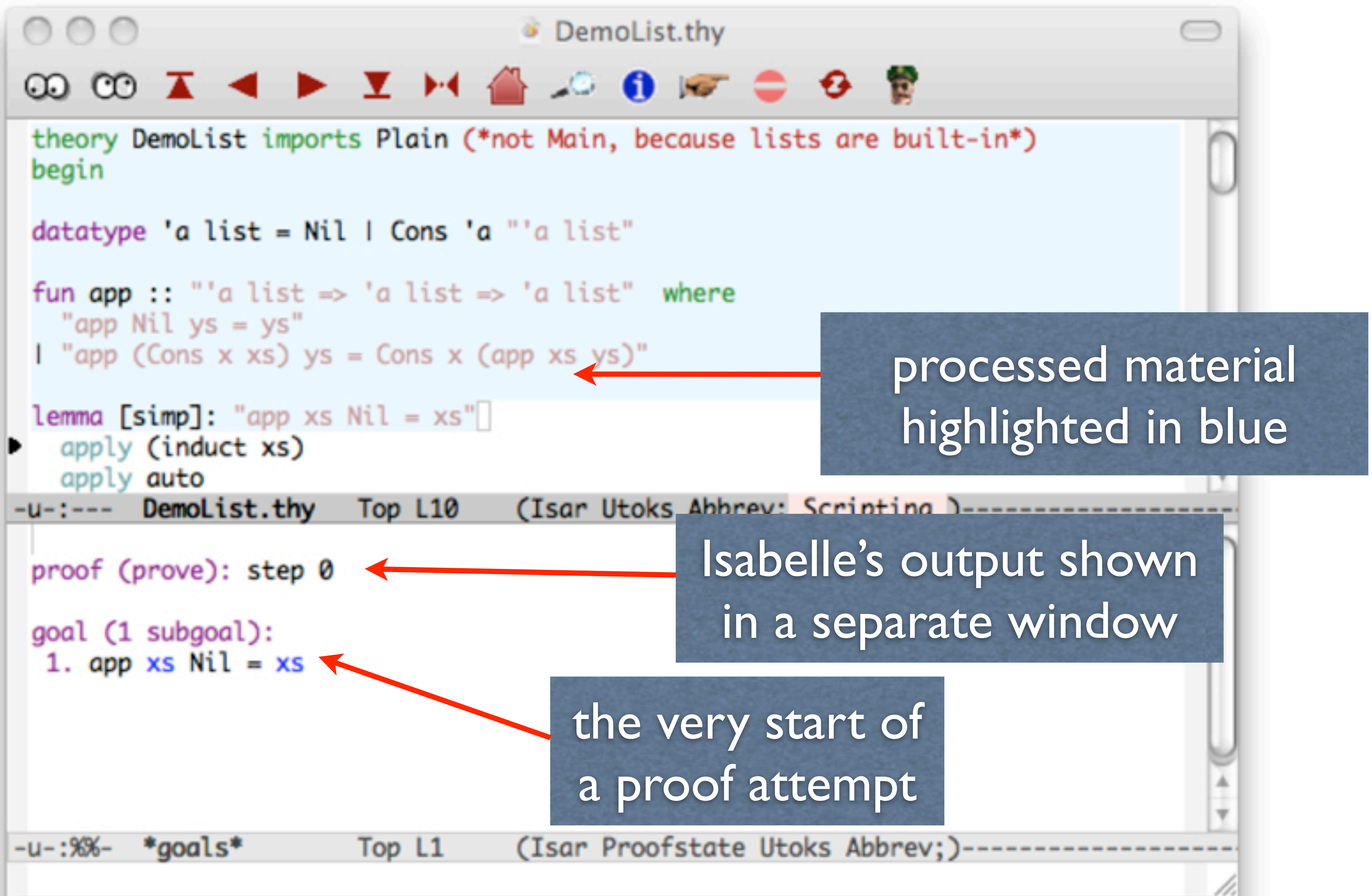
List Induction Principle

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

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

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).

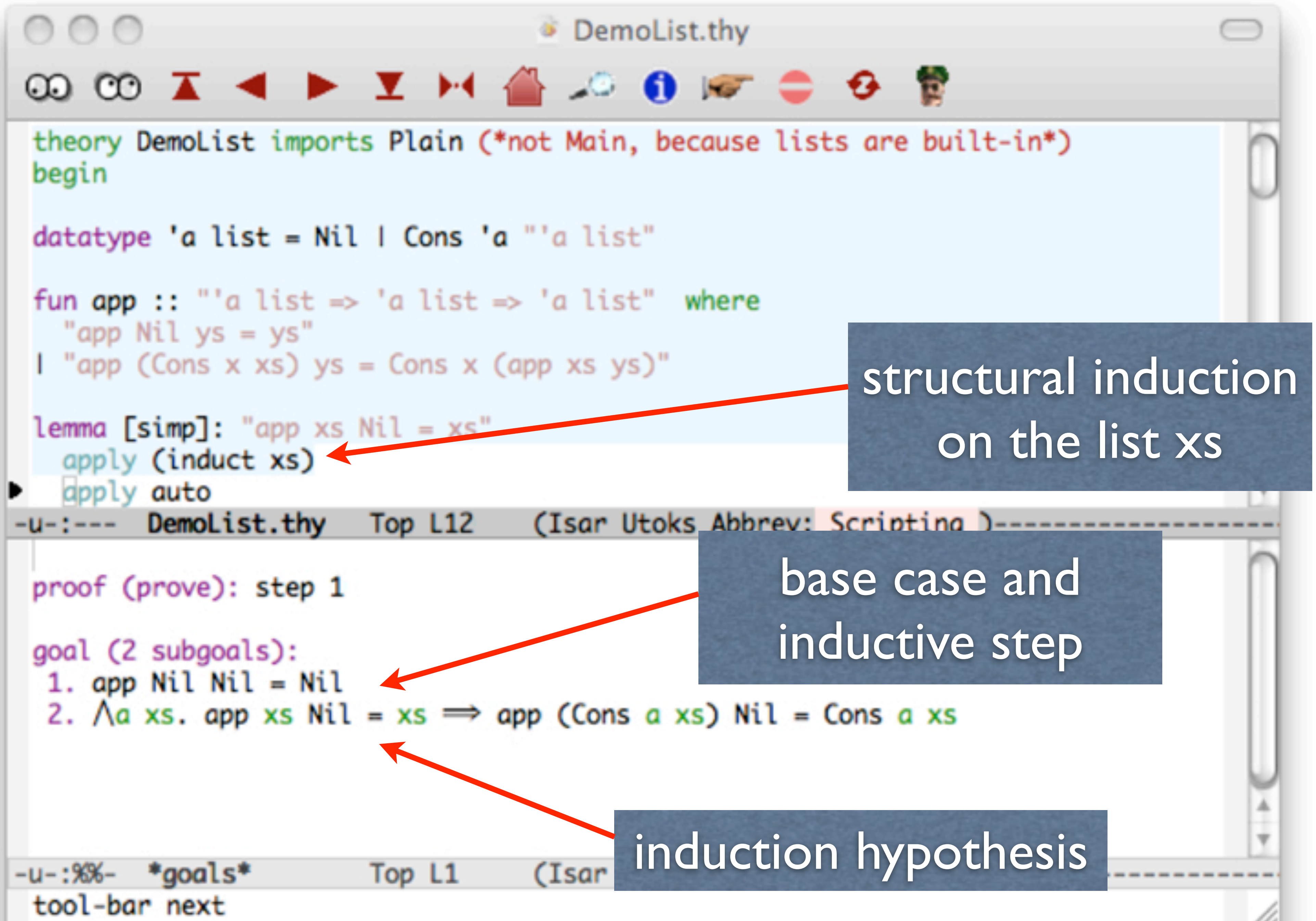
Proof General



Isabelle's user interface, Proof General, was developed by David Aspinall. It has a separate website: http://proofgeneral.inf.ed.ac.uk/

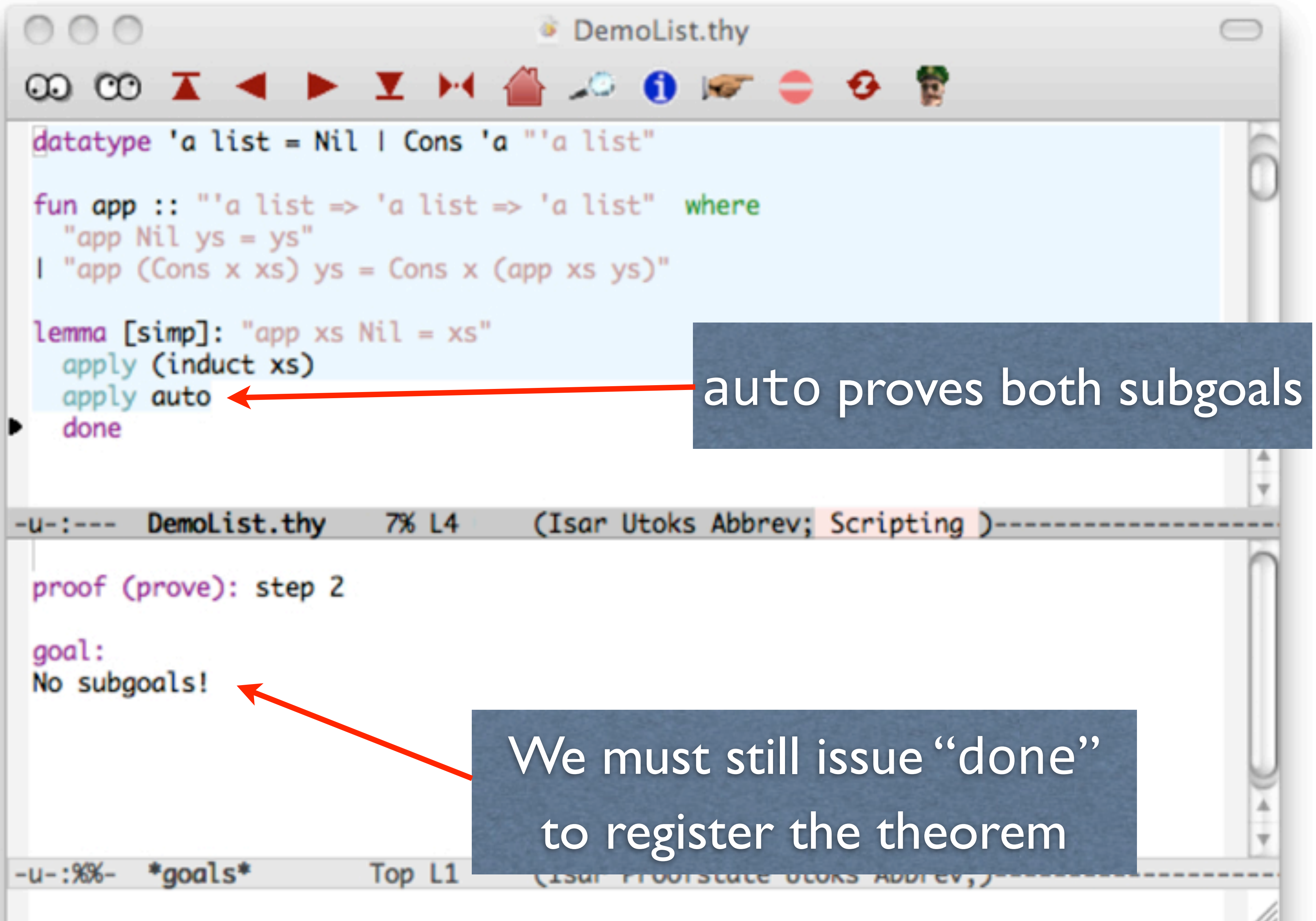
Proof General runs under Emacs, preferably version 23. Isabelle is almost impossible to use other than through Proof General.

Proof by Induction



See the tutorial, section 2.3 (An Introductory Proof). For the moment, there is no important difference between induct_tac (used in the tutorial) and induct (used above). With both of these proof methods, you name an induction variable and it selects the corresponding structural induction rule, based on that variable's type. It then produces an instance of induction sufficient to prove the property in question.

Finishing a Proof

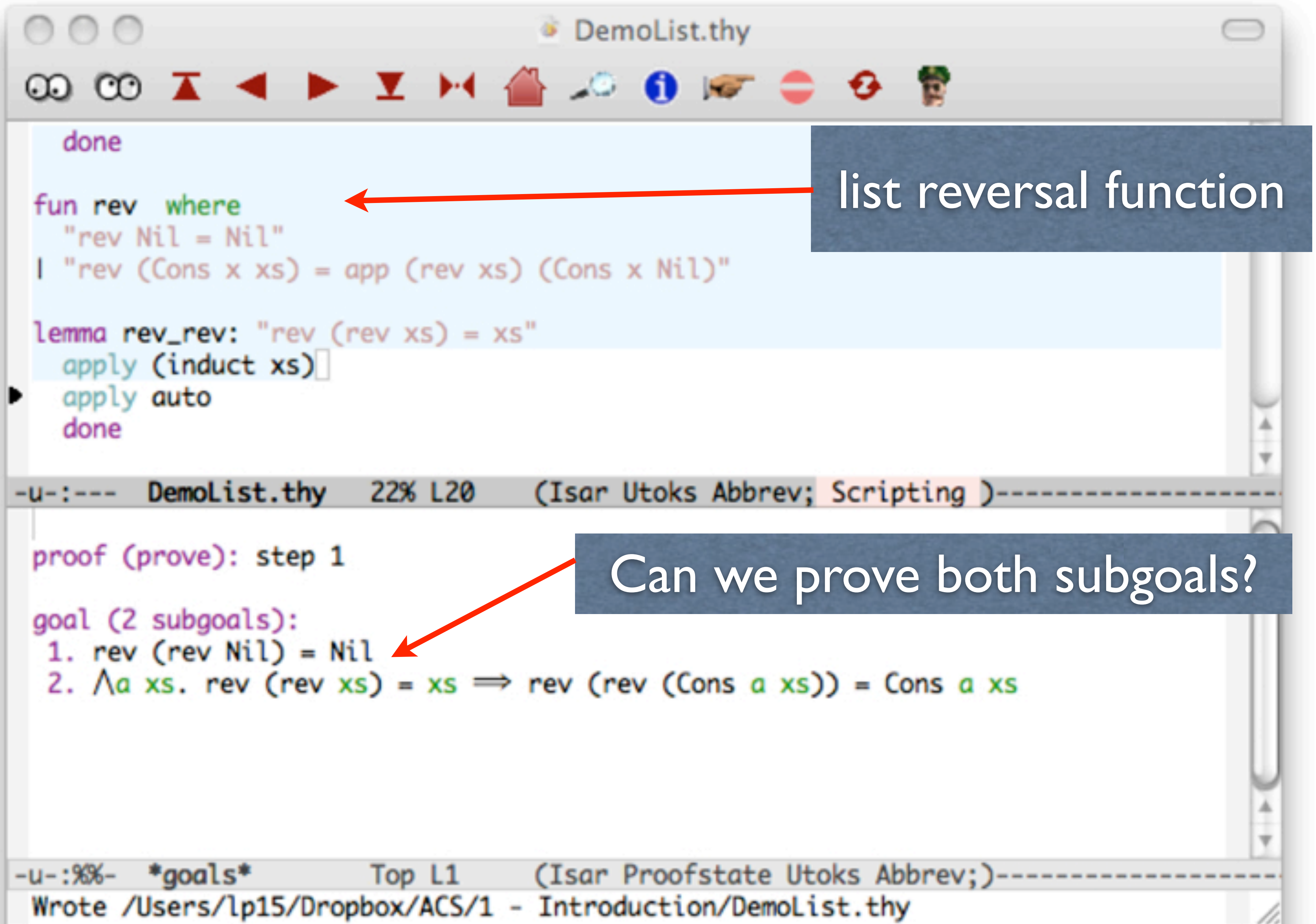


By default, Isabelle simplifies applications of recursive functions that match their defining recursion equations. This is quite different to the treatment of non-recursive definitions.

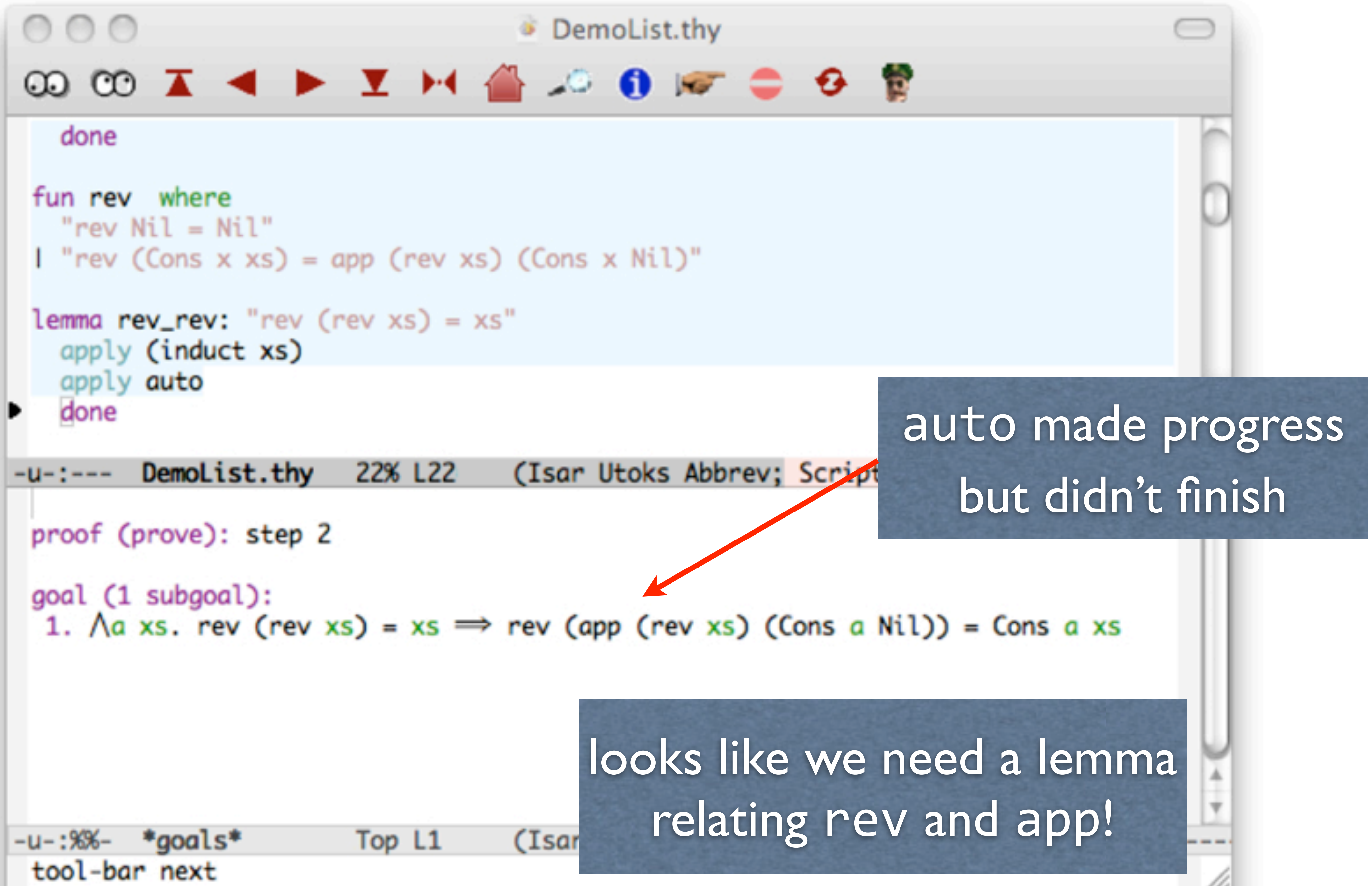
Isabelle's user interface, Proof General, was developed by David Aspinall. It has a separate website: http://proofgeneral.inf.ed.ac.uk/

Proof General runs under Emacs, preferably version 23. Isabelle is almost impossible to use other than through Proof General.

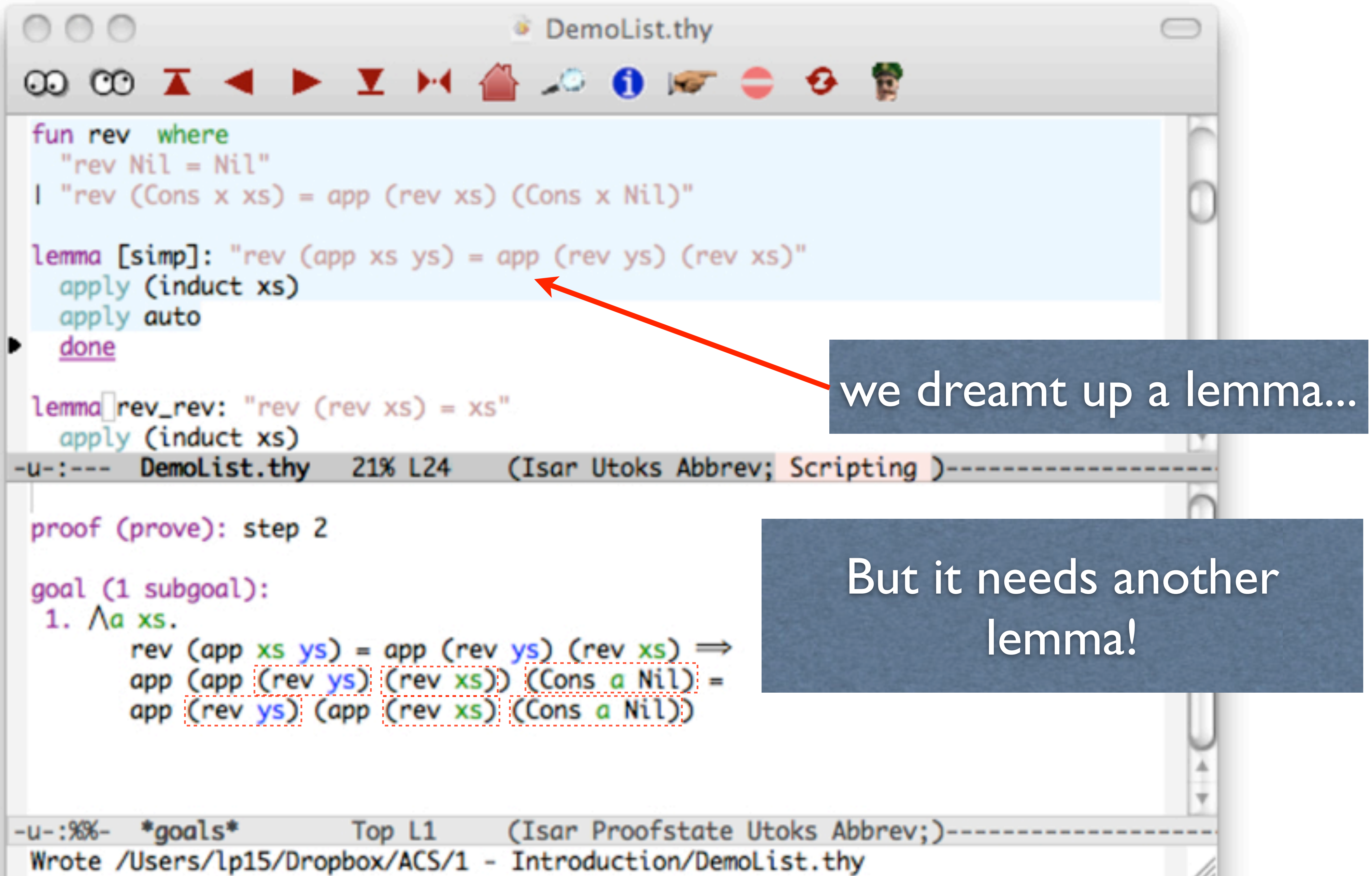
Another Proof Attempt



Stuck!

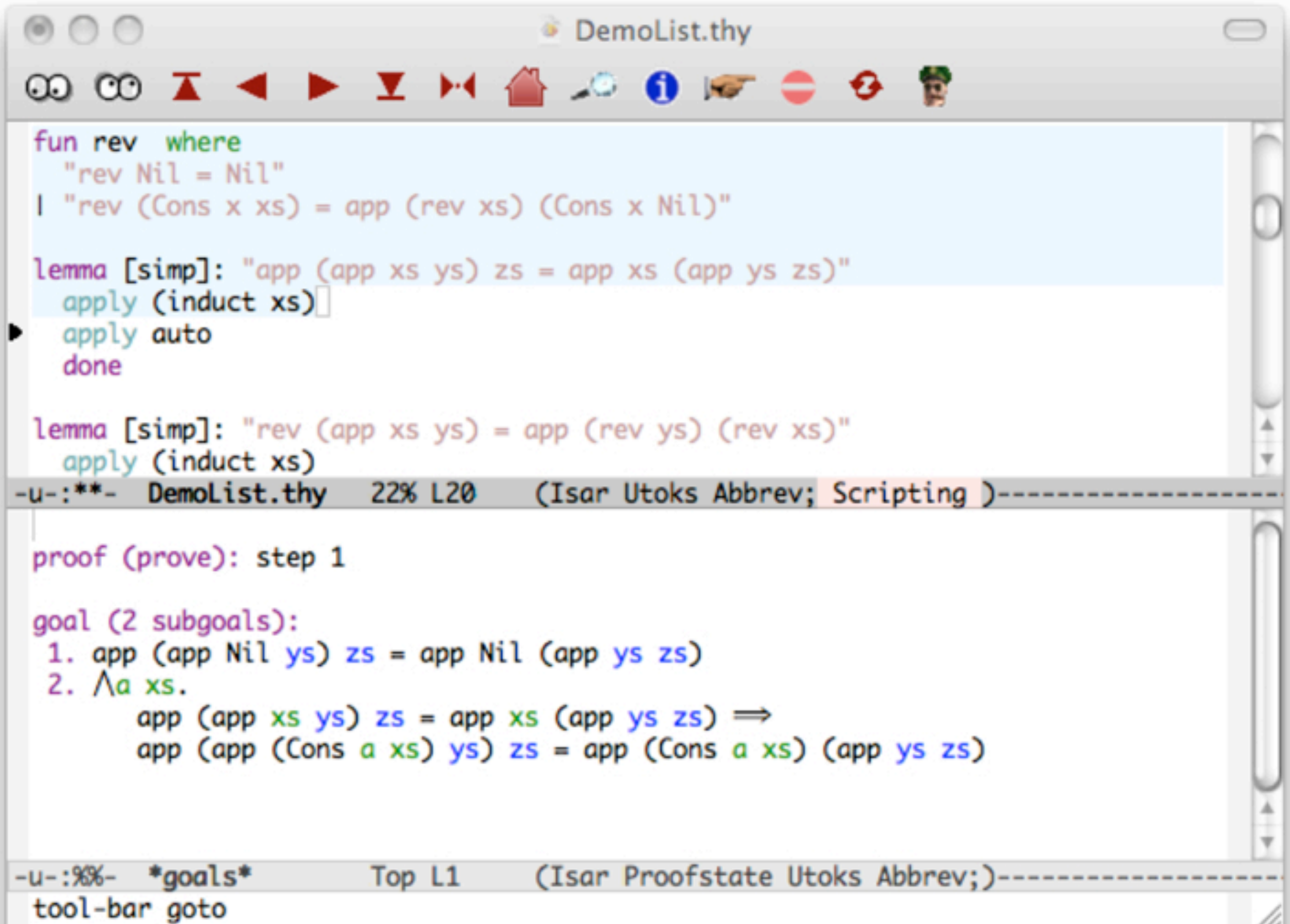

DemoList.thy
done
fun rev where
"rev Nil = Nil"
| "rev (Cons x xs) = app (rev xs) (Cons x Nil)"
lemma rev_rev: "rev (rev xs) = xs"
apply (induct xs)
apply auto
done
-u-:--- DemoList.thy 22% L22 (Isar Utoks Abbrev; Script
proof (prove): step 2
goal (1 subgoal):
1. ⋀a xs. rev (rev xs) = xs ⟹ rev (app (rev 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!



The subgoal that we cannot prove looks very complicated. But when we notice the repeated terms in it, we see that it is an instance of something simple and natural: the associativity of the function `app`. This fact does not involve the function `rev`! We see in this example how crucial it is to prove properties in the most abstract and general form.

The Final Piece of the Jigsaw



This proof of associativity will be successful, and with its help, the other lemmas are easily proved.

Interactive Formal Verification
3: Elementary Proof

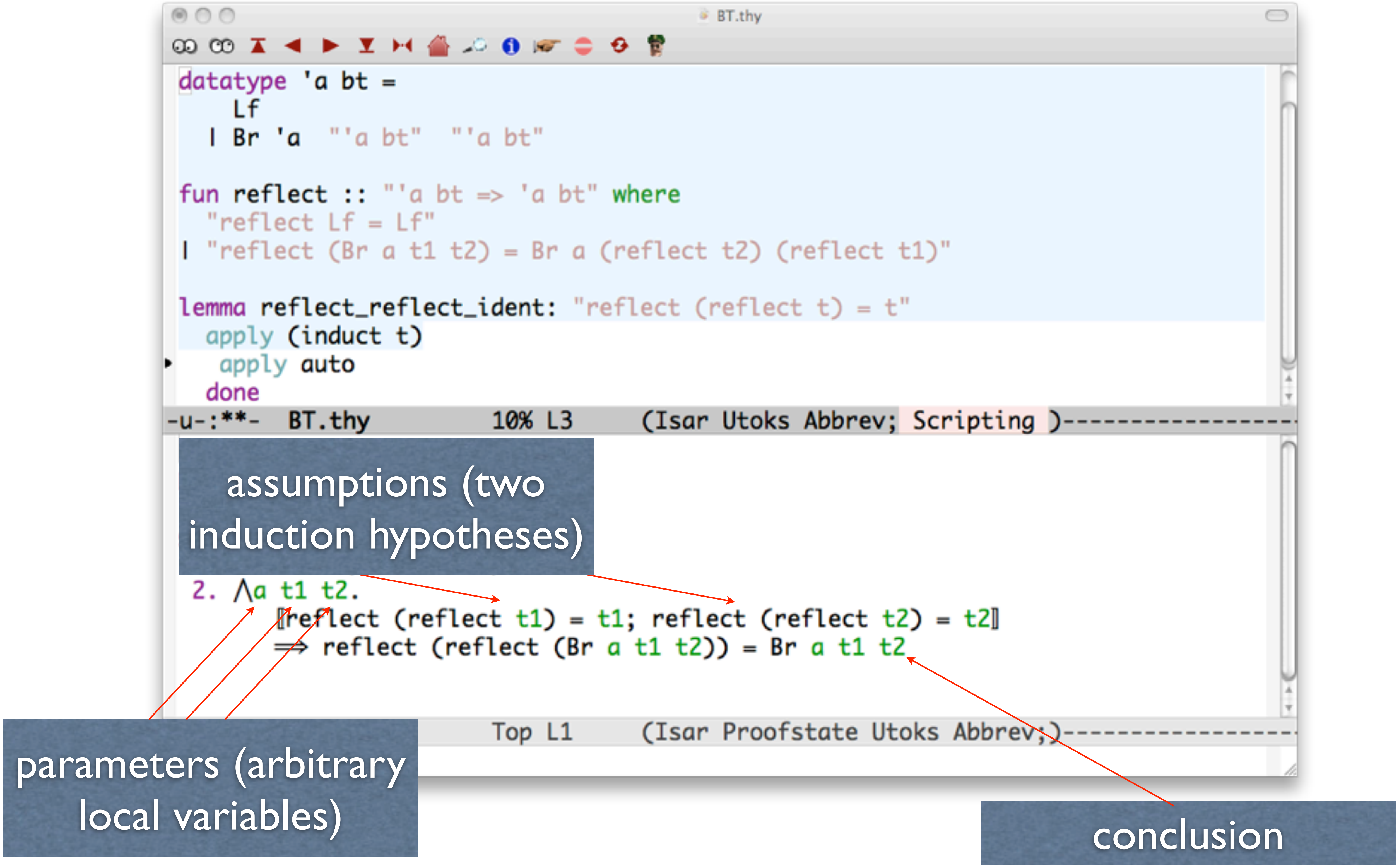
Lawrence C Paulson
Computer Laboratory
University of Cambridge

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.
- Certain methods, notably `auto` and `simp_all`, operate on all outstanding subgoals.
- We finish when no subgoals remain.

See the Tutorial, 2.3 An Introductory Proof. The list of subgoals is always flat. Towards the end of this course, there is a brief introduction to structured proofs.

Structure of a Subgoal



Proof by Rewriting

```
app (Cons x xs) ys ⇒ Cons x (app xs ys)           recursive defns
    rev (Cons x xs) ⇒ app (rev xs) (Cons x Nil)
    rev (app xs ys) ⇒ app (rev ys) (rev xs)        lemma
app (app xs ys) zs ⇒ app xs (app ys zs)           induction hyp
```

```
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))
```

Rewriting with Equivalencies

```
(x dvd -y) = (x dvd y)
(a * b = 0) = (a = 0 ∨ b = 0)
(A - B ⊆ C) = (A ⊆ B ∪ C)
(a*c ≤ b*c) = ((0<c → a ≤ b) ∧ (c<0 → b ≤ a))
```

introduces a case split on the sign of c

- Logical equivalencies are just boolean equations.
- They lead to a clear and simple proof style.
- They can also be written with the syntax $P \leftrightarrow Q$.

Automatic Case Splitting

Simplification will replace

$$P(\texttt{if}\ b\ \texttt{then}\ x\ \texttt{else}\ y)$$

by

$$(b \rightarrow P(x)) \wedge (\neg b \rightarrow P(y))$$

- By default, this only happens when simplifying the conclusion.
- Other case splitting can be enabled.

See the *Tutorial*, 3.1.9 Automatic Case Splits

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.

Termination Issues

- *Looping*: `f(x) = h(g(x))`, `g(x) = f(x+2)`
- *Looping*: `P(x) ⇒ x=0`
  - `simp` will try to use this rule to simplify its own precondition!
- `x+y = y+x` is actually okay!
  - *Permutative rewrite rules* are applied but only if they make the term “lexicographically smaller”.

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

See the *Tutorial*, 3.1 Simplification. This section describes the options and possibilities thoroughly.

Variations on `simp` and `auto`

using another rewrite rule

omitting a certain rule

```
simp add: add_assoc

simp del: rev_rev (no_asm_simp)

simp (no_asm)

simp_all (no_asm_simp) add: … del: …

auto simp add: … del: …
```

not simplifying the assumptions

ignoring all assumptions

do `simp` for all subgoals

auto with options

Rules for Arithmetic

- An identifier can denote a *list* of lemmas.
- `add_ac` and `mult_ac`: associative/commutative properties of addition and multiplication
- `algebra_simps`: useful for multiplying out polynomials
- `field_simps`: useful for multiplying out the denominators when proving inequalities

  Example: `auto simp add: field_simps`

These identifiers denote lists of theorems that work together well as rewrite rules for performing various simplification tasks.

Simple Proof by Induction

- State the desired theorem using "`lemma`", with its name and optionally `[simp]`
- Identify the *induction variable*
  - Its type should be some datatype (incl. `nat`)
  - It should appear as the *argument of a recursive function*.
- Complicating issues include unusual recursions and auxiliary variables.

Completing the Proof

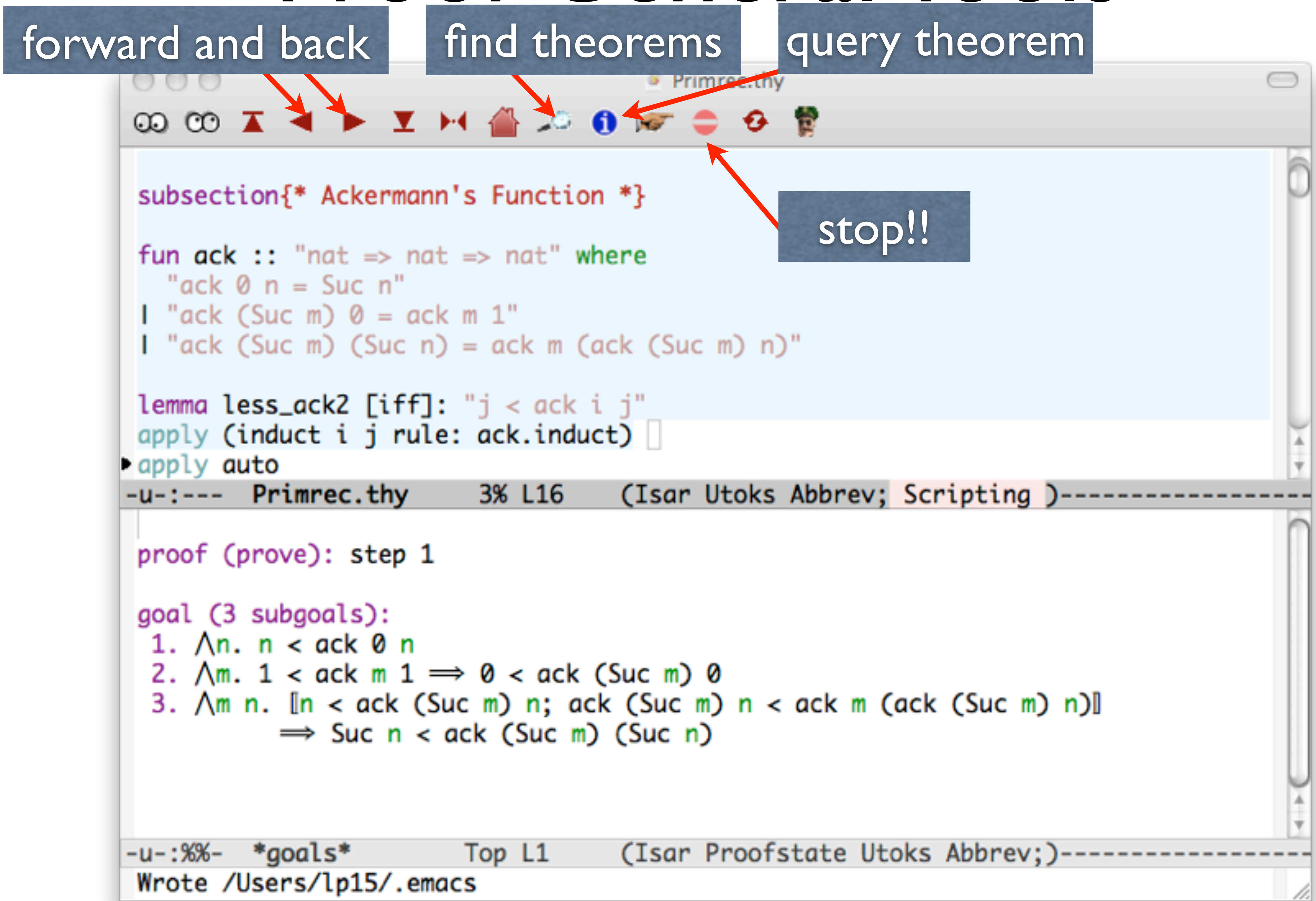
- Apply "`induct`" with the chosen variable.
- The first subgoal will be the base case, and it should be trivial using "`simp`".
- Other subgoals will involve induction hypotheses and the proof of each may require several steps.
- Naturally, the first thing to try is "`auto`", but much more is possible.

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.

See the *Tutorial*, **3.1.11 Finding Theorems,** for a description of allowed search terms.

Proof General Tools



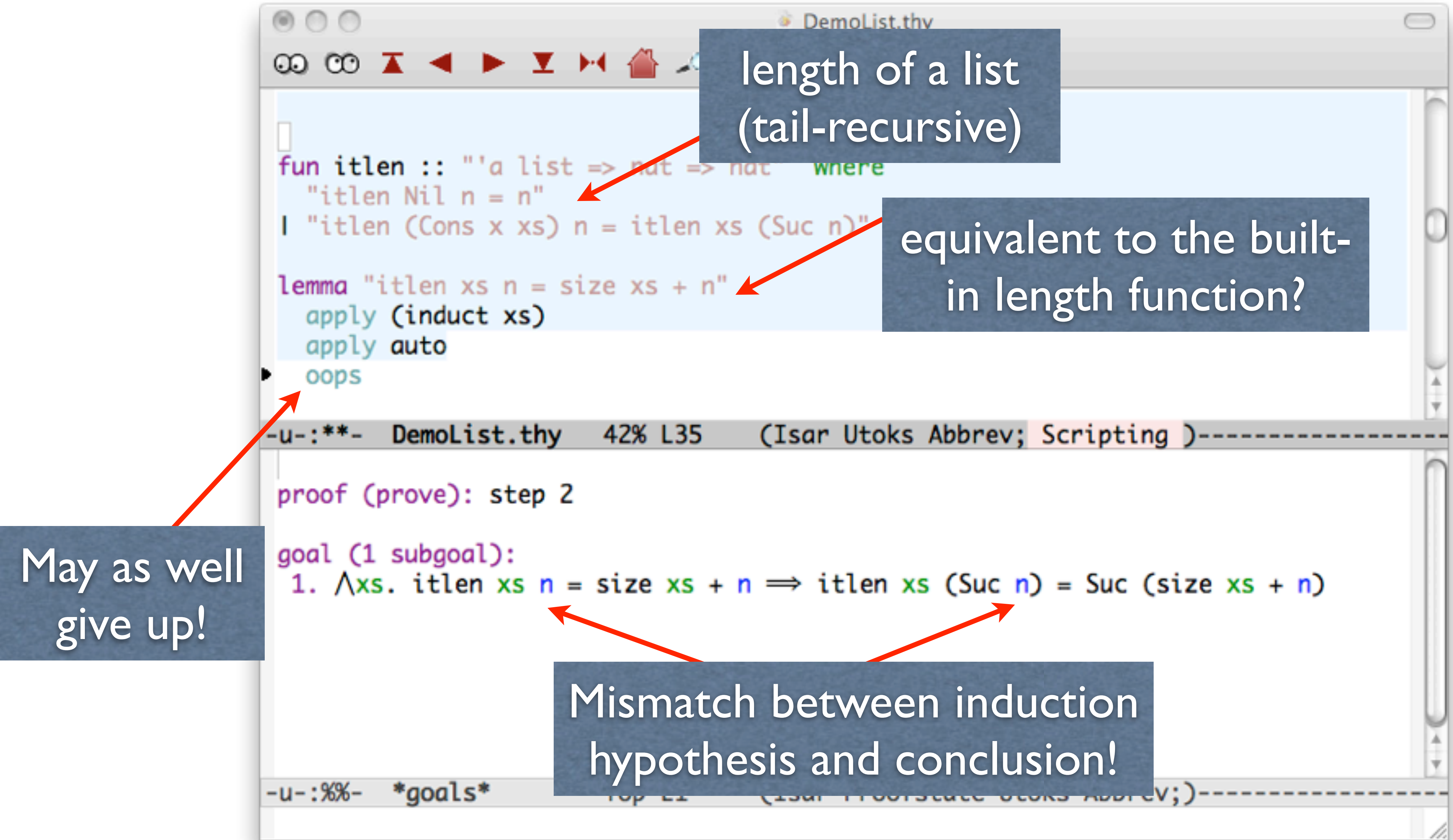
See the *Tutorial*, **3.1.11 Finding Theorems,** for a description of allowed search terms.

Hover the mouse over the tools to see ToolTips (brief descriptions of each).

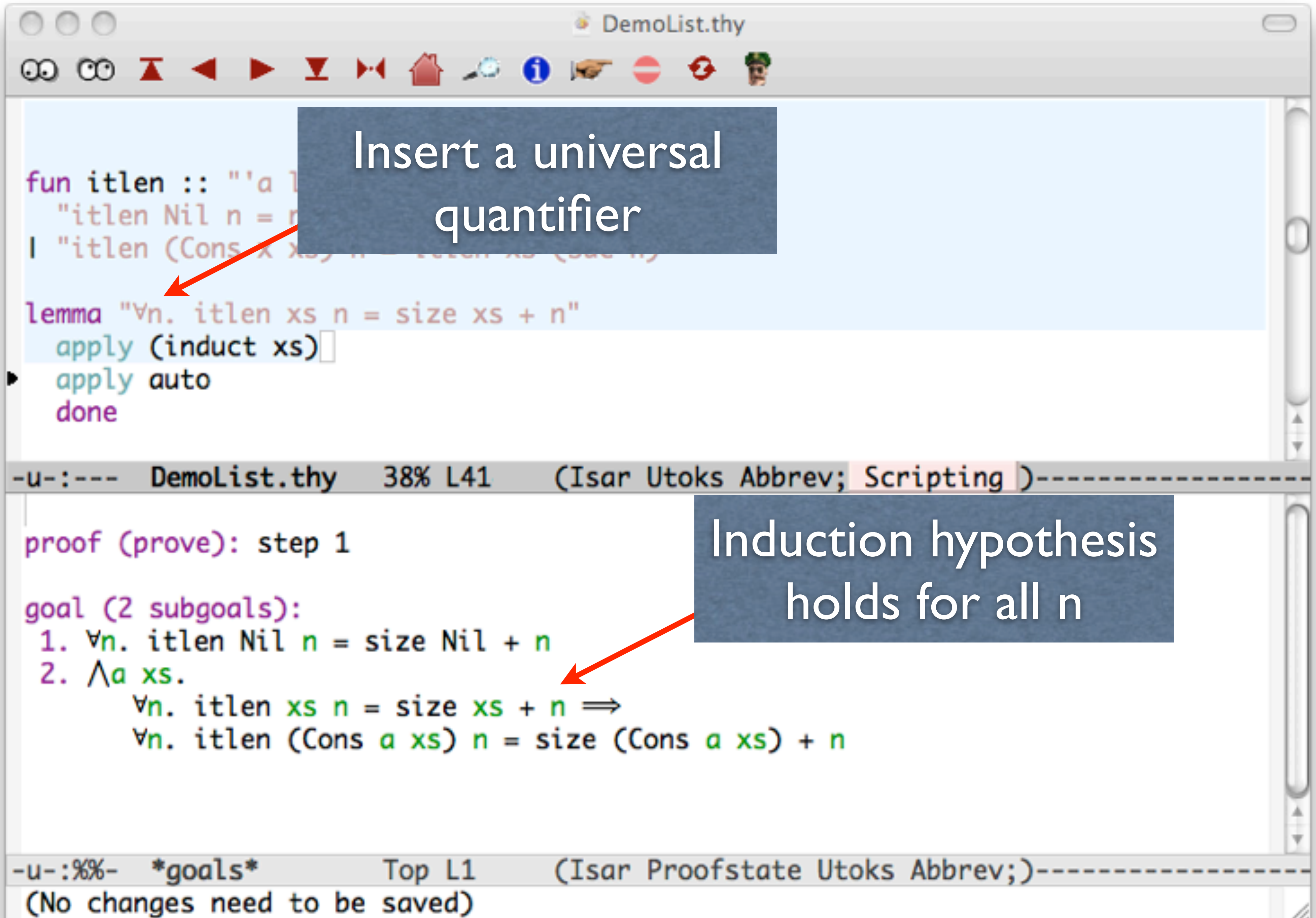
Interactive Formal Verification
4: Advanced Recursion, Induction and Simplification

Lawrence C Paulson
Computer Laboratory
University of Cambridge

A Failing Proof by Induction

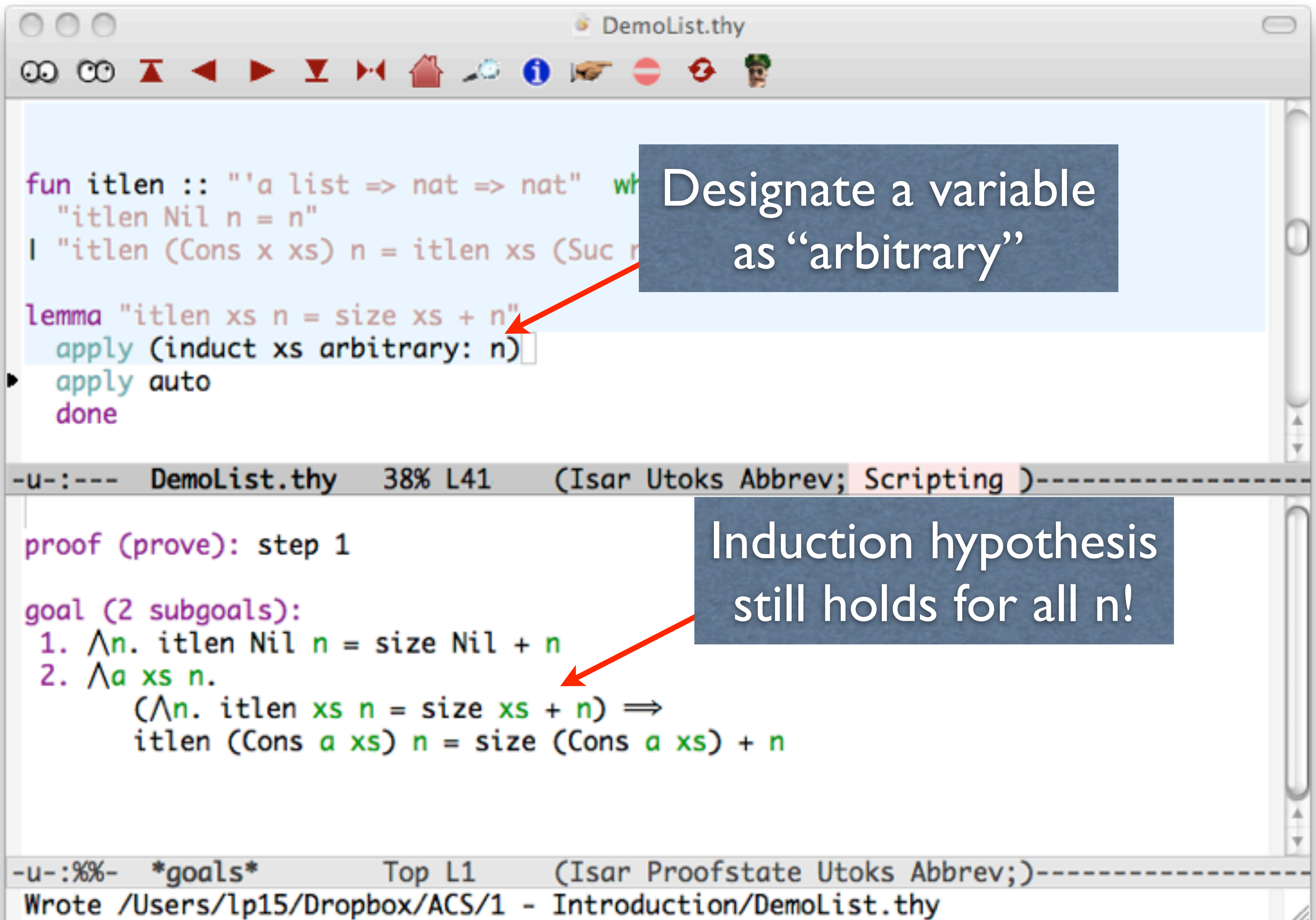


Generalising the Induction



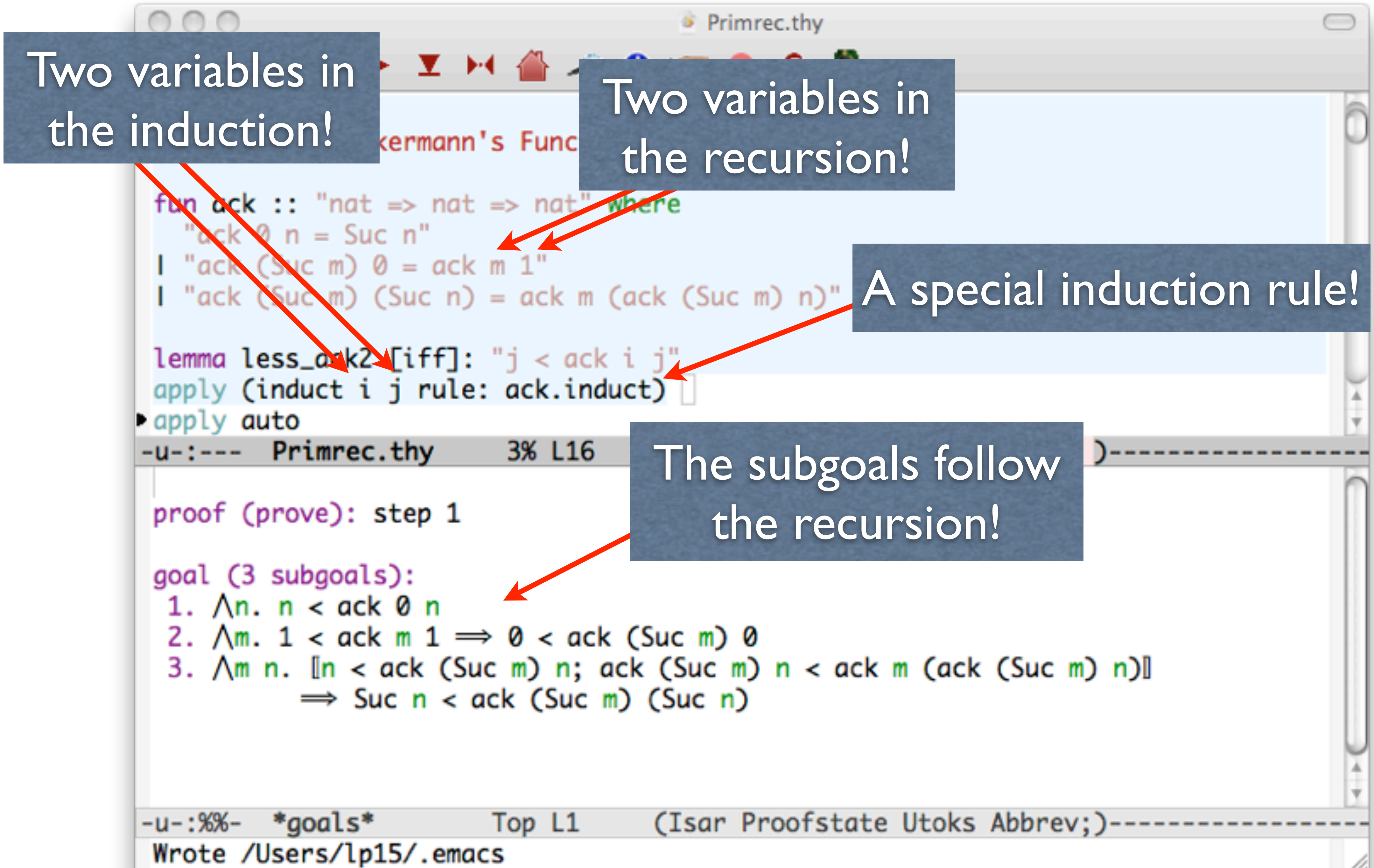
The need to generalise the induction formula in order to obtain a more general induction hypothesis Is well known from mathematics. Logically, note that the induction formula above has only one free variable: xs. The induction formula on the previous slide has two free variables: xs and n.

Generalising: Another Way



The approach described above is logically similar to the one on the previous slide, but it avoids the use of a universal quantifier (∀) in the theorem statement. Because Isabelle is a logical framework, it has meta-level versions of the universal quantifier and the implication symbol, and we generally avoid universal quantifiers in theorems. But it is important to remember that behind the convenience of the method illustrated here is a straightforward use of logic: we are still generalising induction formula. For more complicated examples, see the *Tutorial*, 9.2.1 **Massaging the Proposition**.

Unusual Recursions



For full documentation, see *Defining Recursive Functions in Isabelle/HOL*, by Alexander Krauss.

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**.

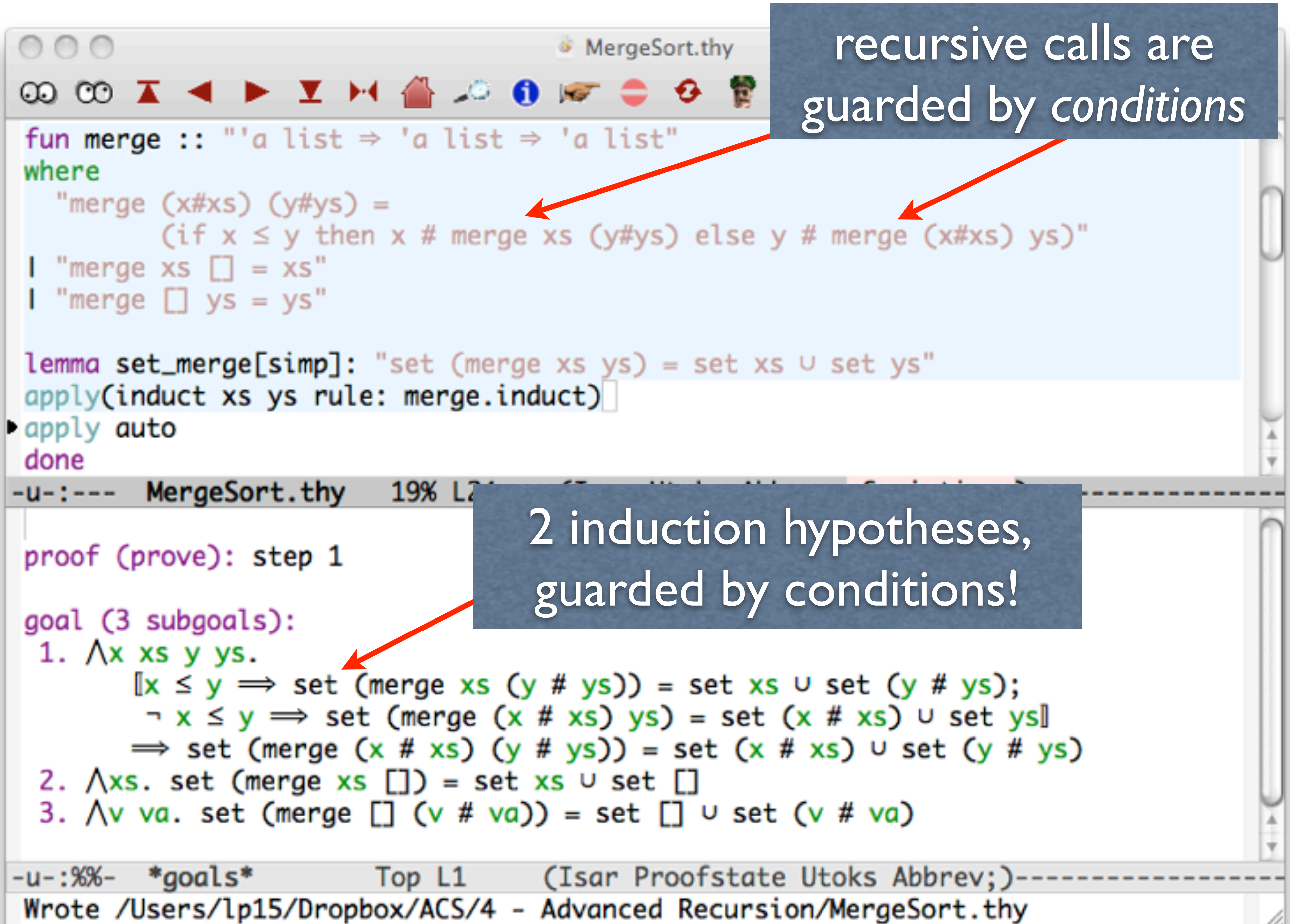
Isabelle provides the command `primrec` for primitive recursion as well. It is closely based on the internal derivation of recursion, and can handle function definitions involving certain complicated features (in particular, higher-order primitive recursion) where fun fails. See the *Tutorial*, **2.1 An Introductory Theory**. More difficult examples of `primrec` are covered in **3.3 Case Study: Compiling Expressions**.

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!

The Ackermann example proves several lemmas using the special rule, but several others using ordinary mathematical induction!

Another Unusual Recursion



Again, see *Defining Recursive Functions in Isabelle/HOL*. Each induction hypothesis can only be used if the corresponding condition is provable.

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 first rewriting step in the proof unfolds the definition of `merge`. The second one is a case-split involving `if`. This step introduces a conjunction of implications, creating contexts that exactly match the induction hypotheses. But first, the definition of `set` (a function that maps a list to the finite set of its elements) must be unfolded. The last step highlighted above applies the induction hypotheses. The remaining steps, not shown, prove the equality between the set expressions just produced and the right-hand side of the original subgoal.

The Case Expression

- Similar to that found in the functional language ML.
- Automatically generated for every Isabelle datatype.
- The simplifier can (upon request!) perform case-splits analogous to those for "`if`".
- Case splits in assumptions (as opposed to the conclusion) never happen unless requested.

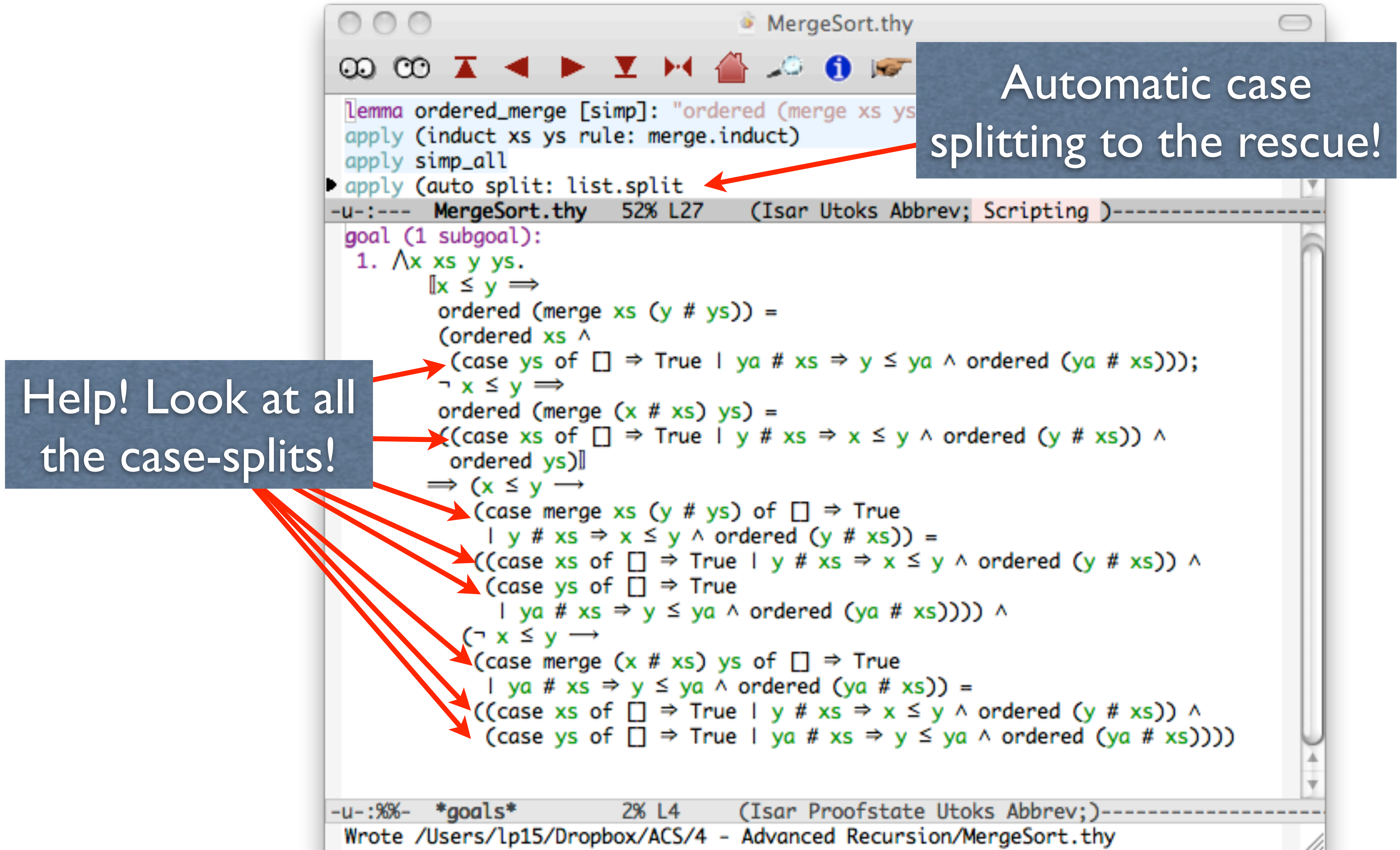
Case-Splits for Lists

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

The definition shown on the slide describes the same function as the following one:

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

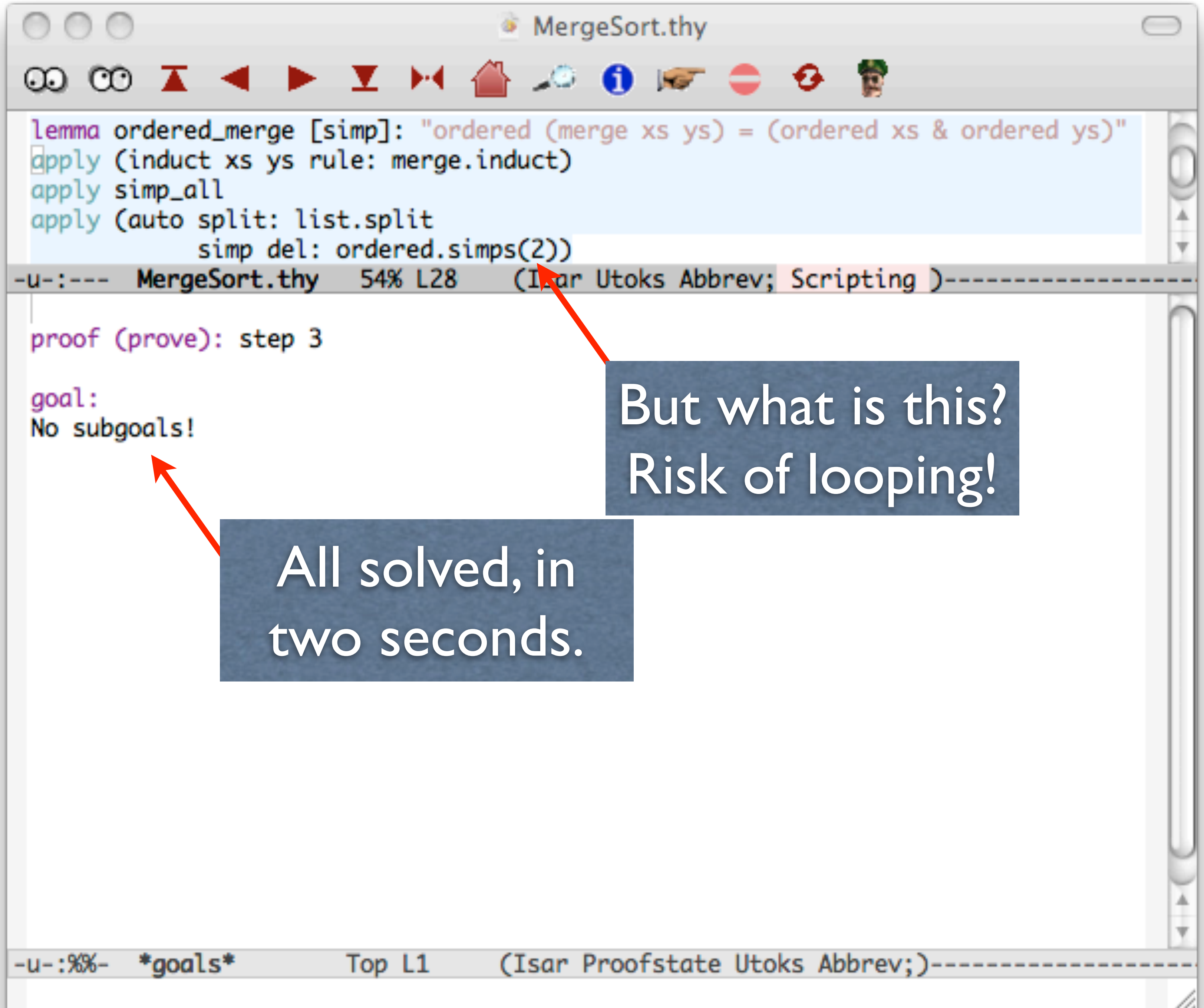
Case-Splitting in Action



There isn't room to show the full subgoal, but the second part of the conjunction (beginning with ¬ x ≤ y) has a similar form to the first part, which is visible above.

Note that the last step used was `simp_all`, rather than `auto`. The latter would break up the subgoal according to its logical structure, leaving us with 14 separate subgoals! Simplification, on the other hand, seldom generates multiple subgoals. The one common situation where this can happen is indeed with case splitting, but in our example, case splitting completely proves the theorem.

Completing the Proof



The identifier `ordered.simps` refers to the two equations that make up the definition of the function `ordered`. The suffix `(2)` selects the second of these. Now "`simp del: ordered.simps (2)`" tells `auto` to ignore this equation. Otherwise, the call will run forever.

Case Splitting for Lists

Simplification will replace

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

by

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

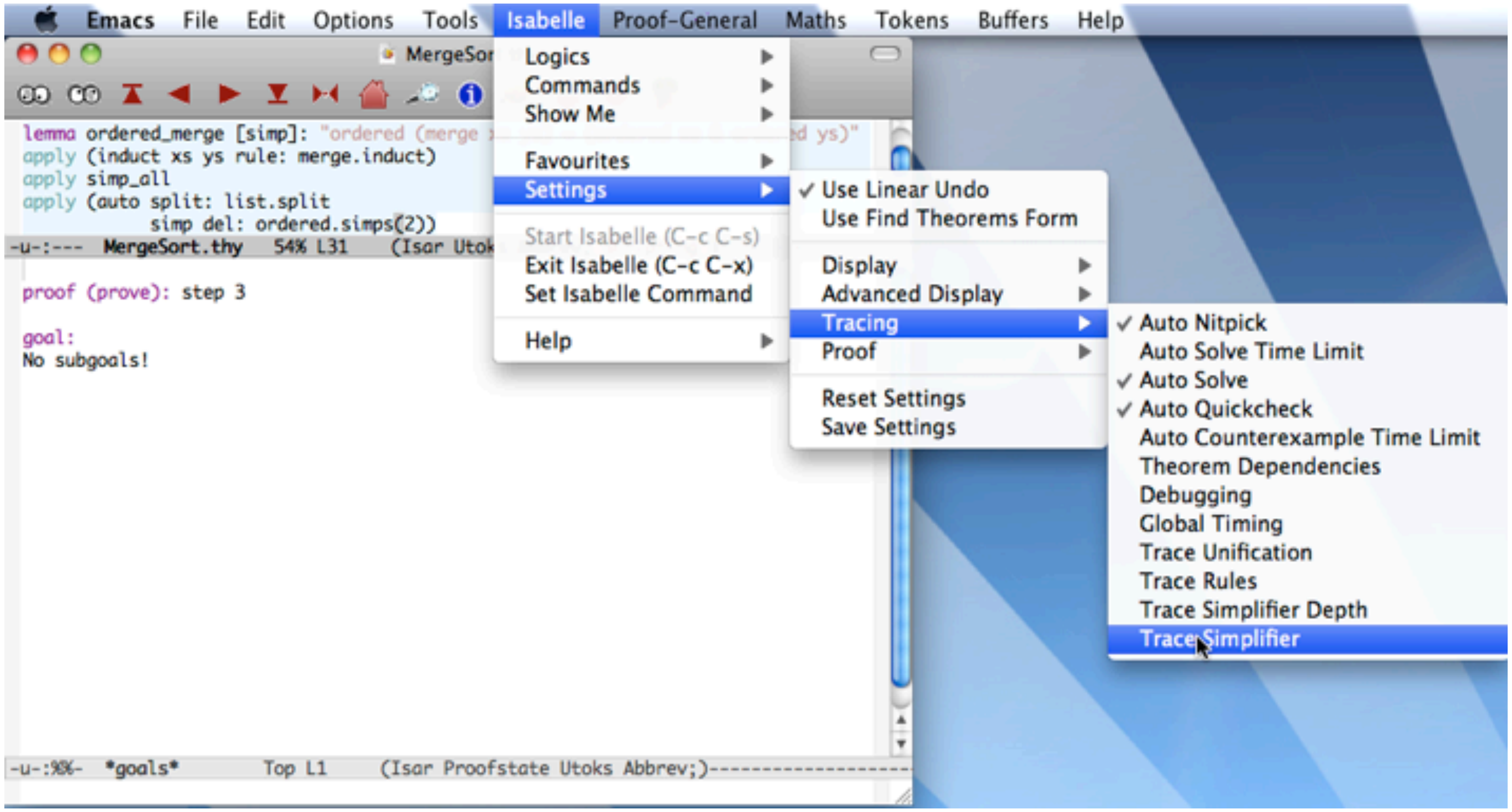
- It creates a case for each datatype constructor.
- Here it causes looping if combined with the second rewrite rule for `ordered`.

Specifically, a case split will create an instance where the list has the form a#l, and therefore ordered(a#l) will rewrite to another instance of case, *ad infinitum*.

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



Many tracing options can be enabled within Proof General. Switch them off unless you need them, because they can generate an enormous output and take a lot of processor time. Their interpretation is seldom easy!

Interactive Formal Verification
5: Logic in Isabelle

Lawrence C Paulson
Computer Laboratory
University of Cambridge

Logical Frameworks

- A formalism to represent other formalisms
- Support for *natural deduction*
- A common basis for implementations
- Type theories are commonly used, but Isabelle uses a simple meta-logic whose main primitives are
  - $\Rightarrow$ (implication)
  - $\bigwedge$ (universal quantification).

Natural Deduction in Isabelle

$$\frac{P \quad Q}{P \wedge Q}$$

`P ⟹ (Q ⟹ P ∧ Q))`

$$\frac{P \wedge Q}{P}$$

`P ∧ Q ⟹ P`

$$\frac{P \wedge Q}{Q}$$

`P ∧ Q ⟹ Q`

$$\frac{P \longrightarrow Q \quad P}{Q}$$

`P⟶Q ⟹ (P ⟹ Q)`

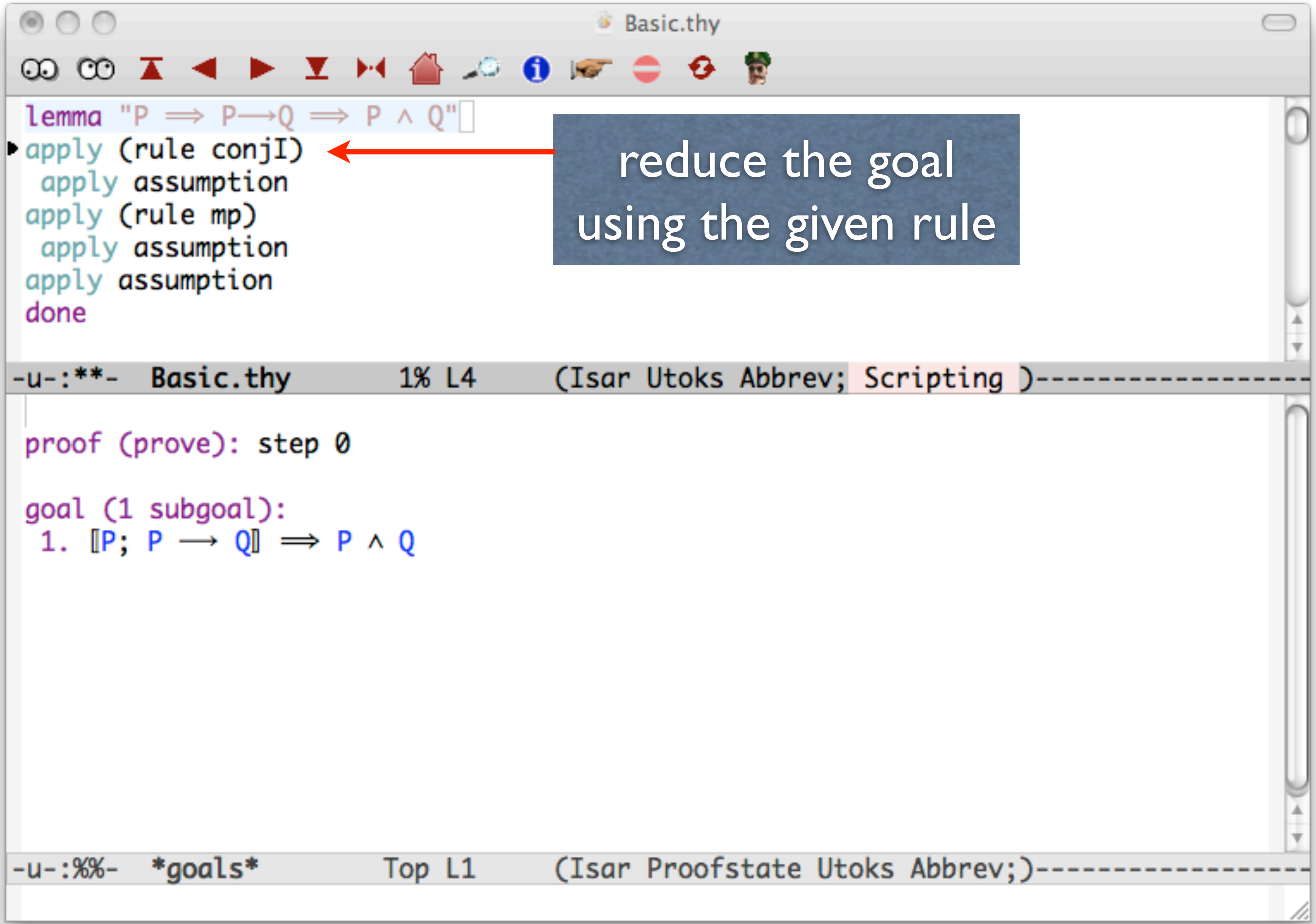
See the *Tutorial*, Chapter 5: **The Rules of the Game**. The first of these is an *introduction* rule, `conjI` in Isabelle. The following three are *elimination* rules: `conjunct1`, `conjunct2`, `and mp`. Isabelle parlance, these three are actually *destruction* rules because they lack the general form of an elimination rule in natural deduction.

Meta-implication

- The symbol $\Rightarrow$ (or ==>) expresses the relationship between premise and conclusion
- ... and between subgoal and goal.
- It is distinct from $\rightarrow$, which is not part of Isabelle's underlying logical framework.
- $P \Rightarrow (Q \Rightarrow R)$ is abbreviated as $[\![P; Q]\!] \Rightarrow R$

The distinction between meta- and object-connectives is a common source of confusion among students. This distinction is inherent in the use of a logical framework. There is no reason why an object-logic would have an implication symbol at all. Isabelle gives a special significance to $\Rightarrow$, in particular for expressing the structure of inference rules, as shown on previous slide. This would be impossible if we make no distinction between $\Rightarrow$ and $\rightarrow$.

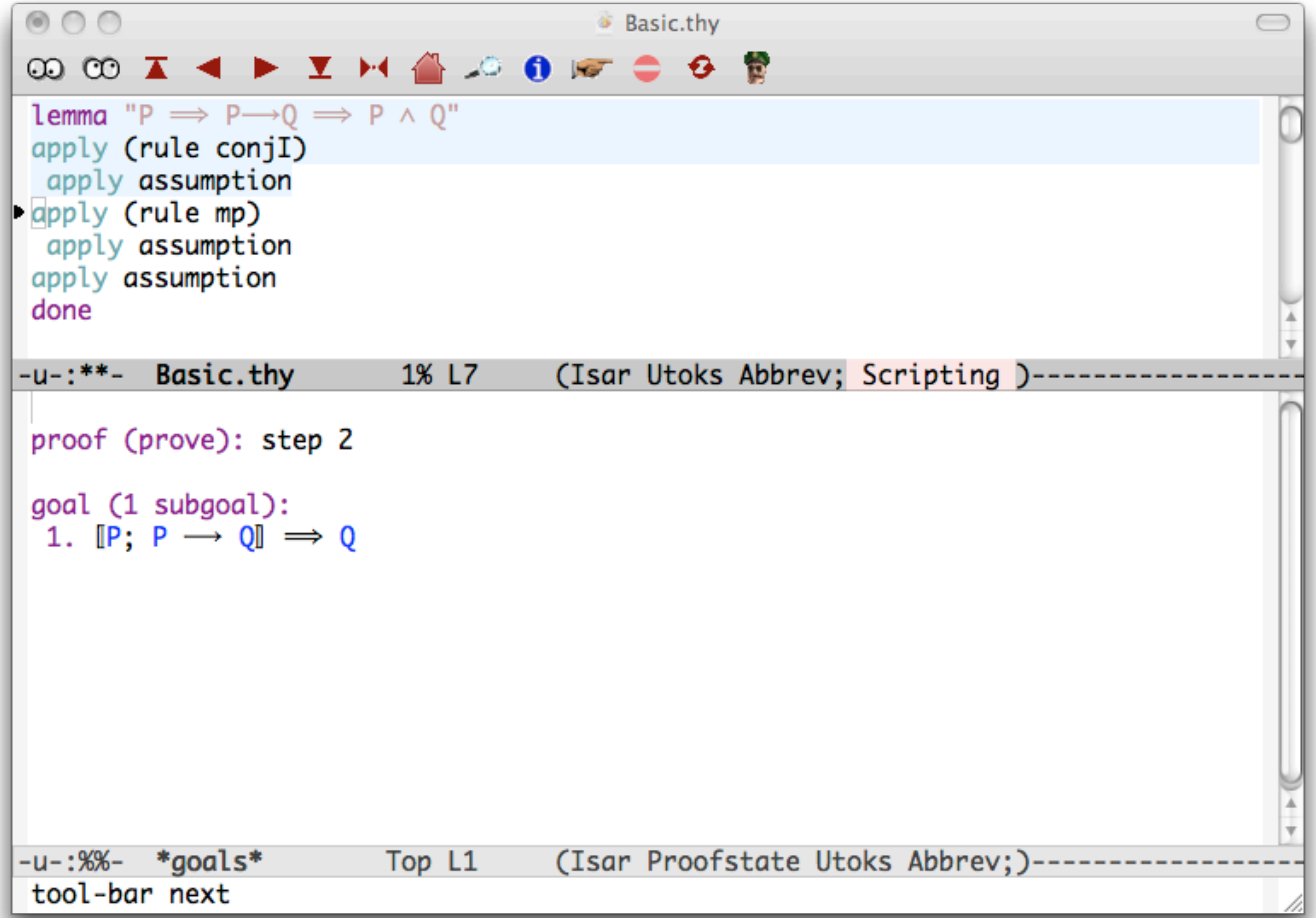
A Trivial Proof



The method "`rule`" is one of the most primitive in Isabelle. It matches the conclusion of the supplied rule with that of the a subgoal, which is replaced by new subgoals: the corresponding instances of the rule's premises. See the *Tutorial*, **5.7 Interlude: the Basic Methods for Rules**.

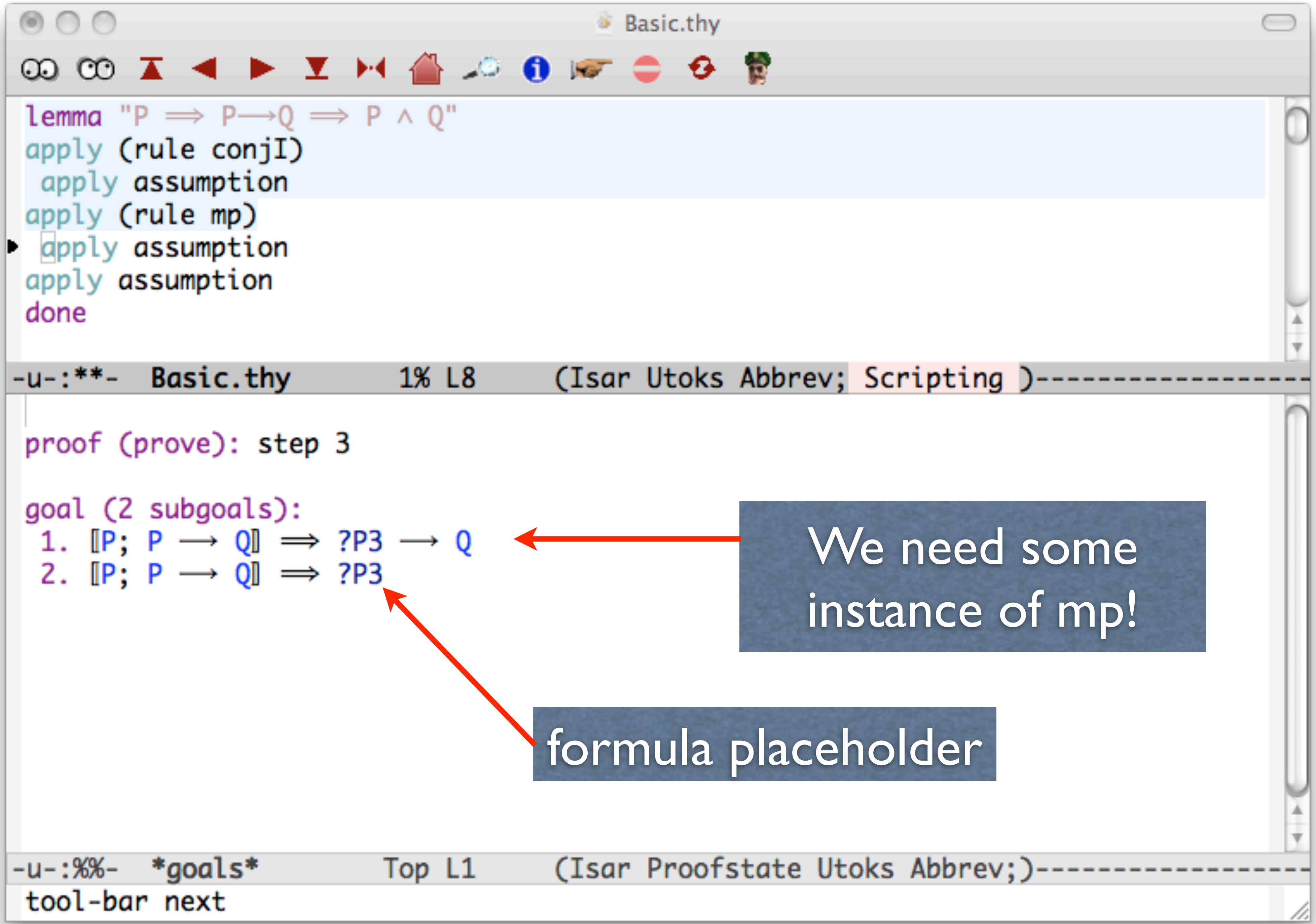
Normally, it applies to the first subgoal, though a specific goal number can be specified; many other proof methods follow the same convention.

Proof by Assumption



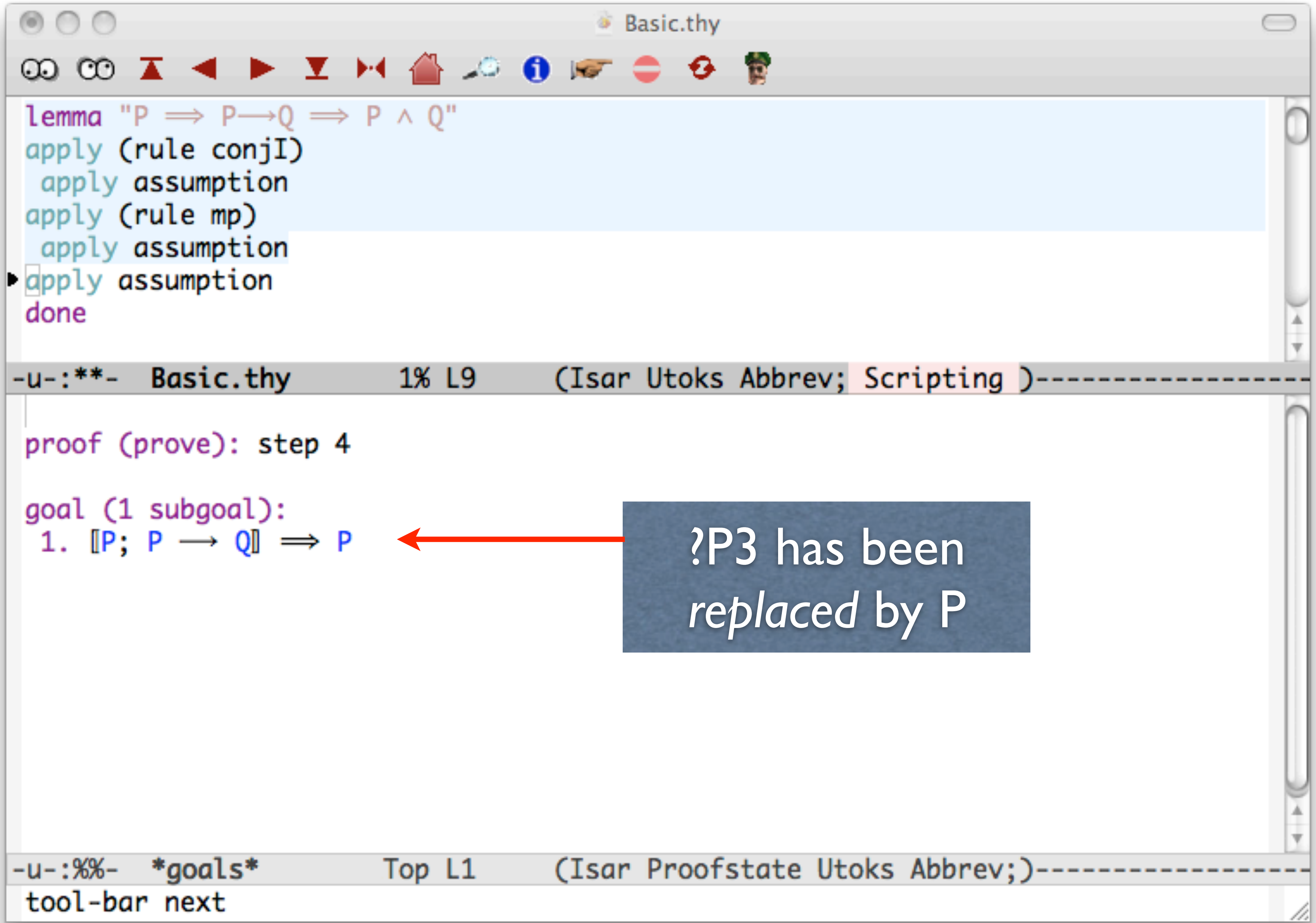
The method “`assumption`” is also primitive. It proves a subgoal if it can unify that subgoal’s conclusion with one of its premises. If successful, it deletes that subgoal.

Unknowns in Subgoals



Isabelle includes a class of variables whose names begin with the ? character. They are called unknowns or schematic variables. Logically, they are no different from ordinary free variables, but Isabelle treats them differently: it allows them to be replaced by other expressions during unification. Isabelle rewrite rules and inference rules contain many such variables, but we normally suppress the question marks to make them easier to read. For example, the rule `conjI` is really `?P ==> (?Q ==> ?P & ?Q)`.

Unknowns and Unification



Proving ?P3→Q from the assumption P→Q performs unification, and the variable ?P3 is updated. All occurrences of the variable are updated. In this way, proving one subgoal can make another subgoal impossible to prove. Sometimes there are multiple choices and only one will allow the proof to go through.

Discharging Assumptions

$$\dfrac{\begin{matrix}[P] \\ \vdots \\ Q\end{matrix}}{P \to Q}$$

```
(P ⇒ Q) ⇒ P→Q
```

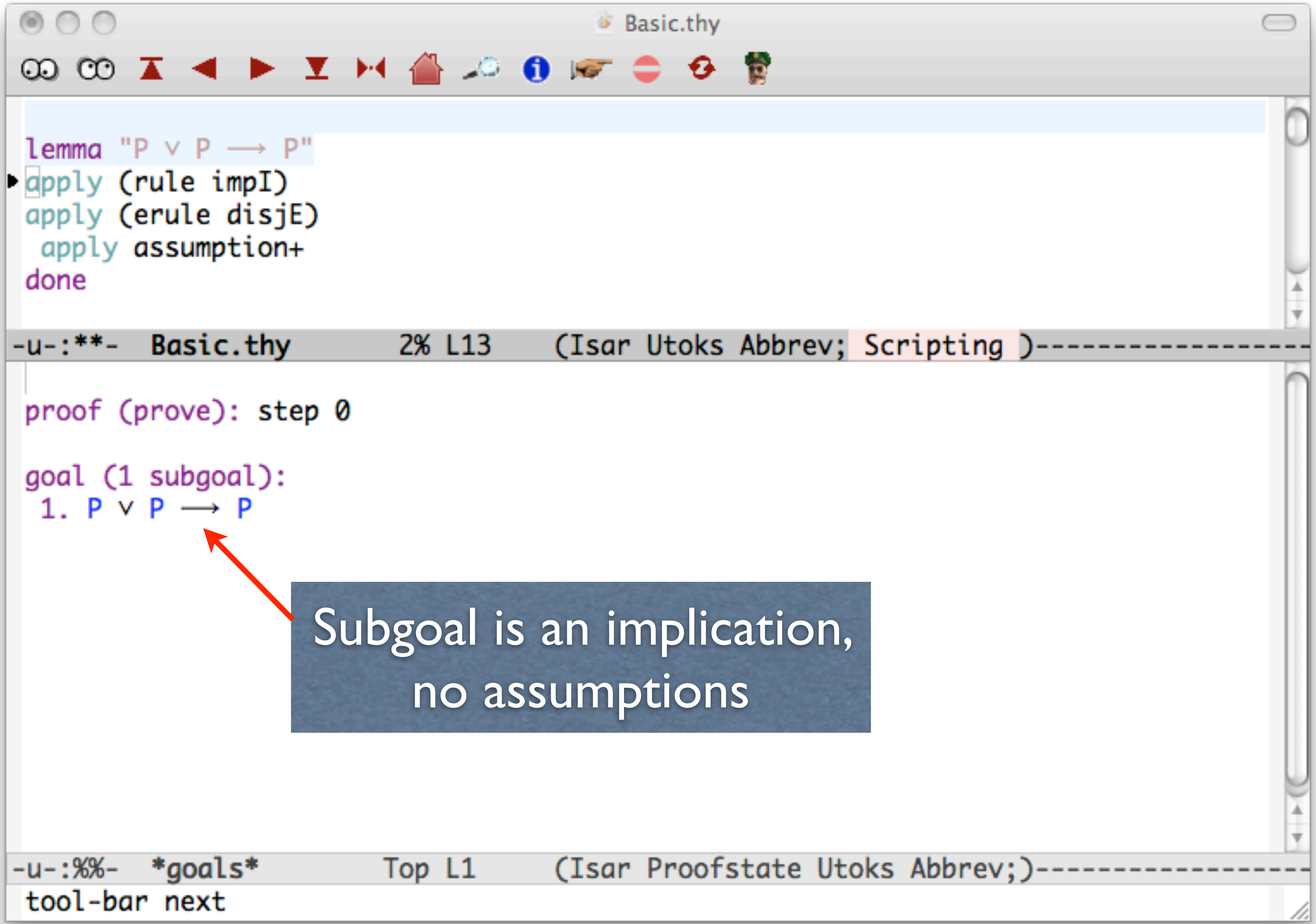
$$\dfrac{P \vee Q \qquad \begin{matrix}[P] \\ \vdots \\ R\end{matrix} \qquad \begin{matrix}[Q] \\ \vdots \\ R\end{matrix}}{R}$$

```
⟦P ∨ Q; P⇒R; Q⇒R⟧ ⇒ R
```

Such rules take derivations that depend upon particular assumptions (written as [P] and [Q] above) and "discharge" those assumptions, which means that the conclusion is not regarded as depending on them. The backwards interpretation is more natural: to prove P→Q, it suffices to assume P and prove Q.

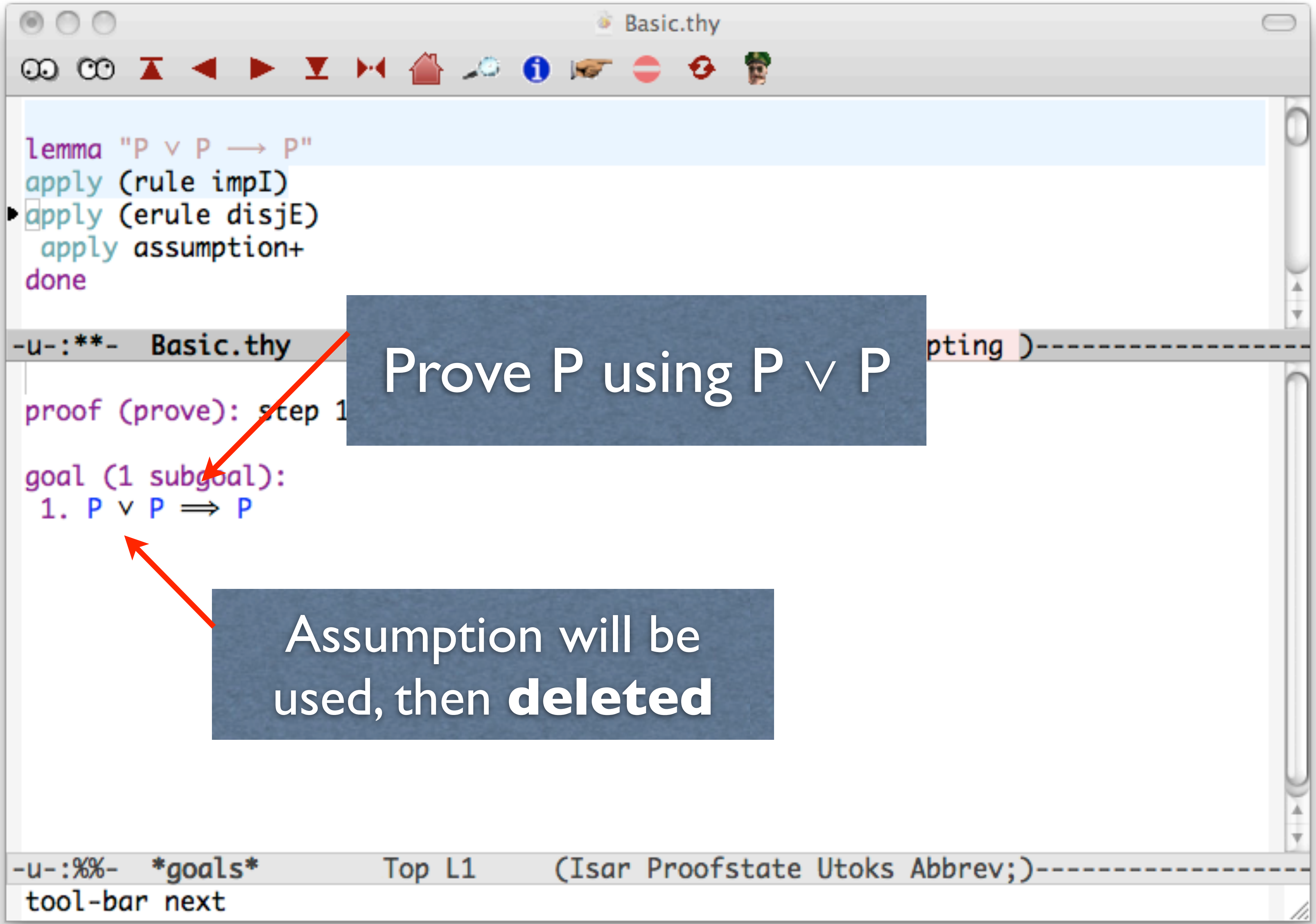
Meta-level implication (⇒) expresses the discharging of assumptions as well as the relationship between premises and conclusion.

A Proof using Assumptions

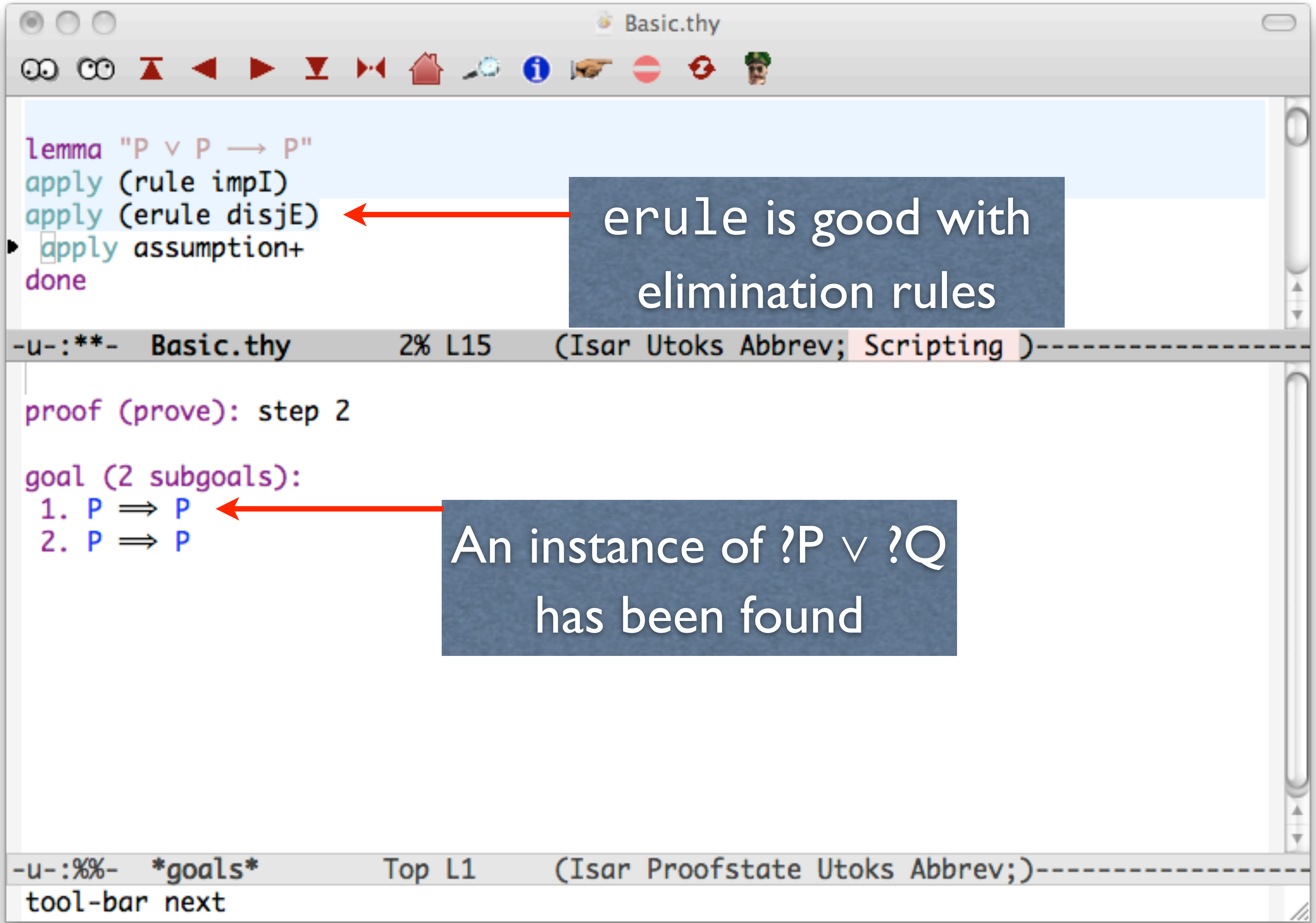


A full list of the predicate calculus inference rules for higher-order logic is available in Isabelle's Logics: HOL, a somewhat outdated but still useful reference manual.

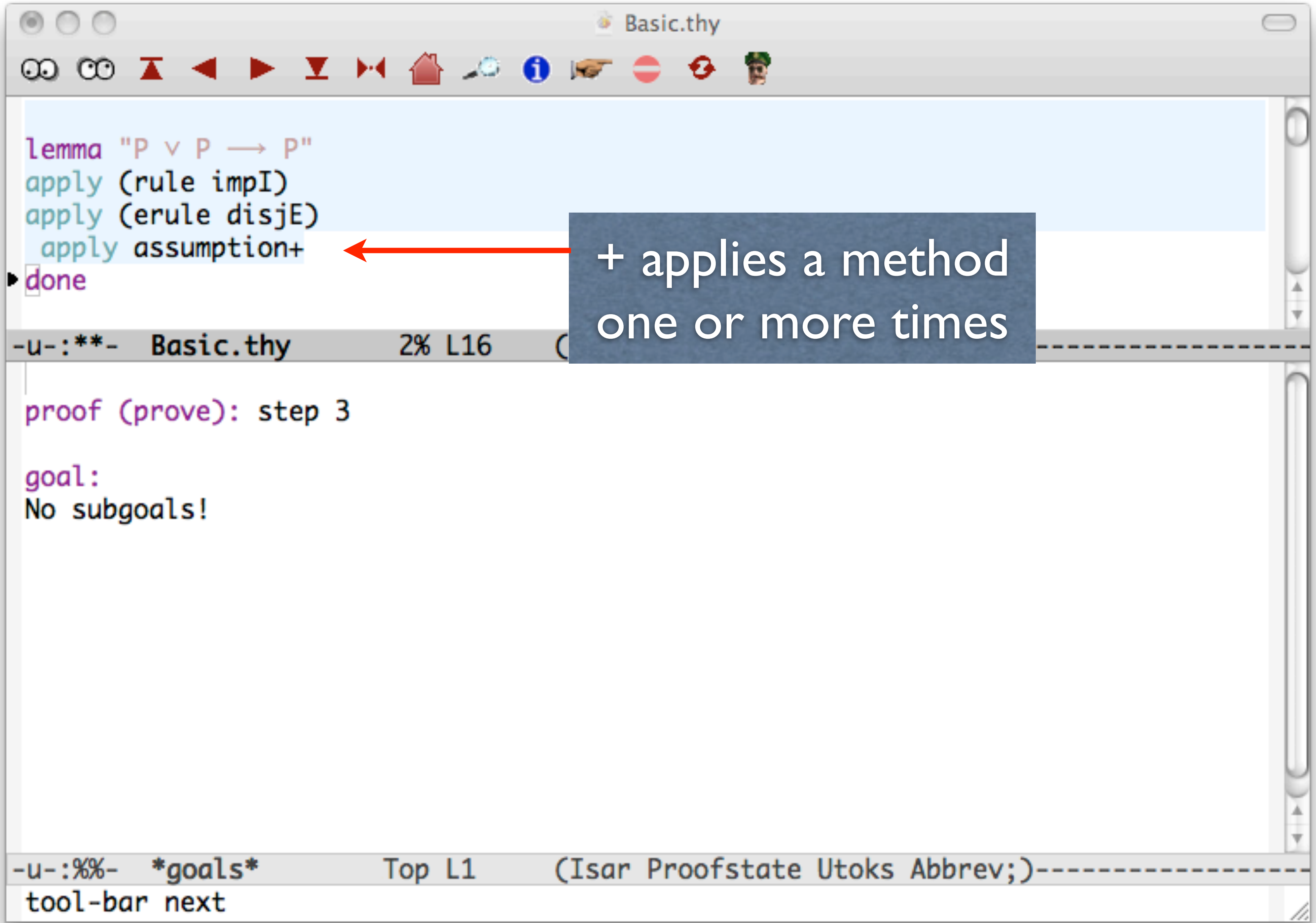
After Implies-Introduction



Disjunction Elimination



The Final Step



Quantifiers

$$\frac{P(t)}{\exists x.\, P(x)}$$

`P(x) ⇒ ∃x.P(x)`

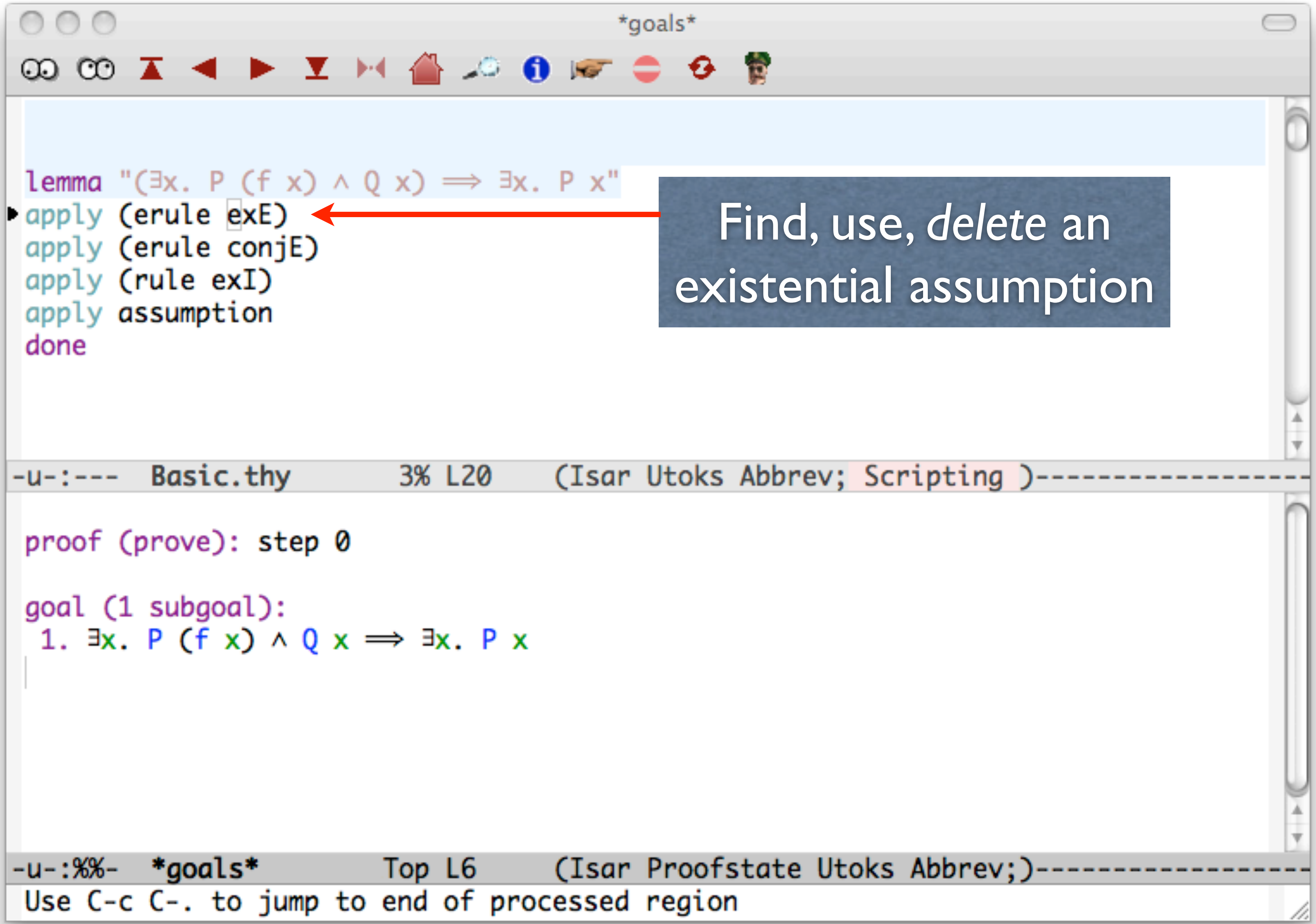
$$\frac{\exists x.\, P(x) \qquad \begin{array}{c}[P(x)]\\ \vdots \\ Q\end{array}}{Q}$$

`⟦∃x.P(x); ⋀x.P(x)⇒Q⟧ ⇒ Q`

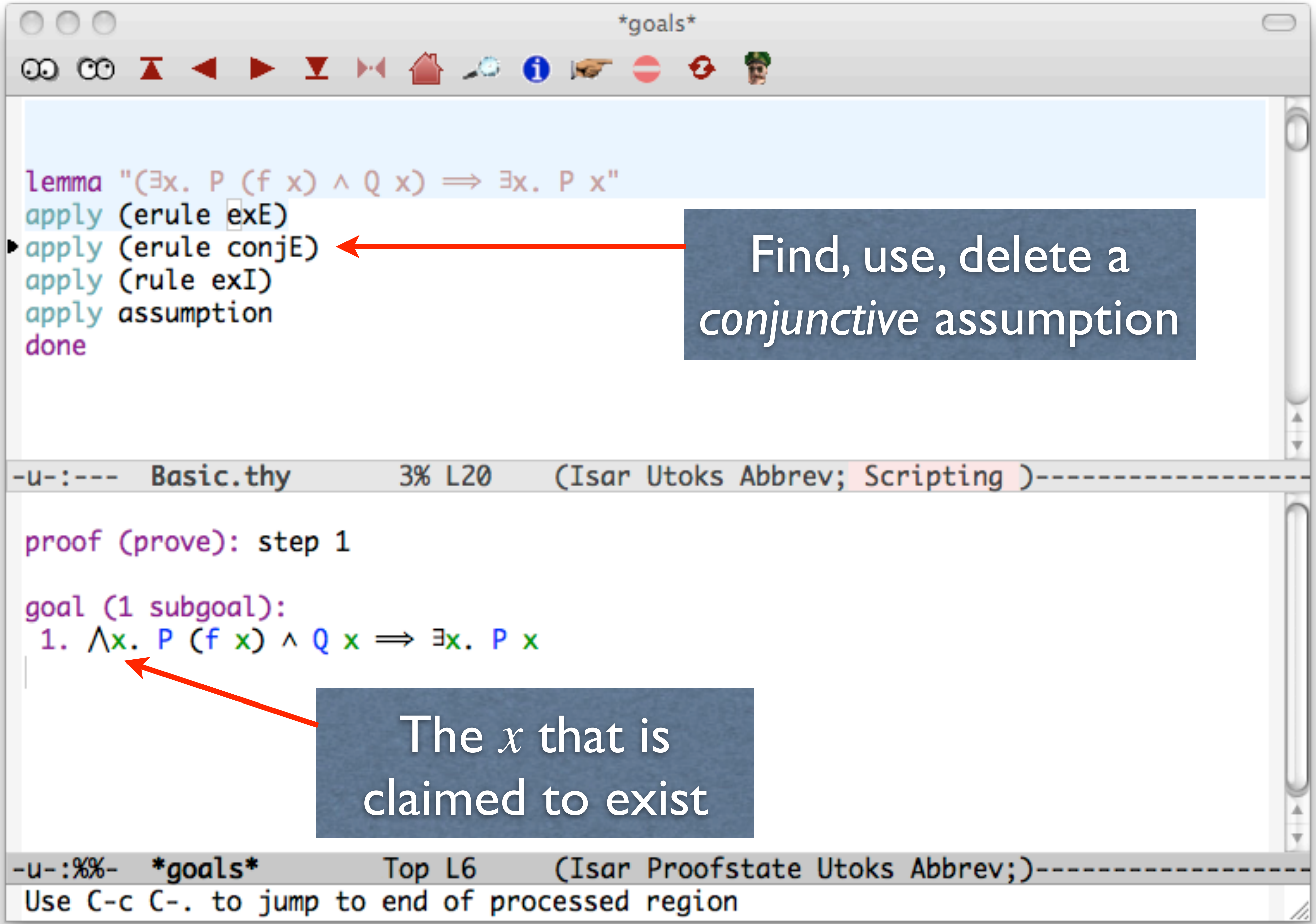
meta-universal quantifier
states the variable condition

Isabelle's logical framework includes the typed lambda calculus, so quantifiers can be declared as constants of appropriate type. Variable-binding syntax can also be specified.

A Tiny Quantifier Proof

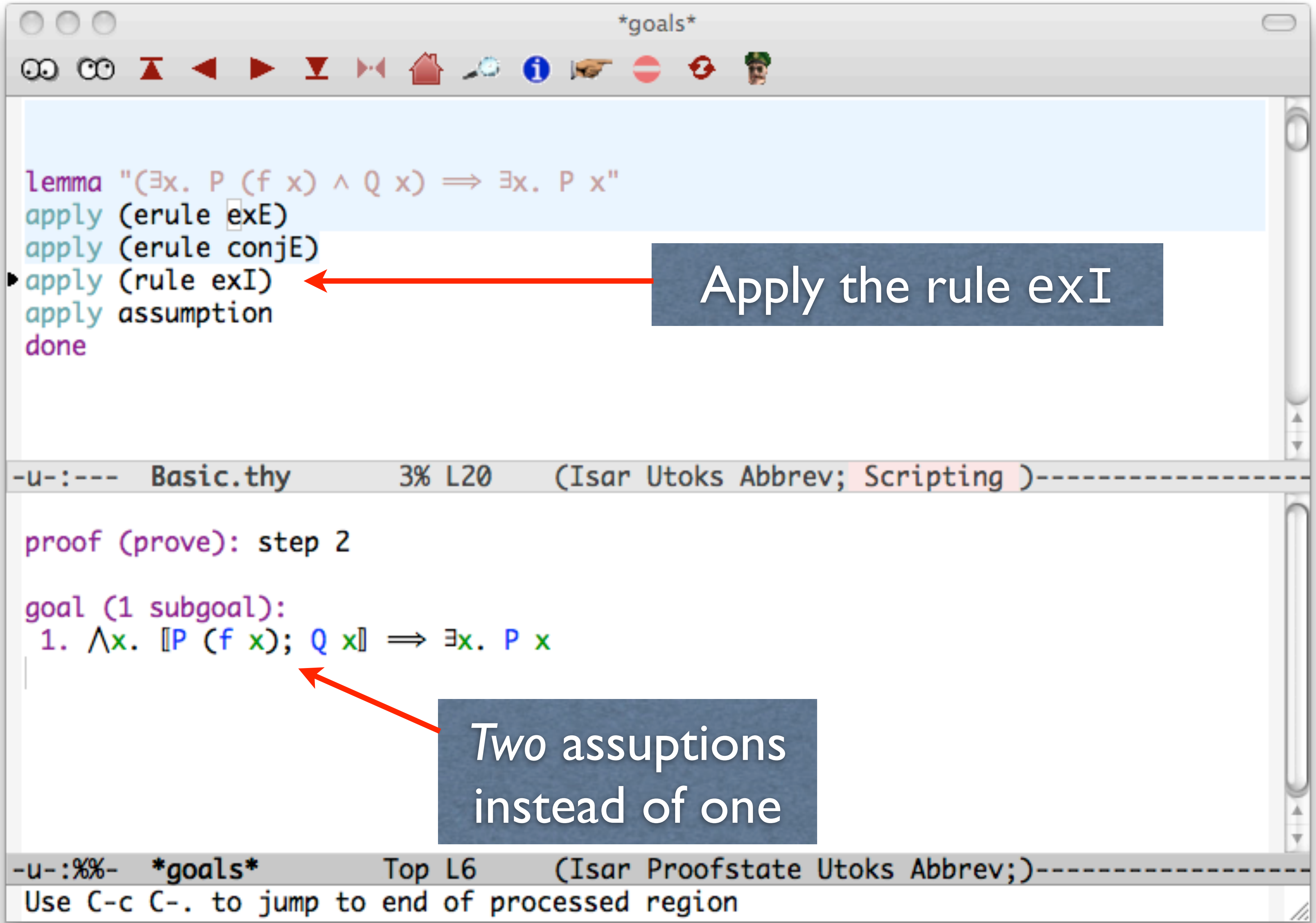


Conjunction Elimination

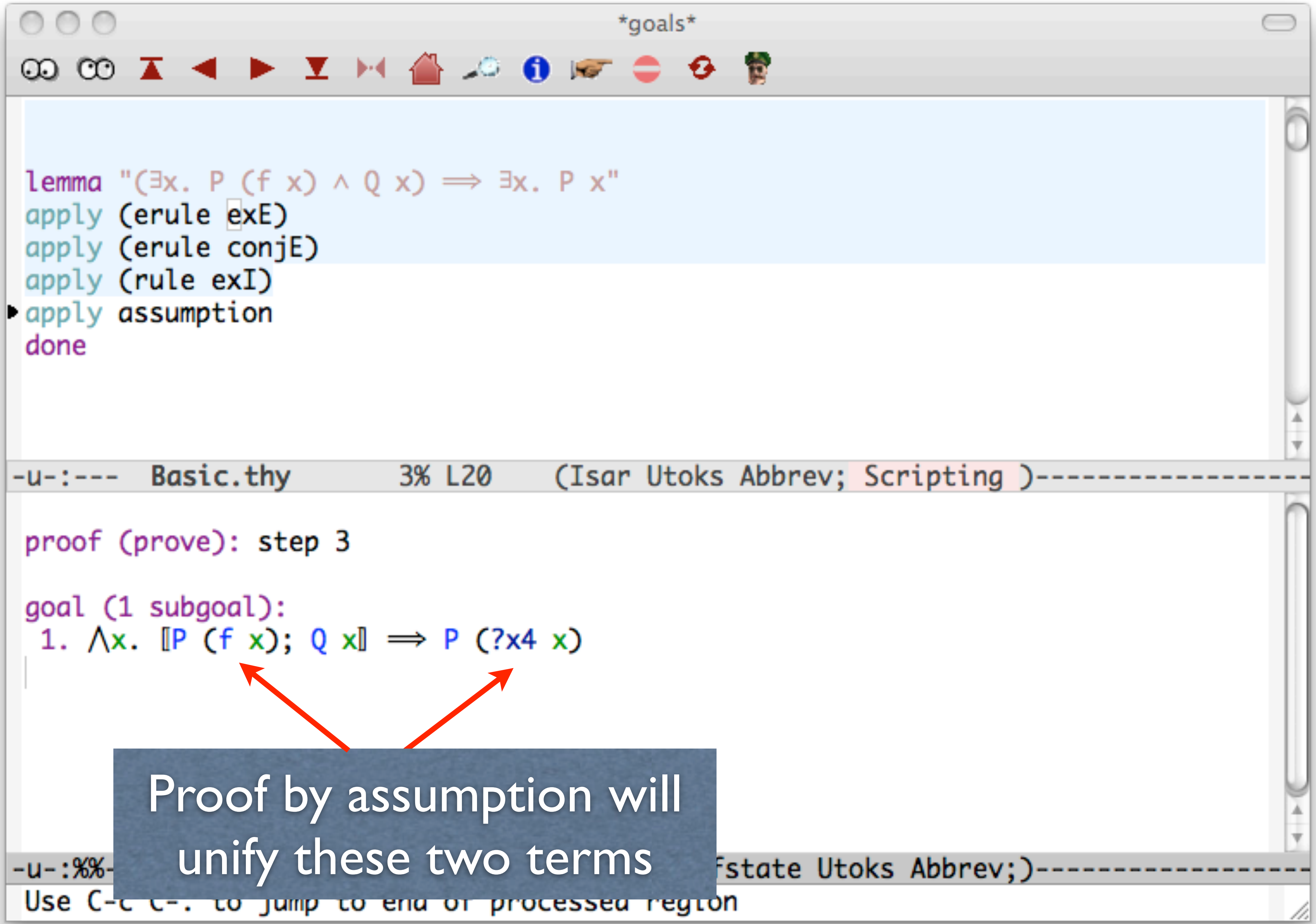


The proof above refers to `conjE`, which is an alternative to the rules `conjunct1` and `conjunct2`. It has the standard elimination format (shared with disjunction elimination and existential elimination), so it can be used with the method `erule`.

Now for ∃-Introduction

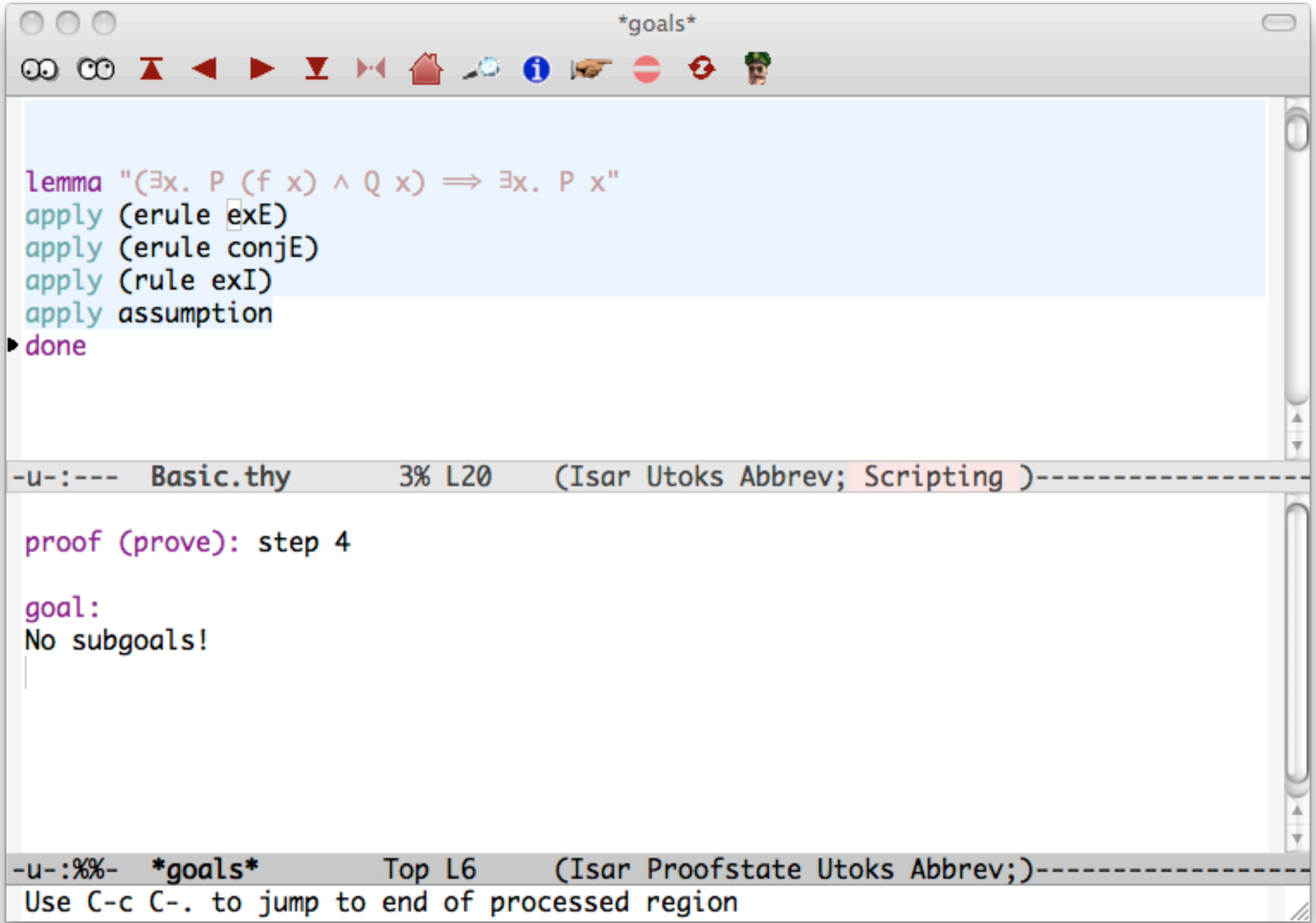


An Unknown for the Witness



A proof of existence normally requires a witness, namely a specific term satisfying the required property. Isabelle allows this choice to be deferred. The structure of the term, in this case `?x4 x`, holds information about which bound variables may appear in the witness. Here, `This is` may appear in the witness.

Done!


goals
lemma "(∃x. P (f x) ∧ Q x) ⟹ ∃x. P x"
apply (erule exE)
apply (erule conjE)
apply (rule exI)
apply assumption
done
-u-:--- Basic.thy 3% L20 (Isar Utoks Abbrev; Scripting)
proof (prove): step 4
goal:
No subgoals!
-u-:%%- *goals* Top L6 (Isar Proofstate Utoks Abbrev;)
Use C-c C-. to jump to end of processed region

Interactive Formal Verification
6: Sets

Lawrence C Paulson
Computer Laboratory
University of Cambridge

Set Notation in Isabelle

- Set notation is crucial to mathematical discourse.
- Set-theoretic abstractions naturally express many complex constructions.
- A set in high-order logic is a *boolean-valued map*.
- The elements of such a set must all have the *same type*...
- and we have the *universal set* for each type.

See the *Tutorial*, section 6.1 Sets.

Set Theory Primitives

$$\begin{aligned}
e \in \{x.\, P(x)\} &\iff P(e) \\
e \in \{x \in A.\, P(x)\} &\iff e \in A \wedge P(e) \\
e \in -A &\iff e \notin A \\
e \in A \cup B &\iff e \in A \vee e \in B \\
e \in A \cap B &\iff e \in A \wedge e \in B \\
e \in \texttt{Pow}(A) &\iff e \subseteq A
\end{aligned}$$

Please note that we do not write {xIP(x)}. Isabelle would interpret the I as expressing disjunction and the expression as denoting the singleton set containing the element xIP(x)!

Big Union and Intersection

$$
\begin{aligned}
e \in \left(\bigcup x.\, B(x)\right) &\iff \exists x.\, e \in B(x) \\
e \in \left(\bigcup x \in A.\, B(x)\right) &\iff \exists x \in A.\, e \in B(x) \\
e \in \bigcup A &\iff \exists x \in A.\, e \in x
\end{aligned}
$$

And the analogous forms of intersections...

Functions

$$e \in (f`A) \iff \exists x \in A.\, e = f(x)$$

$$e \in (f-`A) \iff f(e) \in A$$

$$f(x{:=}y) = (\lambda z.\ \texttt{if}\ z = x\ \texttt{then}\ y\ \texttt{else}\ f(z))$$

- Also `inj`, `surj`, `bij`, `inv`, etc. (injective,...)
- Don't *re-invent* image and inverse image!!

Inverse image is also known as pre-image. Using the actual image primitives gives access to the many theorems proved about them.

Finite Sets

$$\{a_1, \ldots, a_n\} = \texttt{insert}(a_1, \ldots, \texttt{insert}(a_n, \{\}))$$

$$e \in \texttt{insert}(a, B) \iff e = a \lor e \in B$$

$$\texttt{finite}(A \cup B) = (\texttt{finite}\, A \land \texttt{finite}\, B)$$

$$\texttt{finite}\, A \Longrightarrow \texttt{card}(\texttt{Pow}\, A) = 2^{\texttt{card}\, A}$$

Finite sets can be written explicitly, enumerating their elements in the obvious way. The notion of finiteness is also available, along with the notion of cardinality.

Intervals, Sums and Products

```
{..<u} == {x. x < u}
 {..u} == {x. x ≤ u}
{l<..} == {x. l<x}
 {l..} == {x. l≤x}
{l<..<u} == {l<..} ∩ {..<u}
 {l..<u} == {l..} ∩ {..<u}
```

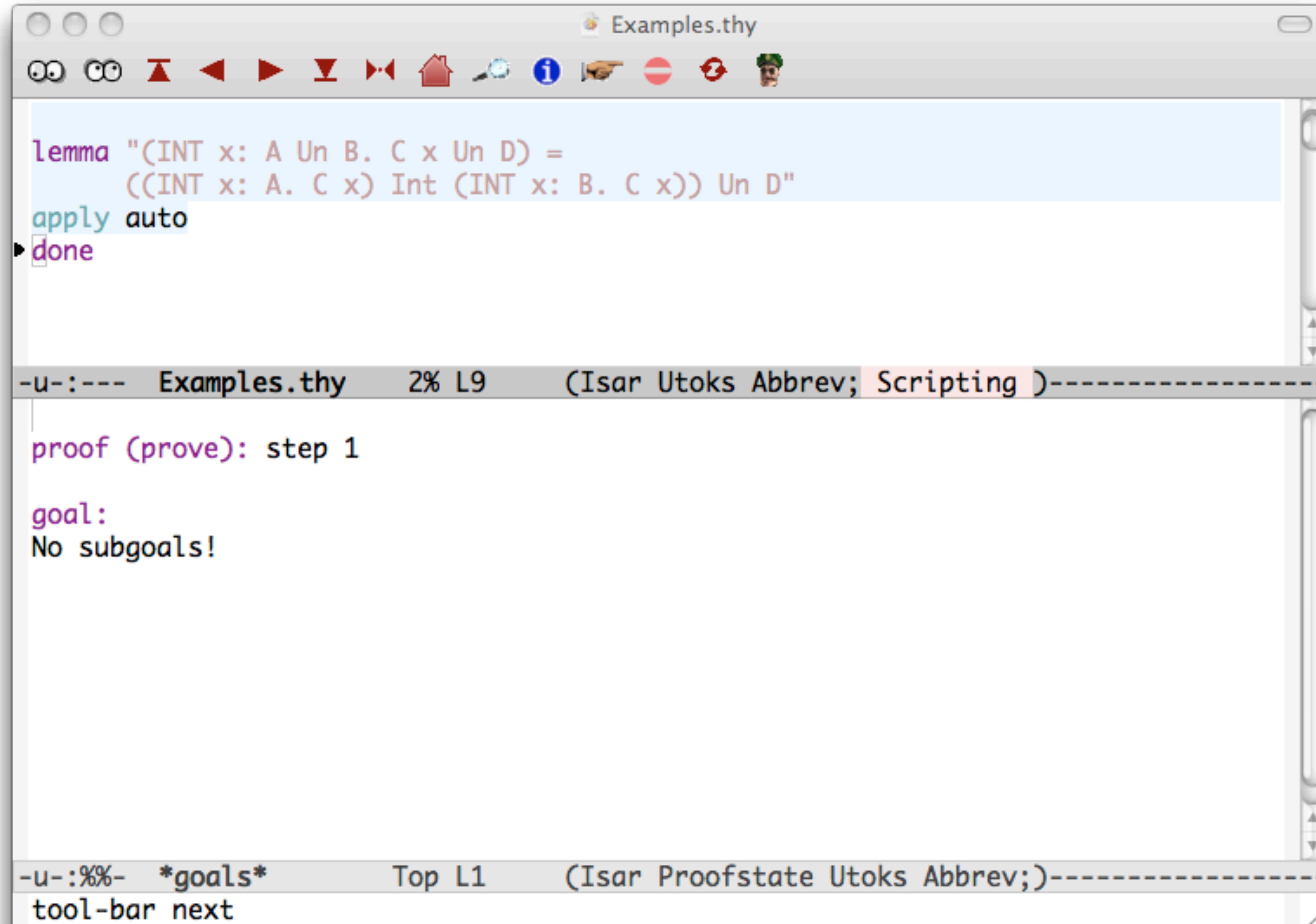
`setsum f A` and `setprod f A`

$\sum i \in I.\ f$ and $\prod i \in I.\ f$

Isabelle provides syntax for bounded and unbounded intervals. These are polymorphic: they are defined over all types that admit an ordering, and in particular they are applicable to intervals over the natural numbers, integers, rationals or reals.

Sums and products of functions over finite sets can also be written.

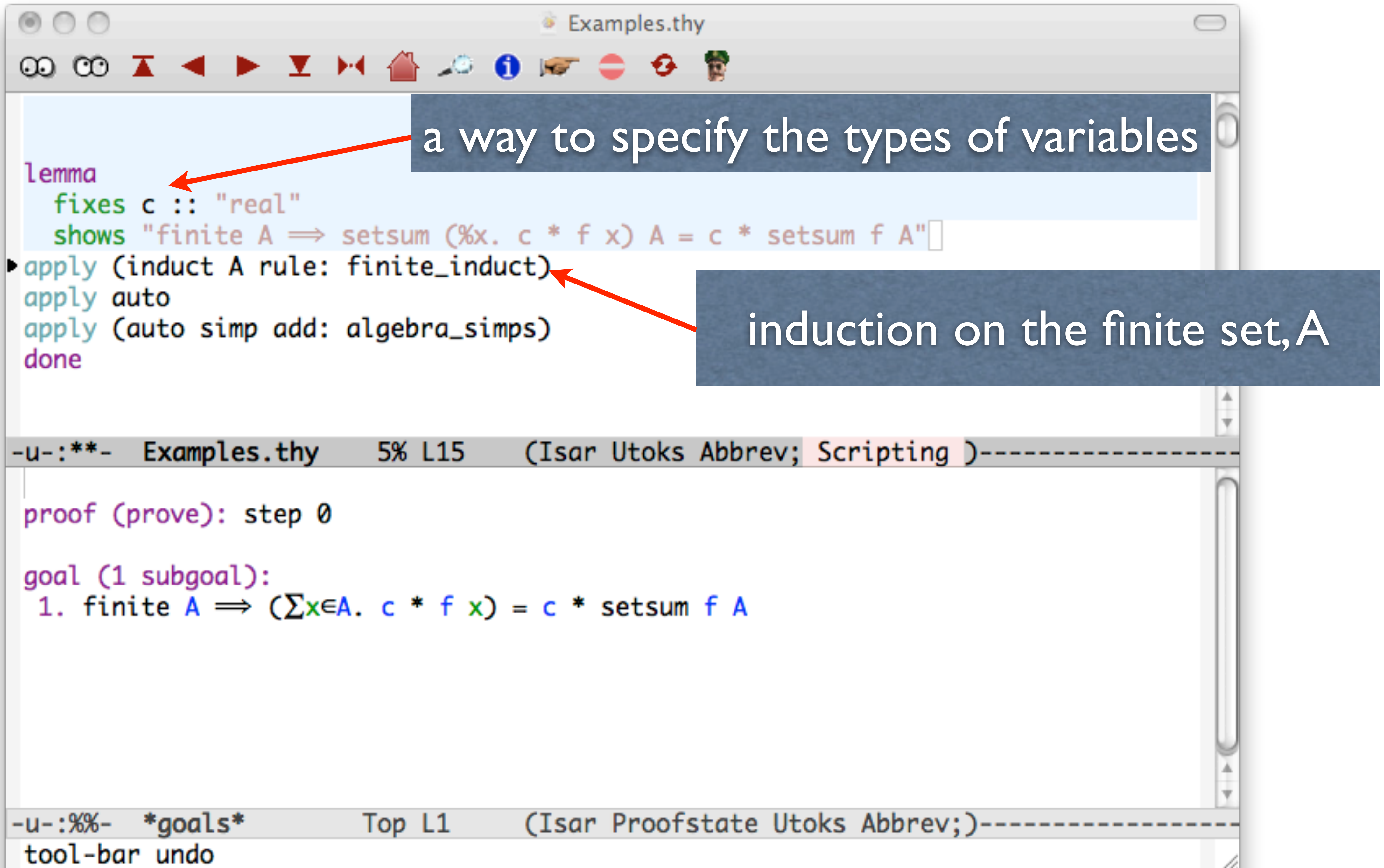
A Simple Set Theory Proof



Special symbols can be inserted using Proof General's maths menu. ASCII can simply be typed.

The main point of this example is that many such proofs are trivial, using `auto` or other automatic proof methods.

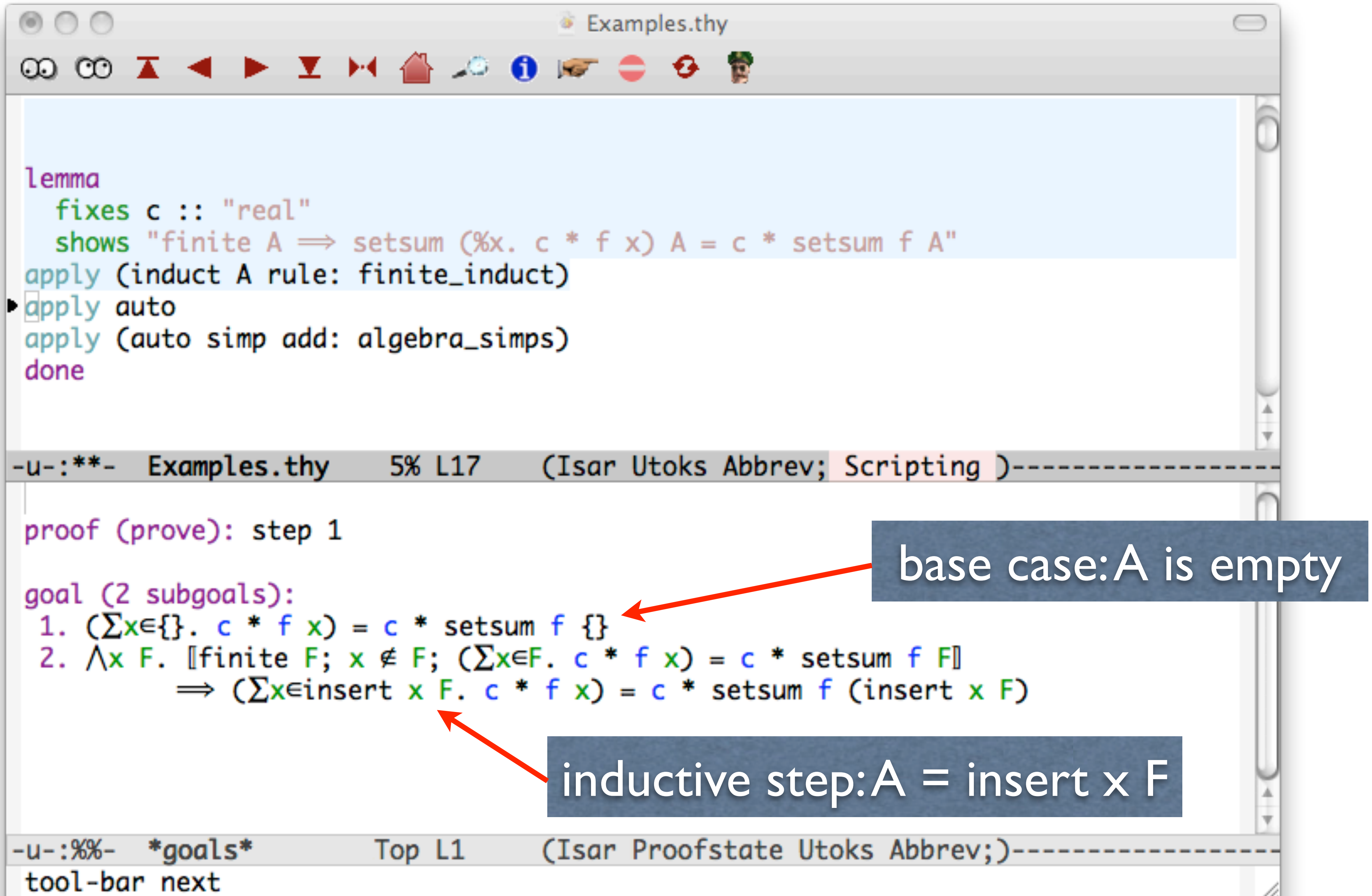
A Harder Proof Involving Sets



This example needs a type constraint because arithmetic concepts such as sum and product are heavily overloaded. If you use `fixes`, then you must also use `shows`!

Isabelle's type classes allow this theorem to be proved in an overloaded form, but for simplicity here we restrict ourselves to type `real`.

Outcome of the Induction

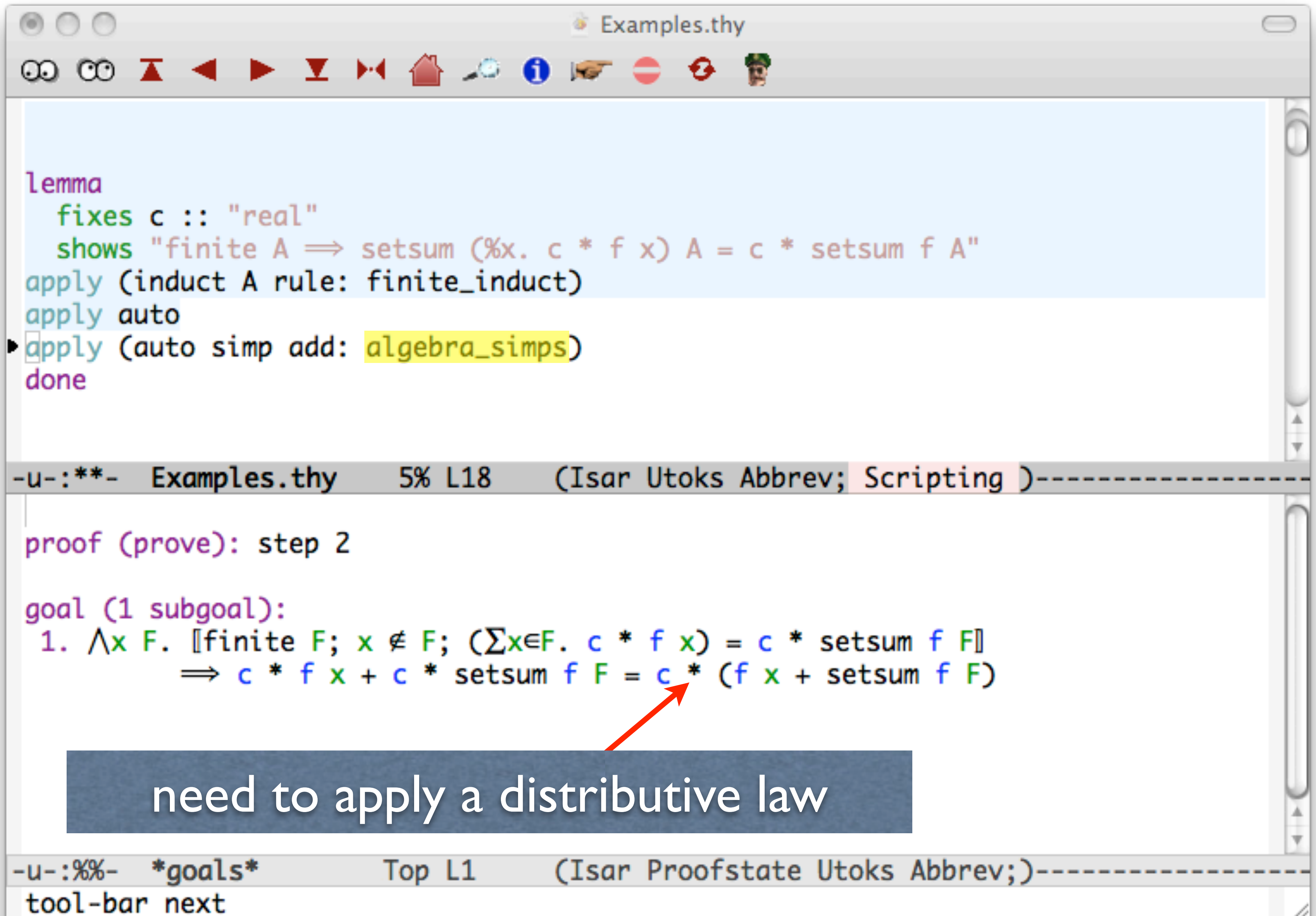


The base case is trivial, because both sides of the equality clearly equal zero. In the induction step, the induction hypothesis (which concerns the set `F`) will be applicable, because

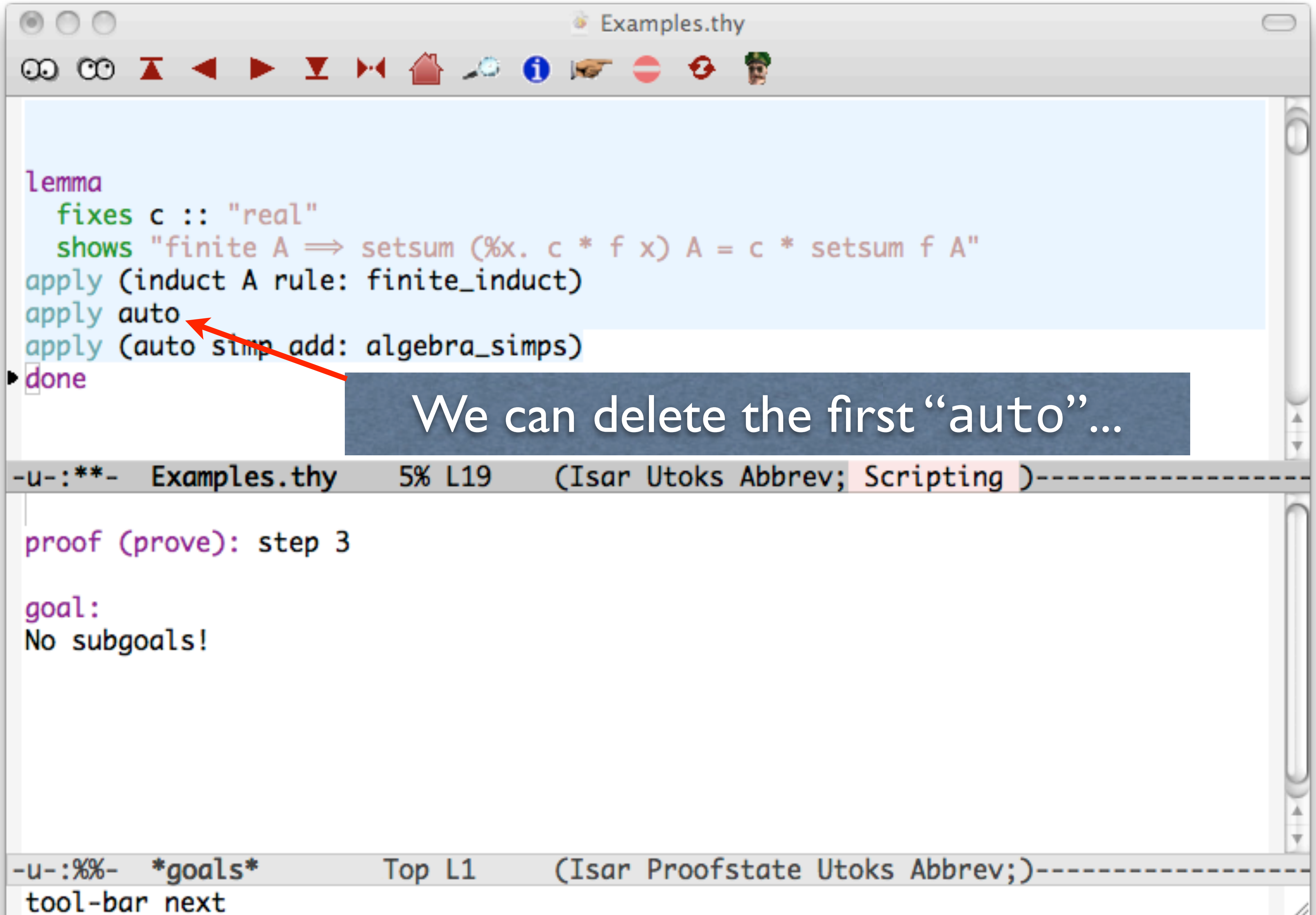
```
setsum f (insert a F) = f a + setsum f F
```

Note that Isabelle uses a fancy notation for summations, but only if the body of the summation is nontrivial.

Almost There!



Finished!

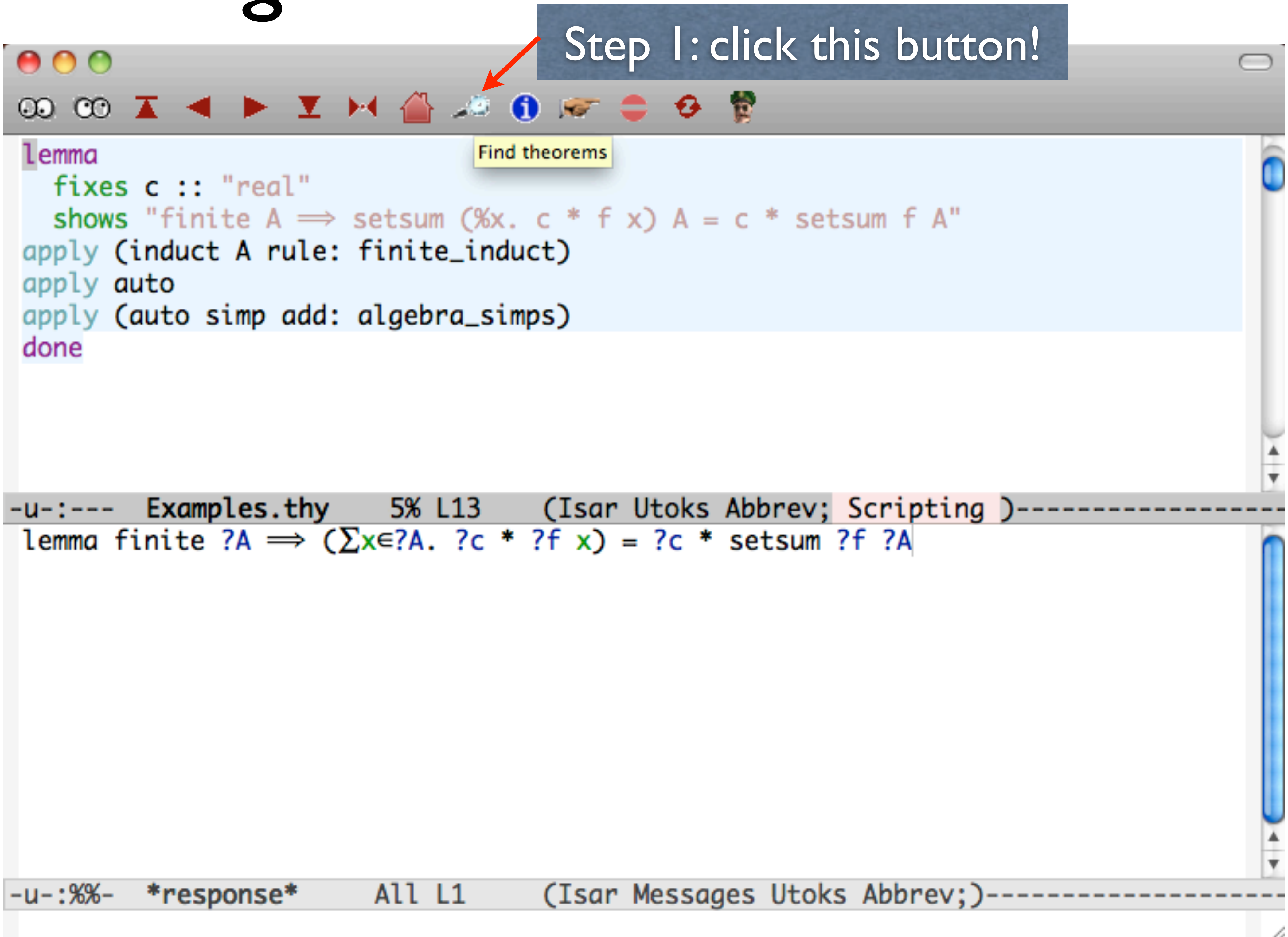


Recall that `algebra_simps` is a list of simplification rules for multiplying out algebraic expressions.

Proving Theorems about Sets

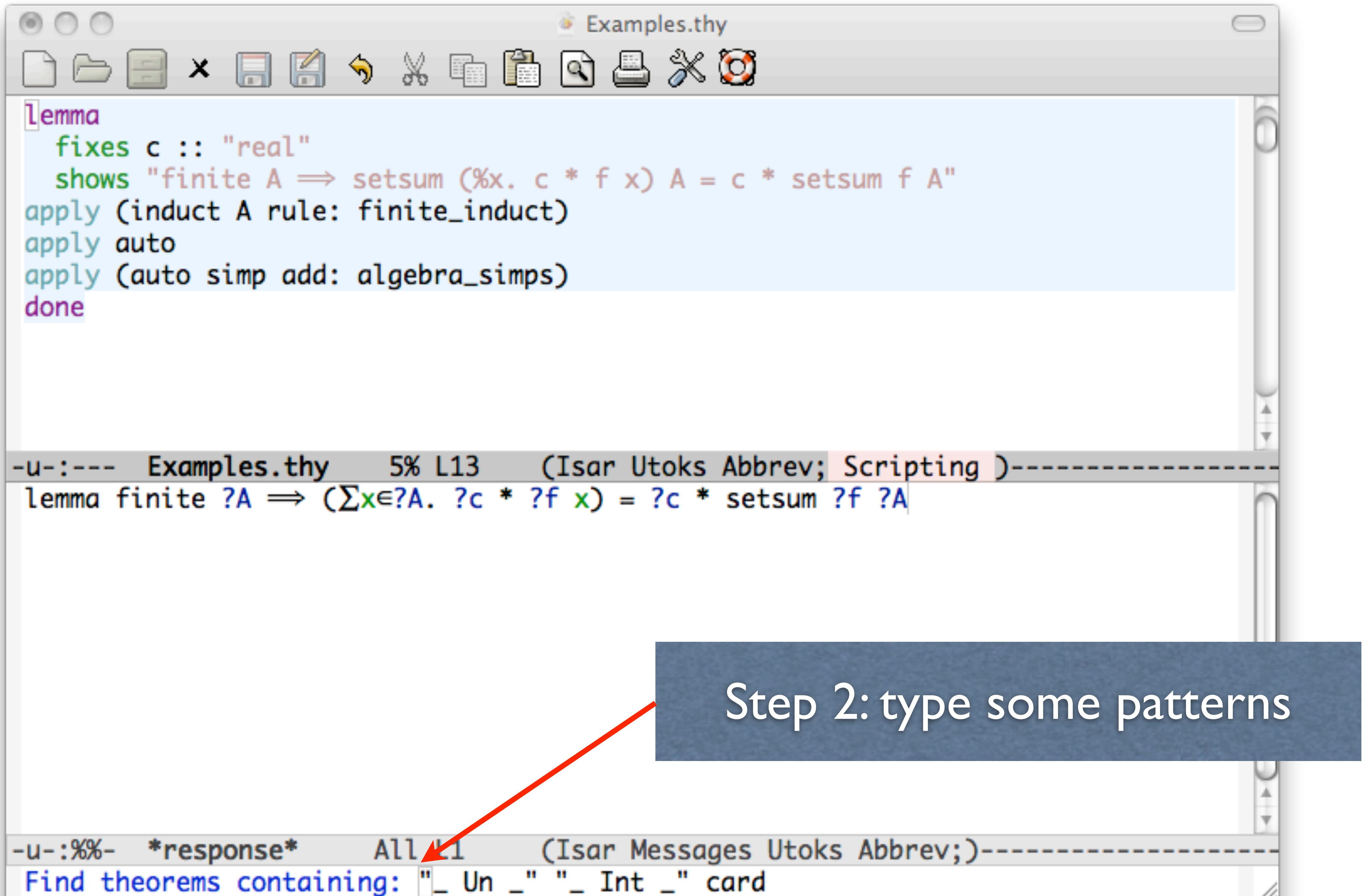
- It is not practical to learn all the built-in lemmas.
- Instead, try an automatic proof method:
  - `auto`
  - `force`
  - `blast`
- Each uses the built-in library, comprising hundreds of facts, with powerful heuristics.

Finding Theorems about Sets



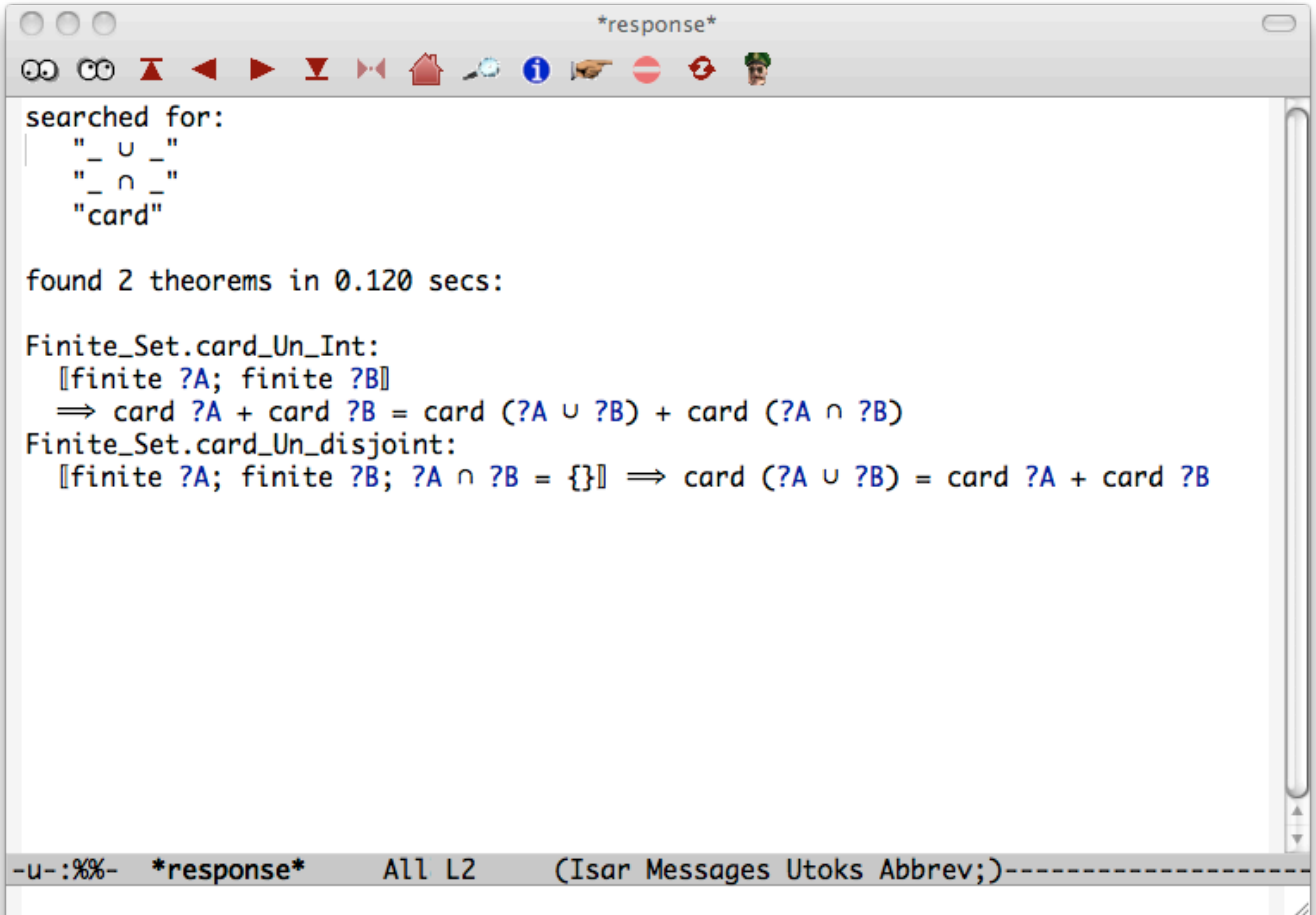
See the *Tutorial*, section **3.1.11 Finding Theorems**. Virtually all theorems loaded within Isabelle can be located using this function. Unfortunately, it does not locate theorems that are proved in external libraries.

Finding Theorems about Sets



The easiest way to refer to infix operators is by entering small patterns, as shown above. More complex patterns are also permitted. The constraints are treated conjunctively: use additional constraints if you get too many results, and fewer constraints if you get no results.

The Results!


response
searched for:
"_ ∪ _"
"_ ∩ _"
"card"
found 2 theorems in 0.120 secs:
Finite_Set.card_Un_Int:
⟦finite ?A; finite ?B⟧
⟹ card ?A + card ?B = card (?A ∪ ?B) + card (?A ∩ ?B)
Finite_Set.card_Un_disjoint:
⟦finite ?A; finite ?B; ?A ∩ ?B = {}⟧ ⟹ card (?A ∪ ?B) = card ?A + card ?B
-u-:%%- *response* All L2 (Isar Messages Utoks Abbrev;)------------------

Interactive Formal Verification
7: Inductive Definitions

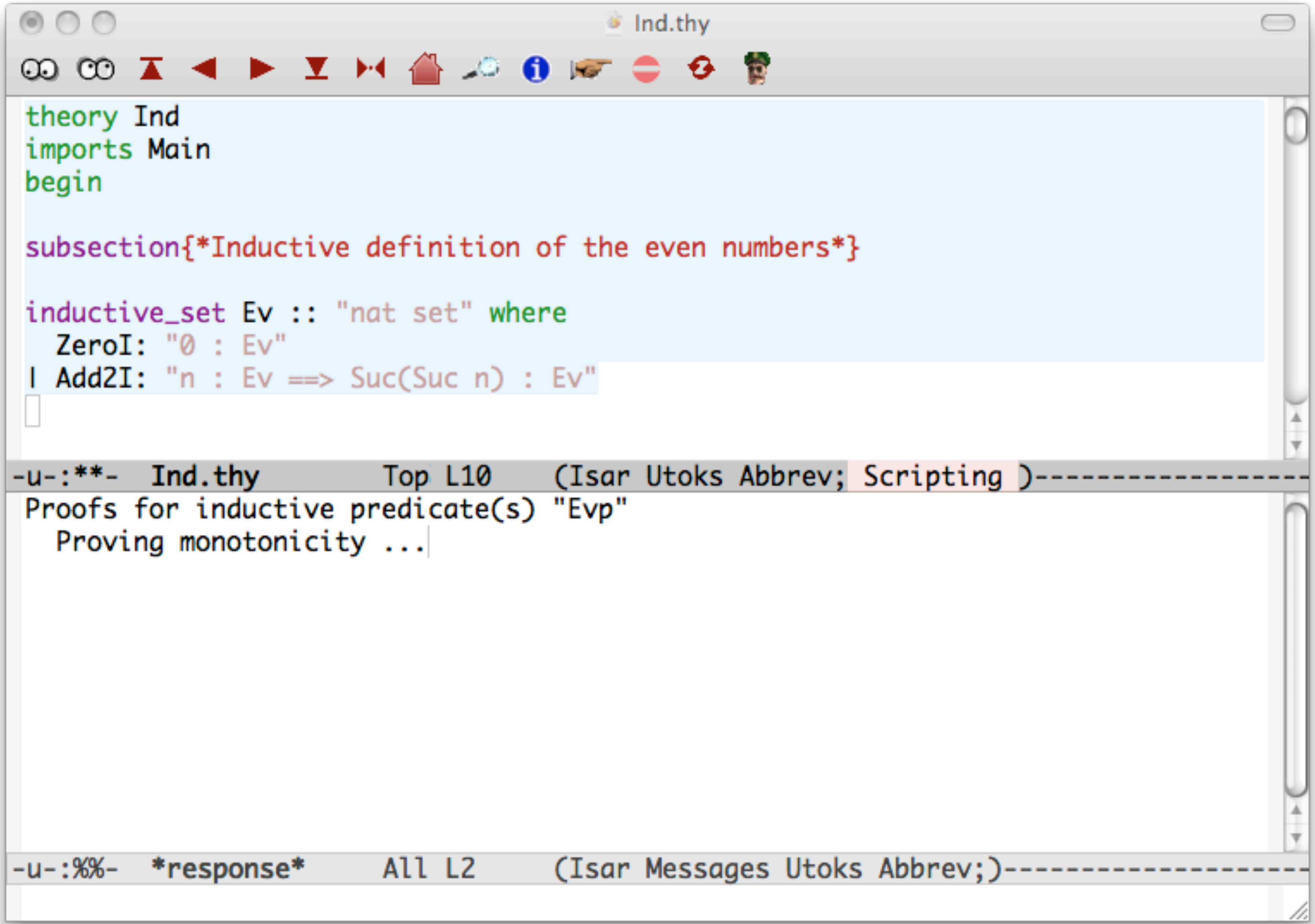
Lawrence C Paulson
Computer Laboratory
University of Cambridge

Defining a Set Inductively

- The set of even numbers is the least set such that
  - 0 is even.
  - If *n* is even, then *n*+2 is even.
- These can be viewed as *introduction rules*.
- We get an *induction principle* to express that no other numbers are even.
- Induction is used throughout mathematics, and to express the semantics of programming languages.

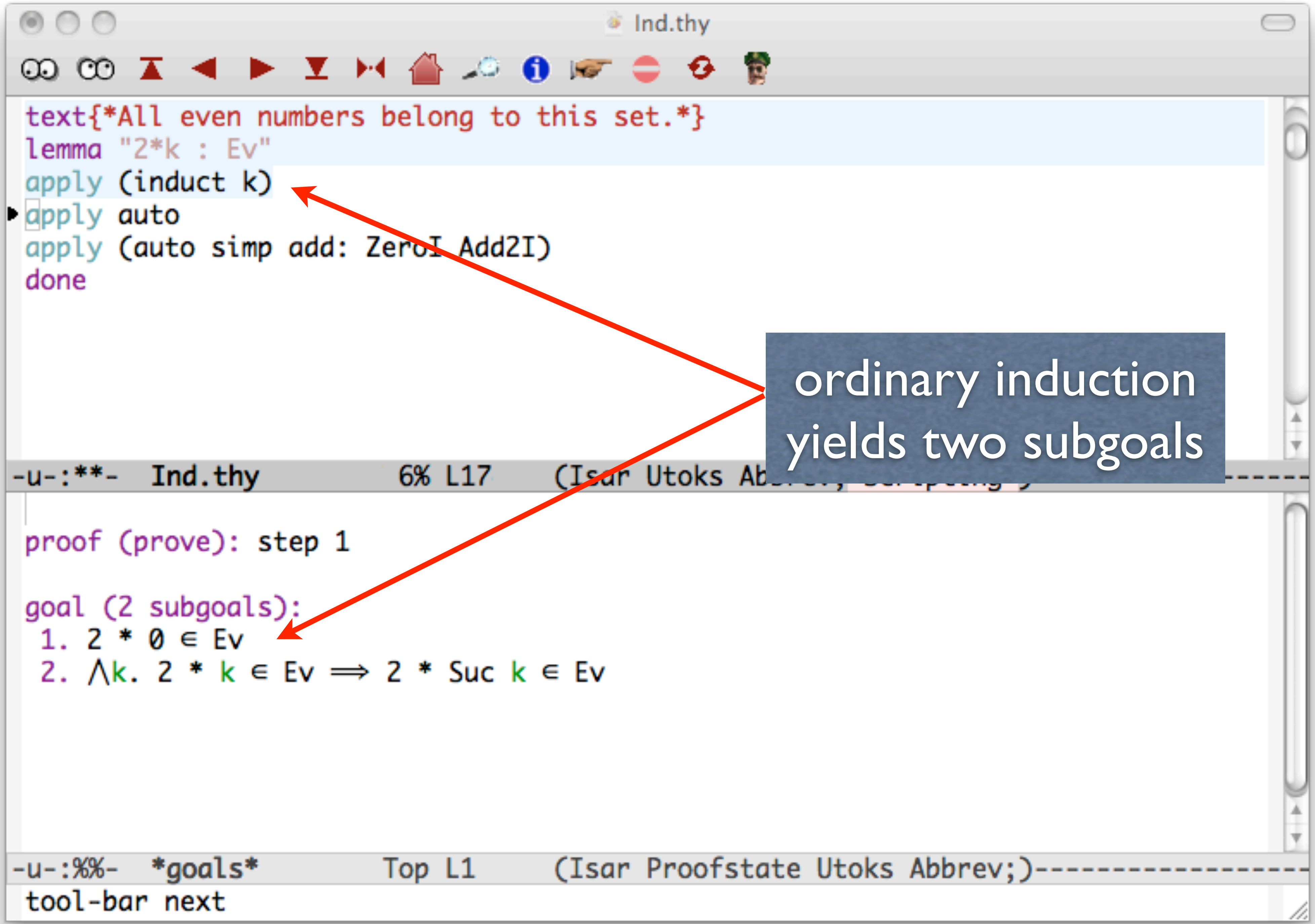
See the *Tutorial*, Chaper 7. **Inductively Defined Sets**.

Inductive Definitions in Isabelle

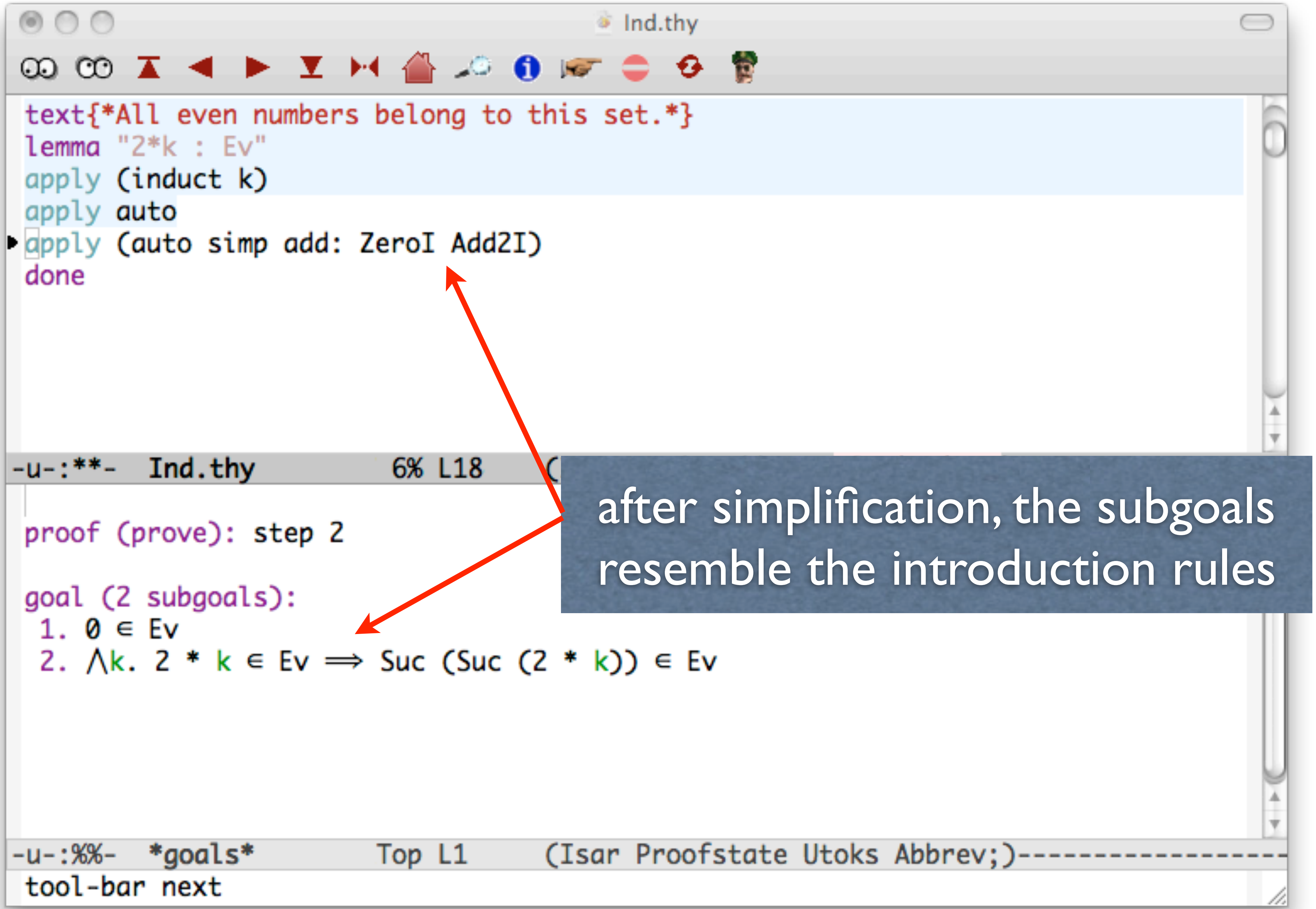


The *Tutorial* discusses precisely the same example, section 7.1.1.

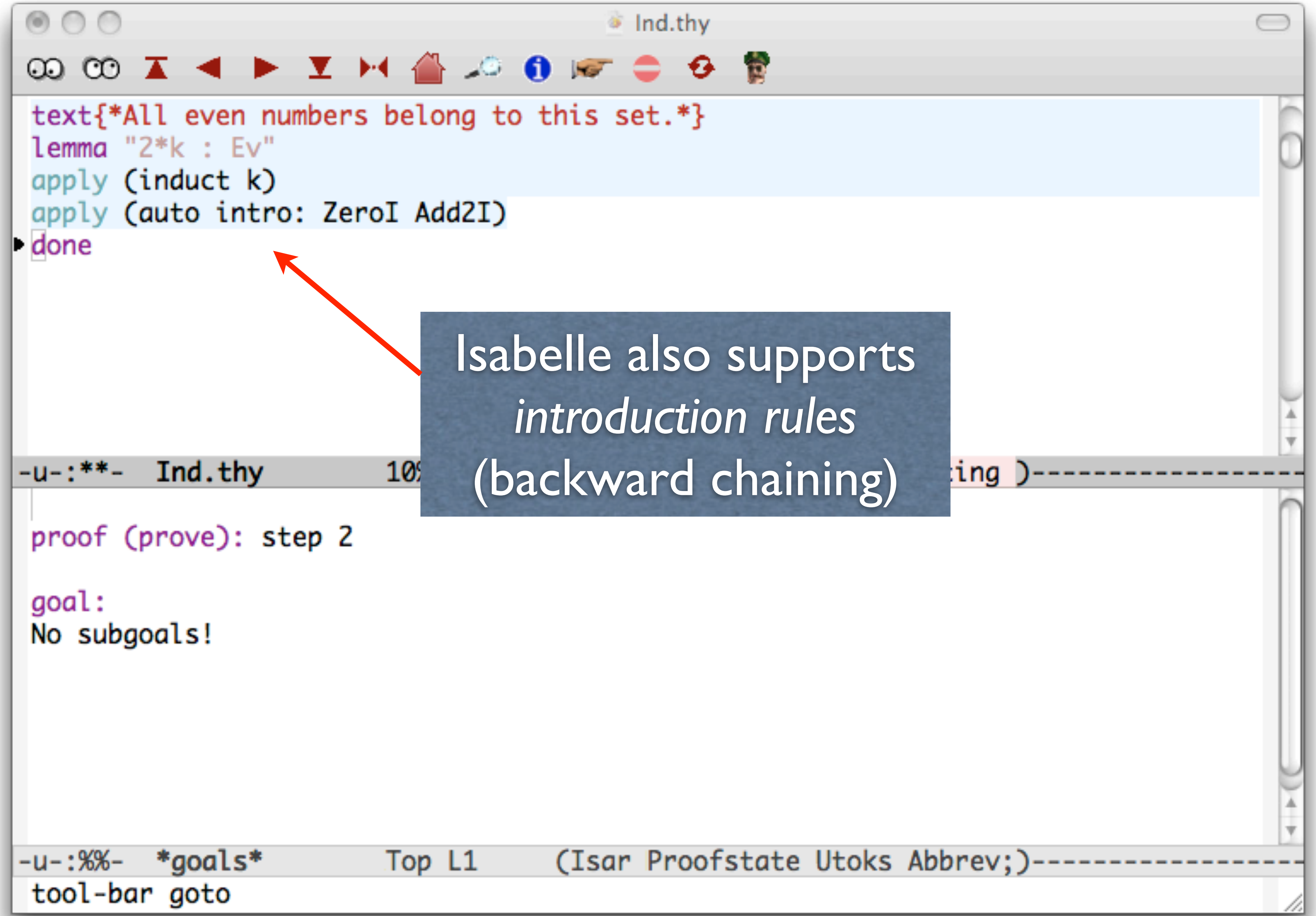
Even Numbers Belong to Ev



Proving Set Membership



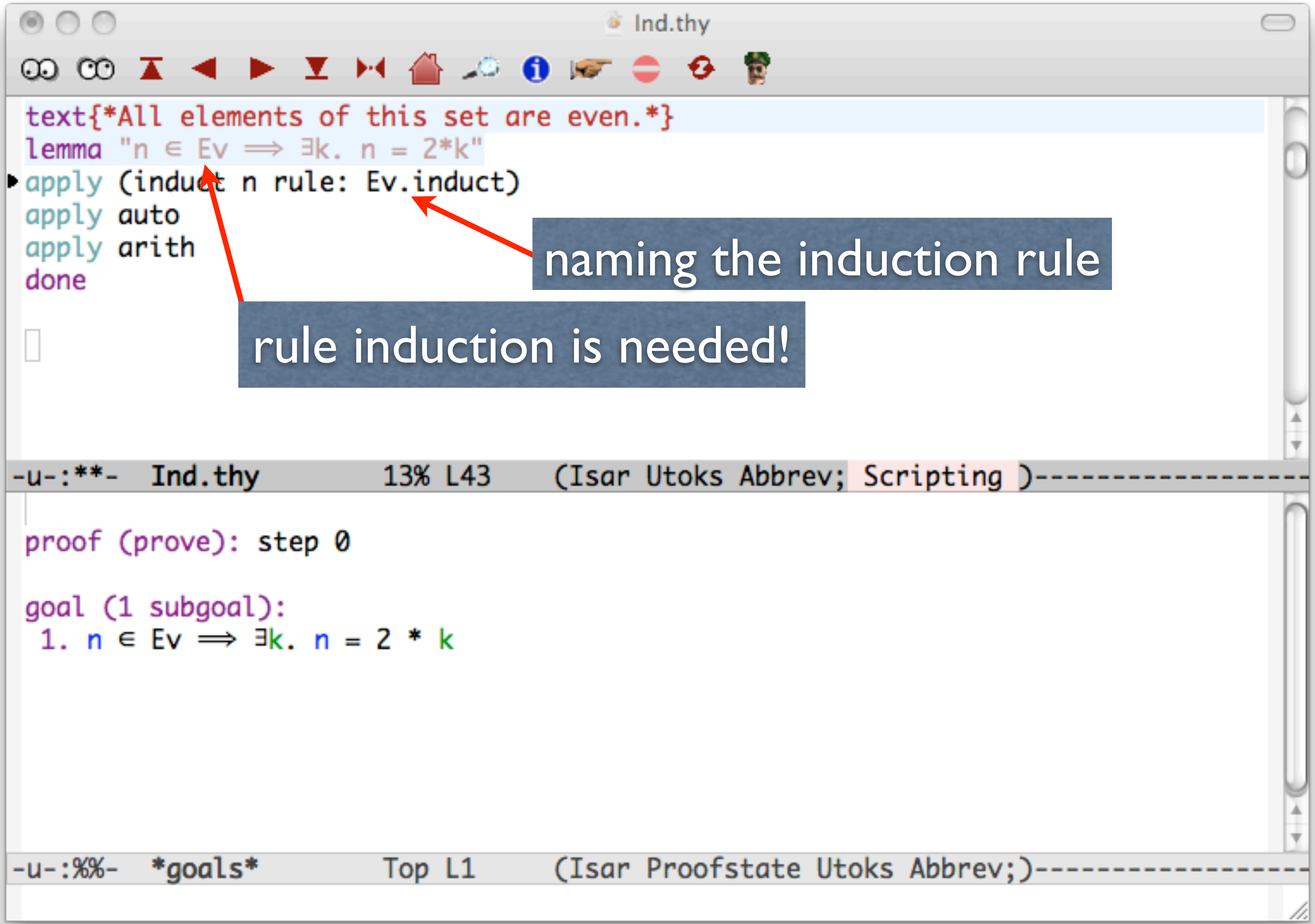
Finishing the Proof



Rule Induction

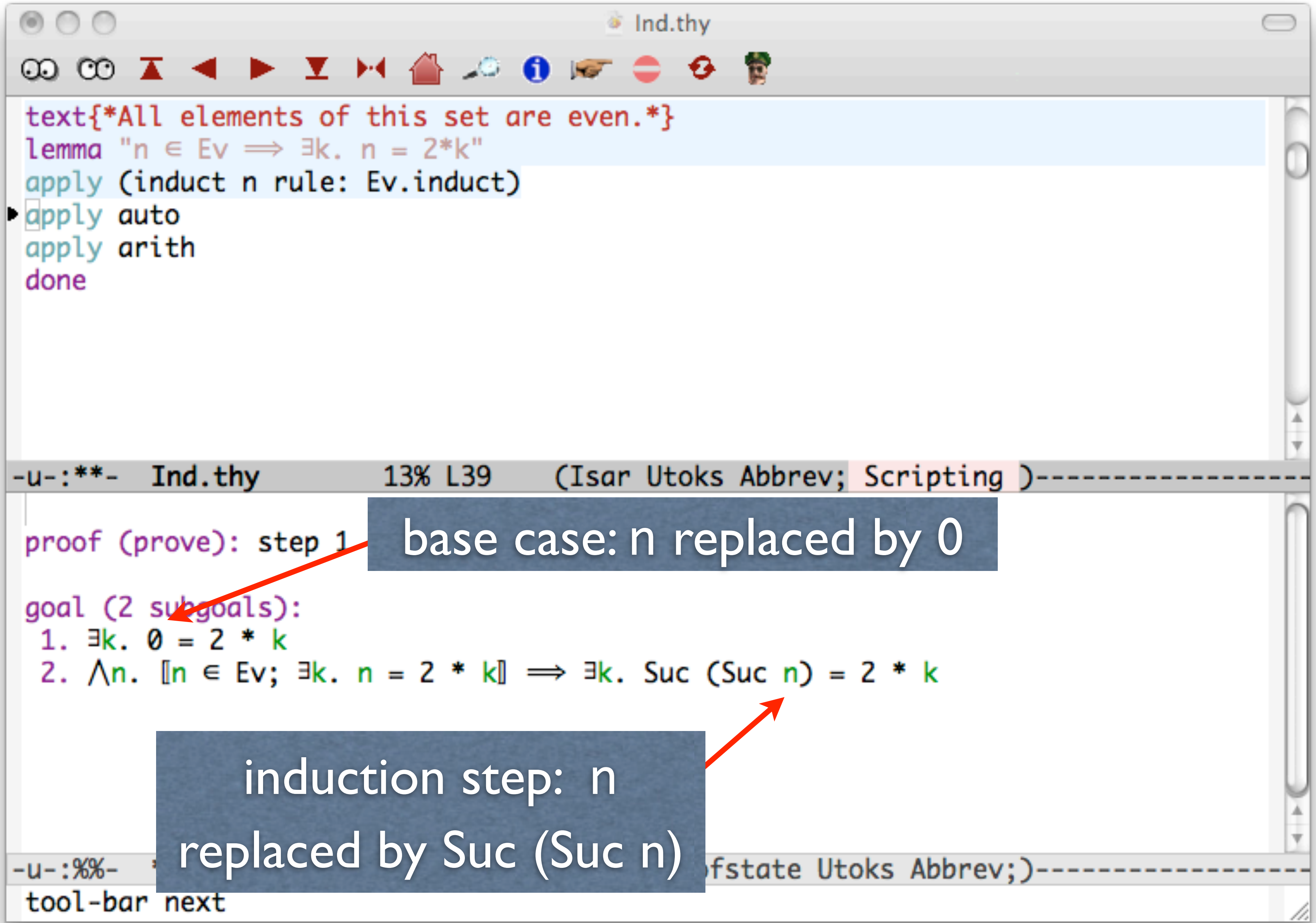
- Proving something about *every* element of the set.
- It expresses that the inductive set is *minimal*.
- It is sometimes called "induction on derivations"
- There is a *base case* for every non-recursive introduction rule
- ...and an *inductive step* for the other rules.

Ev Has only Even Numbers

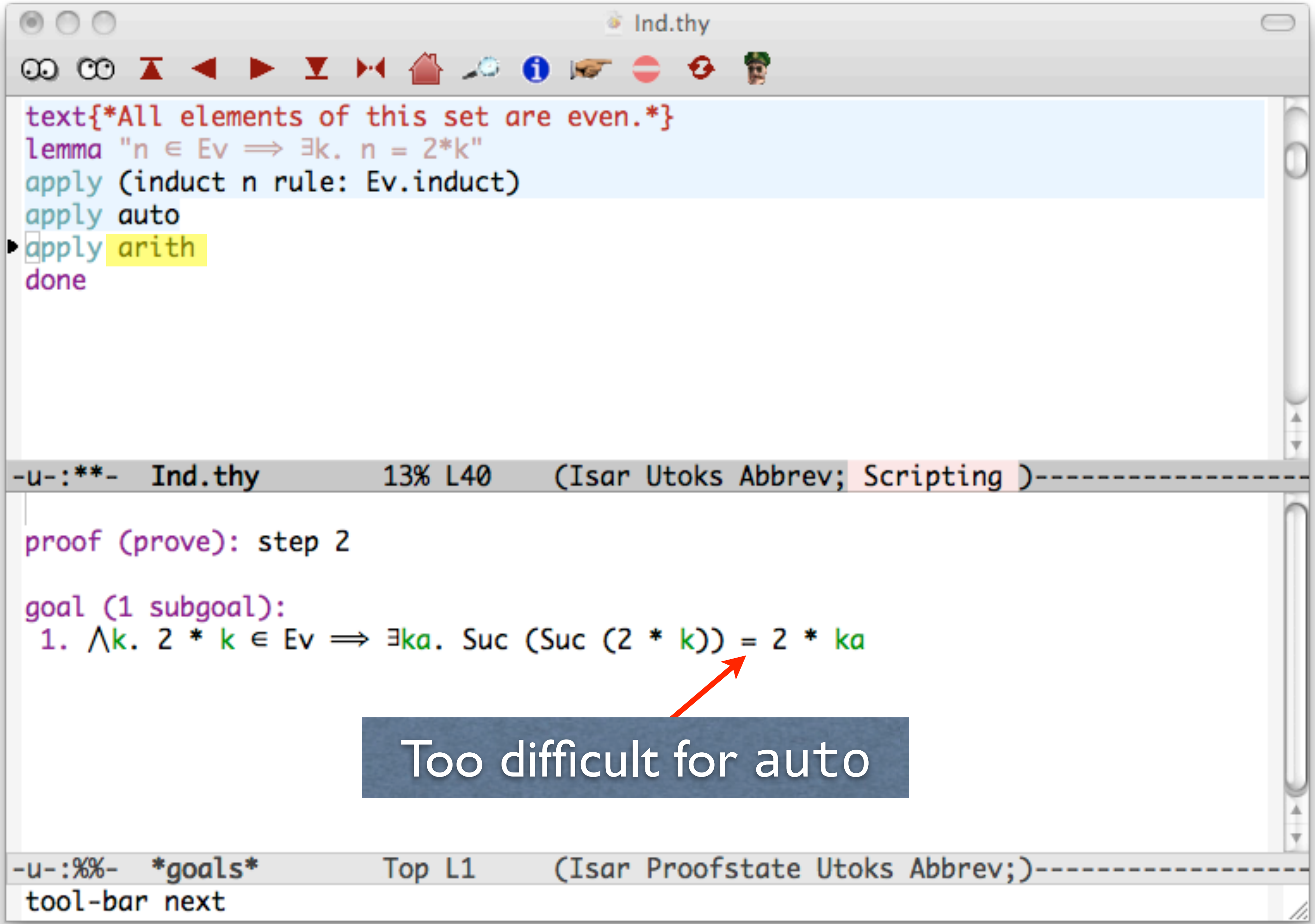


The classic sign that we need rule induction is an occurrence of the inductive set as a premise of the desired result. Of course, sometimes the theorem can be proved by referring to other facts that have been previously proved using rule induction.

An Example of Rule Induction

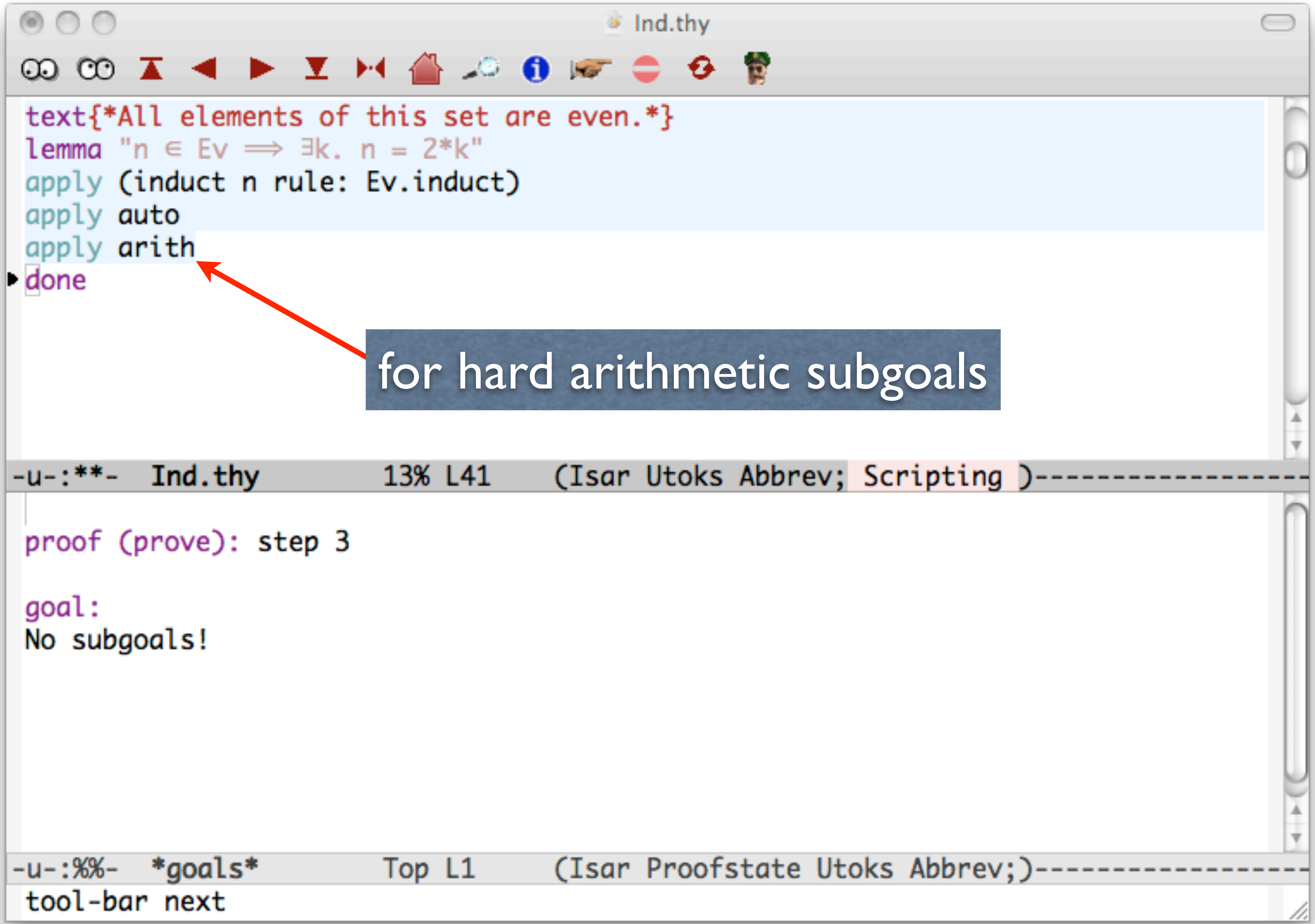


Nearly There!



The `auto` method provides some support for arithmetic. However, complicated arithmetic arguments require specialised proof methods.

The `arith` Proof Method



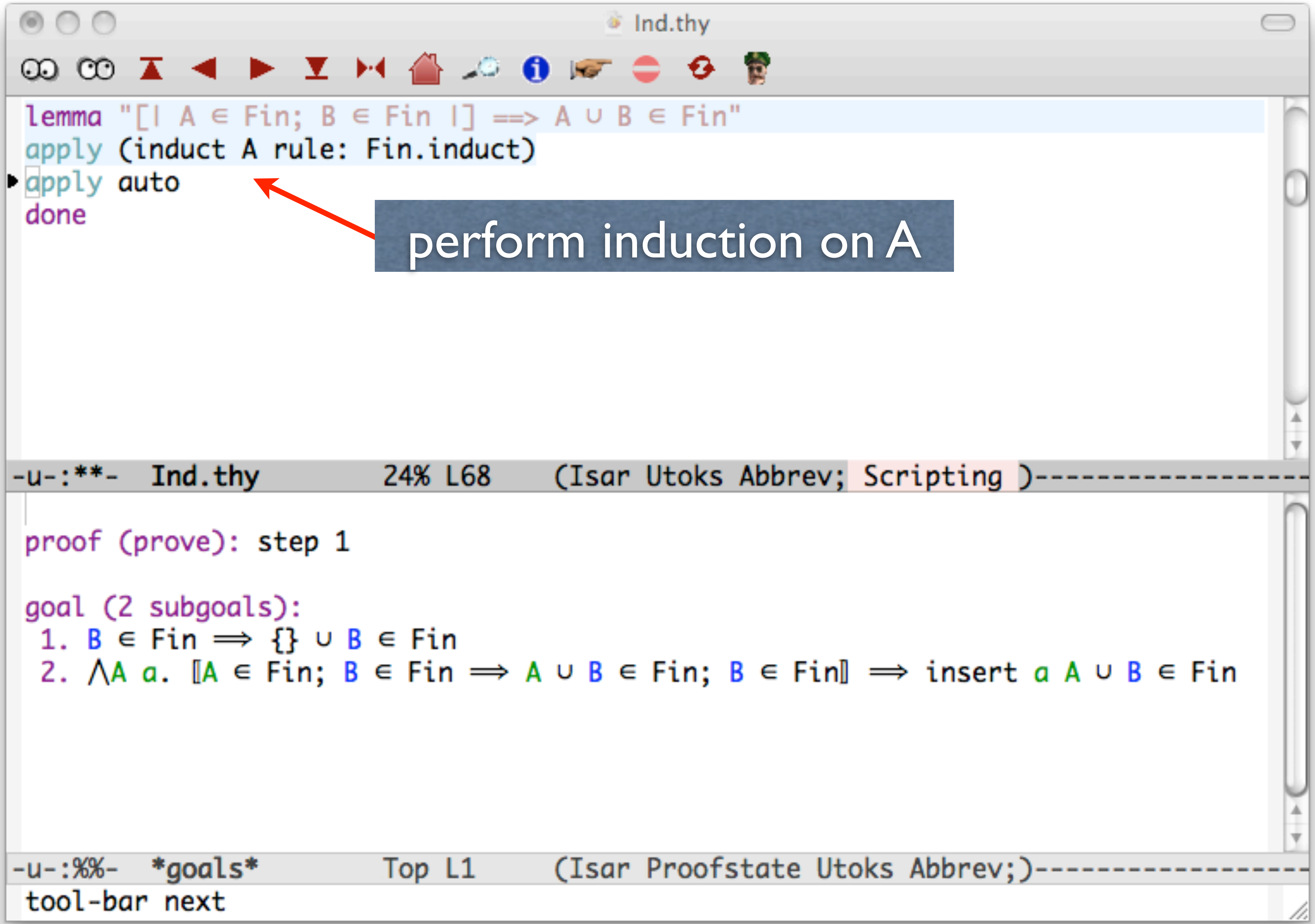
Defining Finiteness

```
subsection{* Proofs about finite sets *}

text{*The finite powerset operator*}

inductive_set Fin :: "'a set set" where
  emptyI:  "{} ∈ Fin"
| insertI: "A ∈ Fin ==> insert a A ∈ Fin"

declare Fin.intros [intro]
```

make the rules available to auto, blast

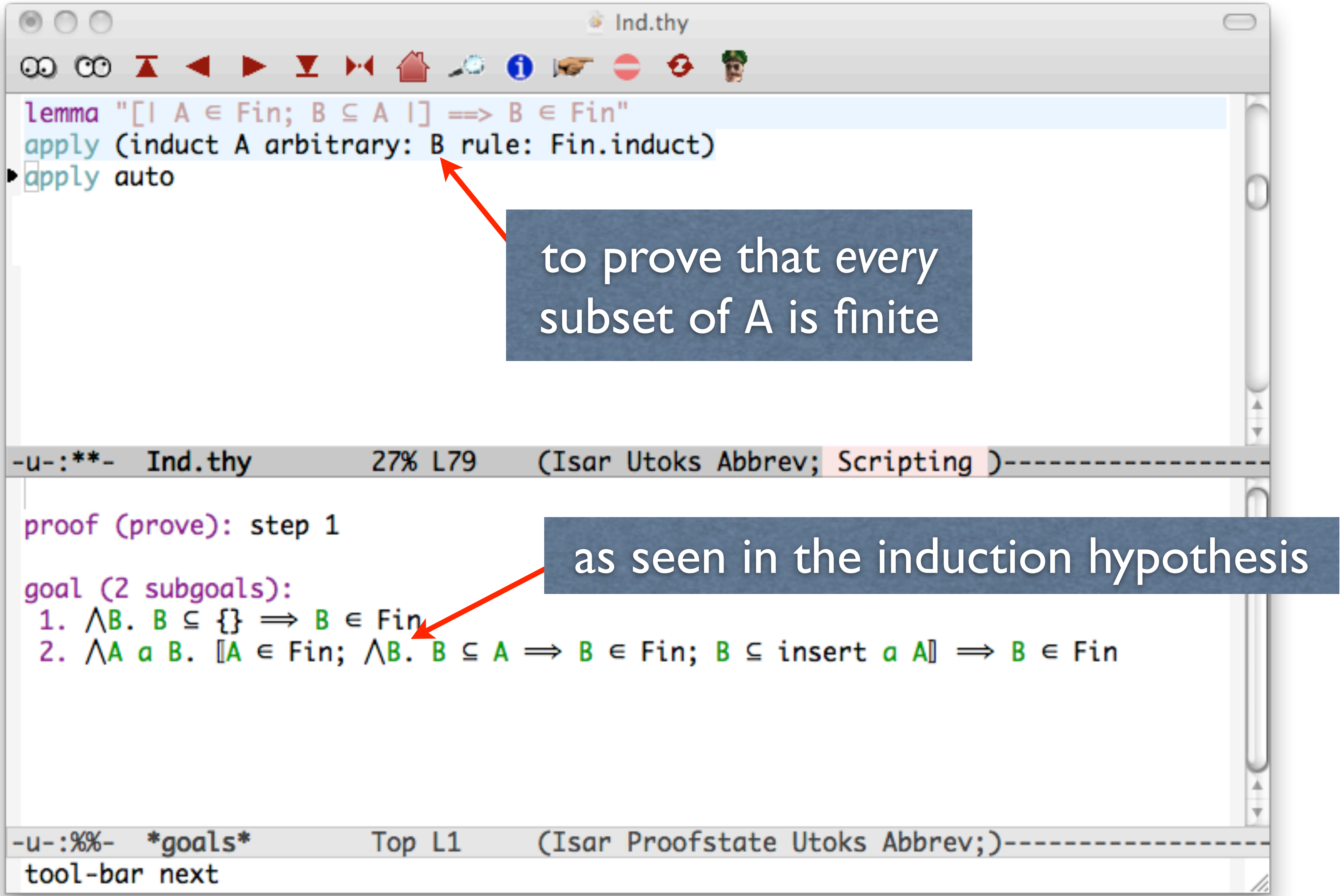
The empty set is finite. Adding one element to a finite set yields another finite set.

The Union of Two Finite Sets



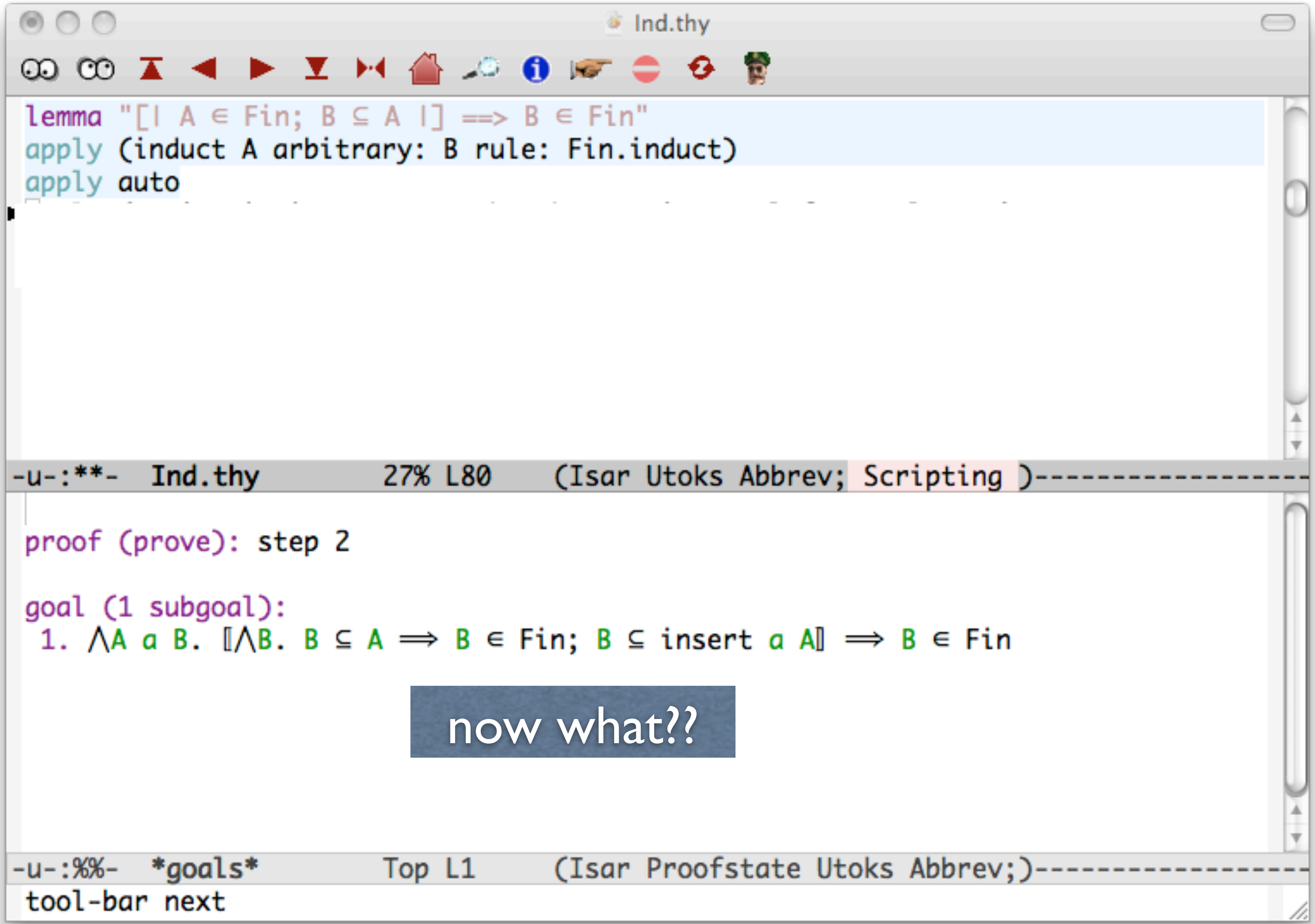
The goals are easily proved by the properties of sets and the introduction rules.

A Subset of a Finite Set



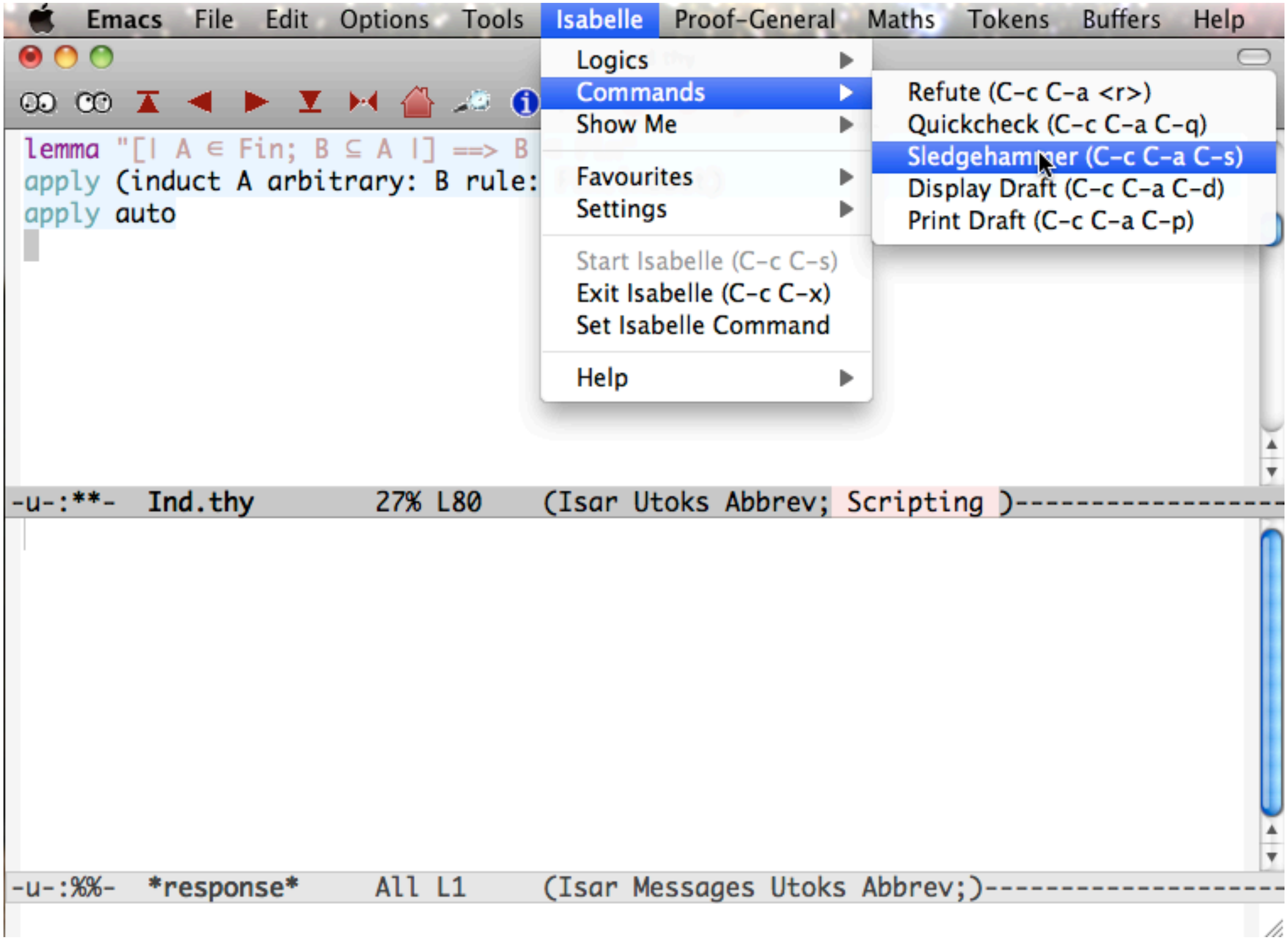
The proof is far more difficult than the preceding one, illustrating advanced techniques, in particular the sledgehammer tool.

A Crucial Point in the Proof



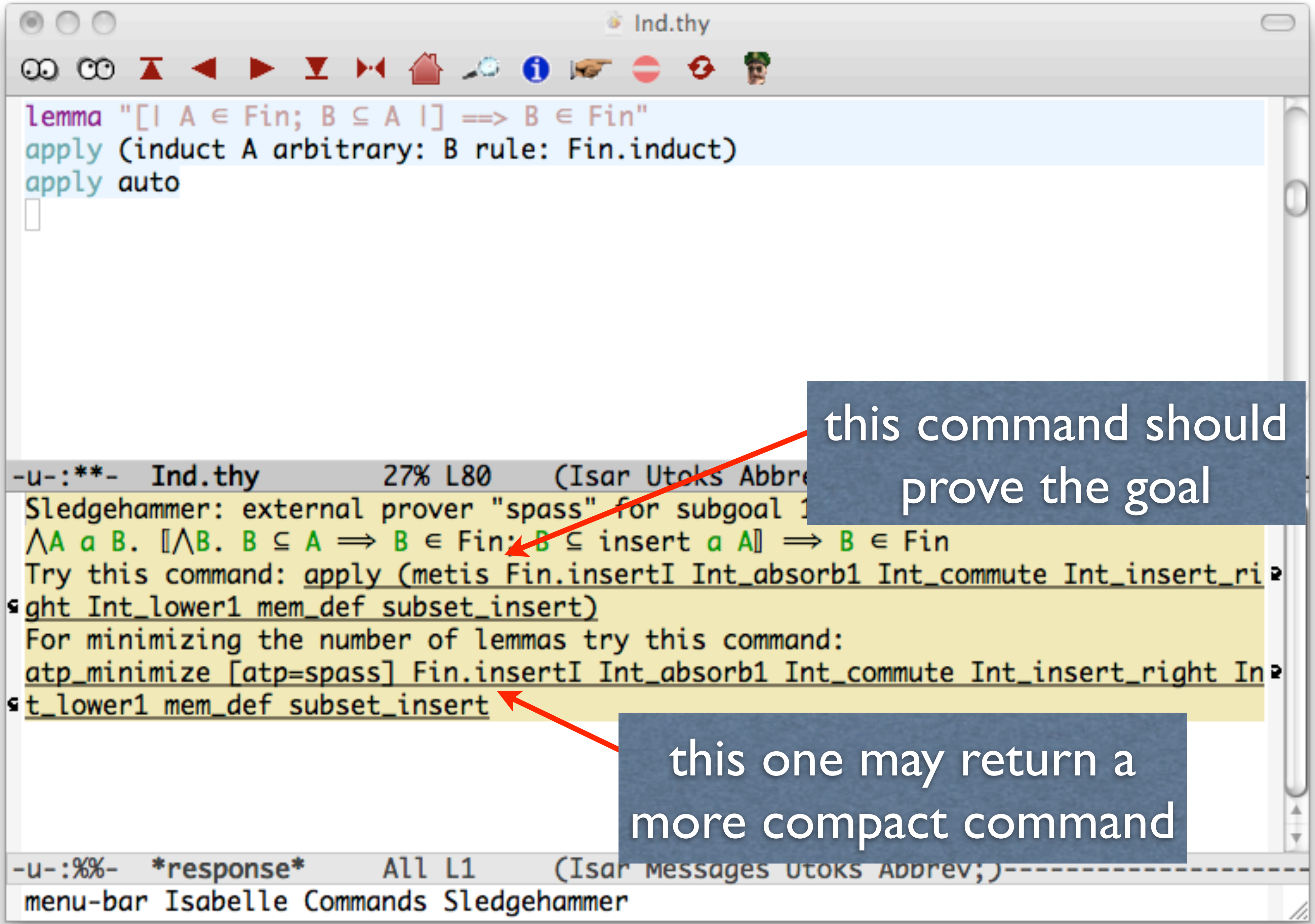
None of Isabelle's automatic proof methods (`auto`, `blast`, `force`) have any effect on this subgoal. Informally, we might consider case analysis on whether a∈B. This would require using proof tactics that have not been covered. Fortunately, Isabelle provides a general automated tool, sledgehammer.

Time to Try Sledgehammer!



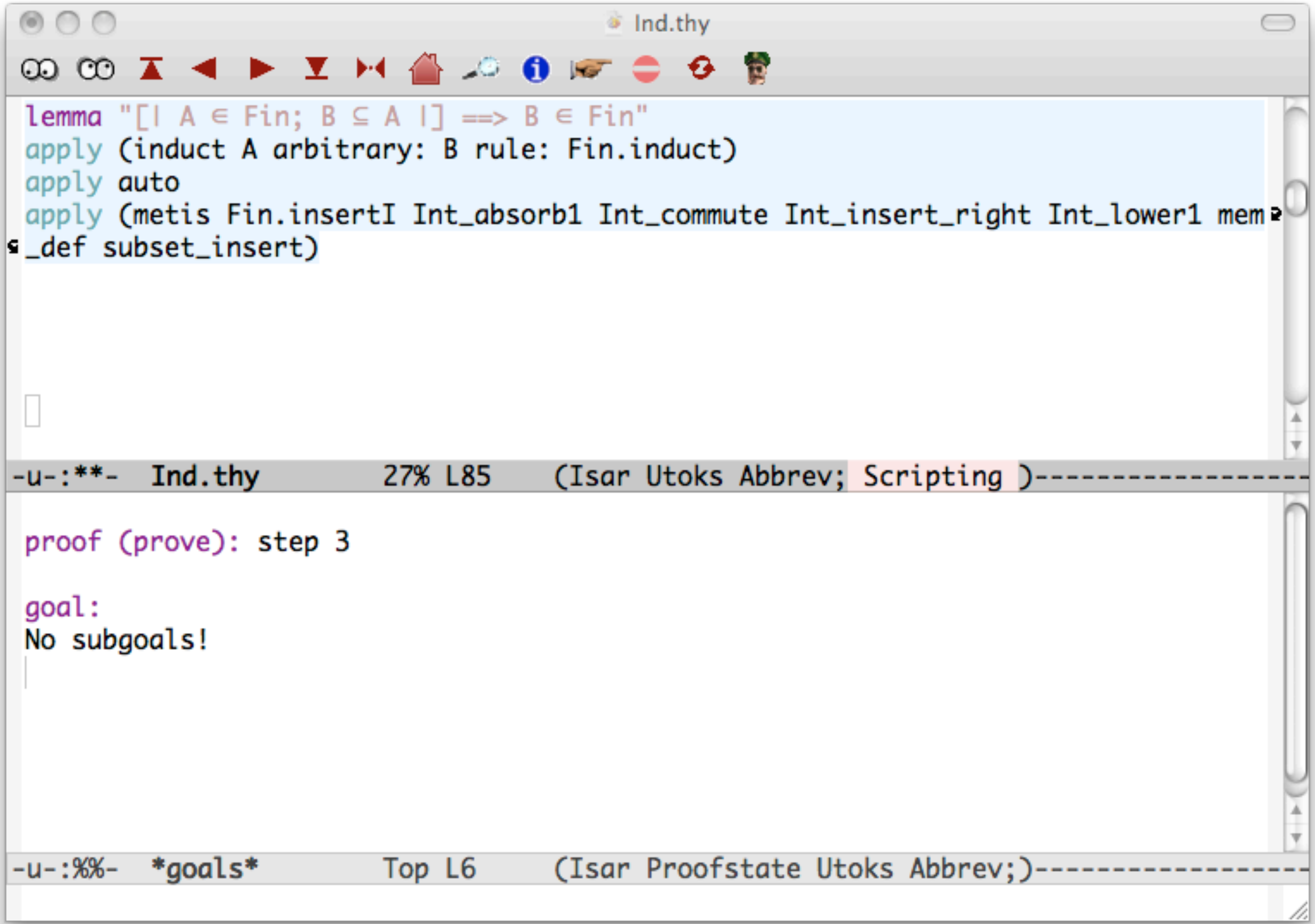
Sledgehammer calls several automated theorem provers in the background: in other words, Isabelle is still receptive to commands. You can continue to look for a proof manually.

Success!



Both outputs are highlighted in Proof General. They are live: clicking on either will insert that command into the proof script and execute it.

The Completed Proof



Notes on Sledgehammer

- It is always available, though it usually fails...
- It does not prove the goal, but returns a call to `metis`. This command *usually* works...
- The minimise option removes redundant theorems, increasing the likelihood of success.
- Calling `metis` directly is difficult unless you know exactly which lemmas are needed.

Metis is an automatic theorem prover for first order logic, written by Joe Hurd. Sledgehammer calls high-performance theorem provers, such as E and Vampire, using them as relevance filters to select from the thousands of lemmas available in Isabelle. Isabelle problems are translated for these automatic theorem provers using lightweight translations, which do not preserve soundness. For that reason, proofs found by those theorem provers may be incorrect. If that happens, the call to `metis` will generate an error message or fail to terminate. It is possible to force the use of sound translations, but sledgehammer seldom finds proofs using those.

Interactive Formal Verification 8: Operational Semantics

Lawrence C Paulson
Computer Laboratory
University of Cambridge

Overview

- The operational semantics of programming languages can be given *inductively*.
  - Type checking
  - Expression evaluation
  - Command execution, including concurrency
- Properties of the semantics are frequently proved by induction.
- Running example: an abstract language with WHILE

Language Syntax

```
typedecl loc -- "an unspecified type of locations"
types
  val   = nat -- "values"
  state = "loc => val"
  aexp  = "state => val"
  bexp  = "state => bool"    -- "just 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)
```

Arithmetic & boolean expressions are just functions over the state

For simplicity, this example does not specify arithmetic or boolean expressions in any detail. Although this approach is unrealistic, it allows us to illustrate key aspects of formalised proofs about programming language semantics.

Language Semantics

$$\langle \mathbf{skip}, s\rangle \rightarrow s \qquad\qquad \langle x := a, s\rangle \rightarrow s[x := a]$$

$$\frac{\langle c_0, s\rangle \rightarrow s'' \qquad \langle c_1, s''\rangle \rightarrow s'}{\langle c_0; c_1, s\rangle \rightarrow s'}$$

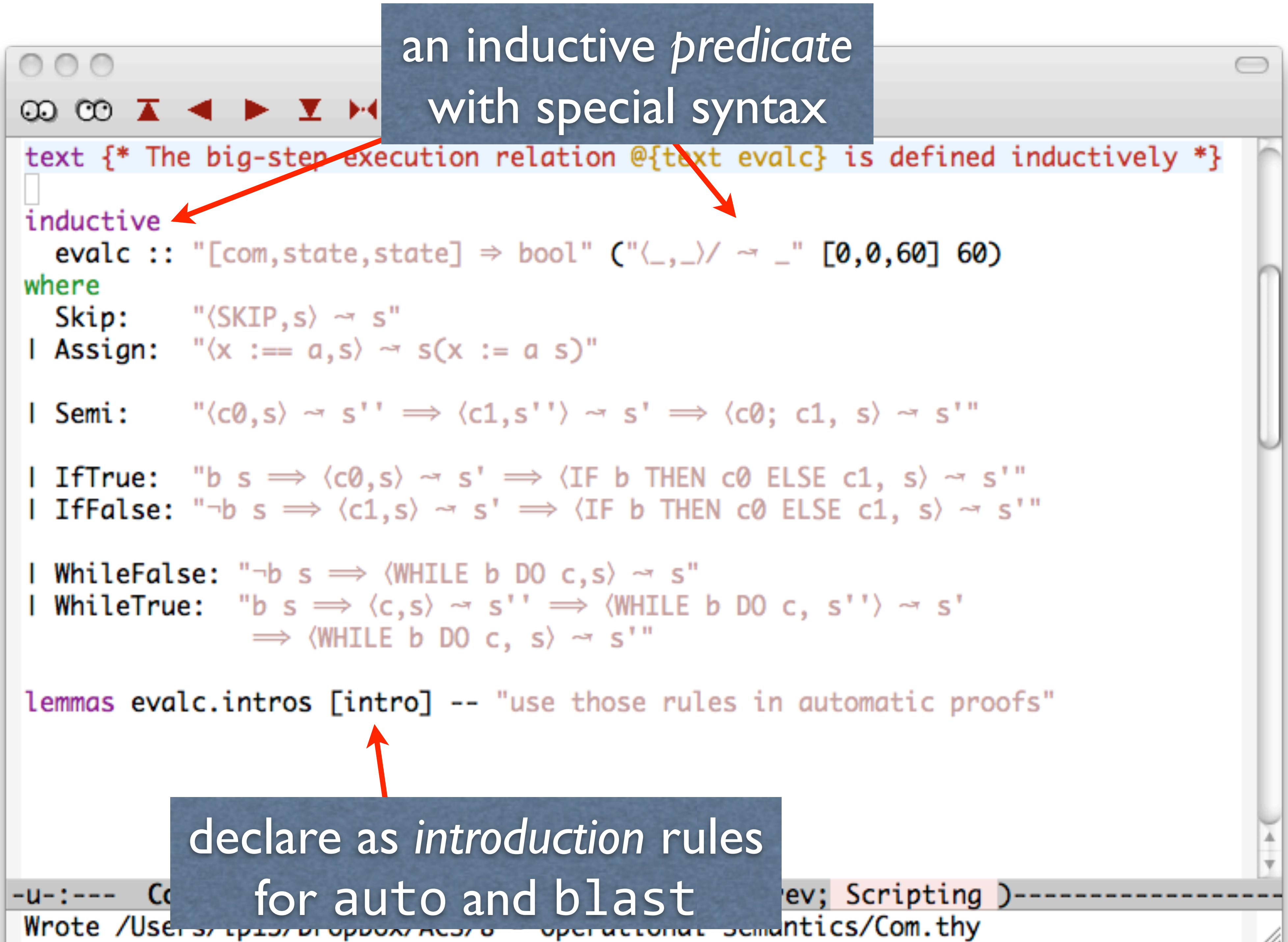
$$\frac{b\,s \qquad \langle c_0, s\rangle \rightarrow s'}{\langle \mathbf{if}\ b\ \mathbf{then}\ c_0\ \mathbf{else}\ c_1, s\rangle \rightarrow s'} \qquad \frac{\neg\, b\,s \qquad \langle c_1, s\rangle \rightarrow s'}{\langle \mathbf{if}\ b\ \mathbf{then}\ c_0\ \mathbf{else}\ c_1, s\rangle \rightarrow s'}$$

$$\frac{\neg\, b\,s}{\langle \mathbf{while}\ b\ \mathbf{do}\ c, s\rangle \rightarrow s} \qquad \frac{b\,s \quad \langle c, s\rangle \rightarrow s'' \quad \langle \mathbf{while}\ b\ \mathbf{do}\ c, s''\rangle \rightarrow s'}{\langle \mathbf{while}\ b\ \mathbf{do}\ c, s\rangle \rightarrow s'}$$

A "big-step" semantics

In a big step semantics, the transition $\langle c,s\rangle \rightarrow s'$ means, executing the command *c* starting in the state *s* can terminate in state *s'*.

Formalised Language Semantics



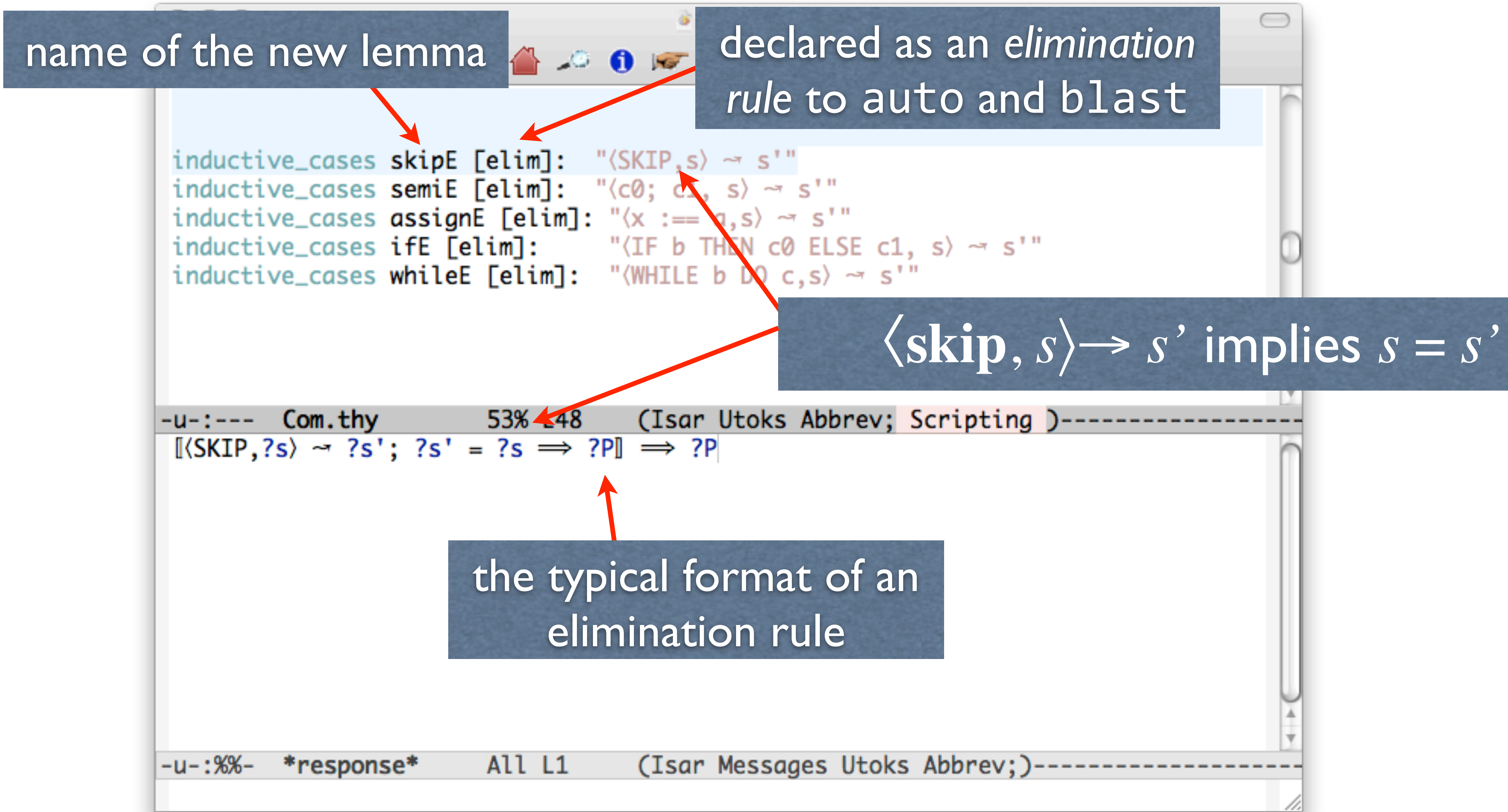
In the previous lecture, we used a related declaration, `inductive_set`. Note that there is no real difference between a set and a predicate of one argument. However, formal semantics generally requires a predicate three or four arguments, and the corresponding set of triples is a little more difficult to work with. Attaching special syntax, as shown above, also requires the use of a predicate. Therefore, formalised semantic definitions will generally use `inductive`.

Rule Inversion

- When $\langle \mathbf{skip}, s\rangle \rightarrow s'$ we know $s = s'$
- When $\langle \mathbf{if}\ b\ \mathbf{then}\ c_0\ \mathbf{else}\ c_1, s\rangle \rightarrow s'$ we know
  - b and $\langle c_0, s\rangle \rightarrow s'$, or...
  - $\neg b$ and $\langle c_1, s\rangle \rightarrow s'$
- This sort of case analysis is easy in Isabelle.

Rule inversion refers to case analysis on the form of the induction, matching the conclusions of the introduction rules (those making up the inductive definition) with a particular pattern. It is useful when only a small percentage of the introduction rules can match the pattern. This type of reasoning is extremely common in informal proofs about operational semantics. It would not be useful in the inductive definitions covered in the previous lecture, where the conclusions of the rules had little structure.

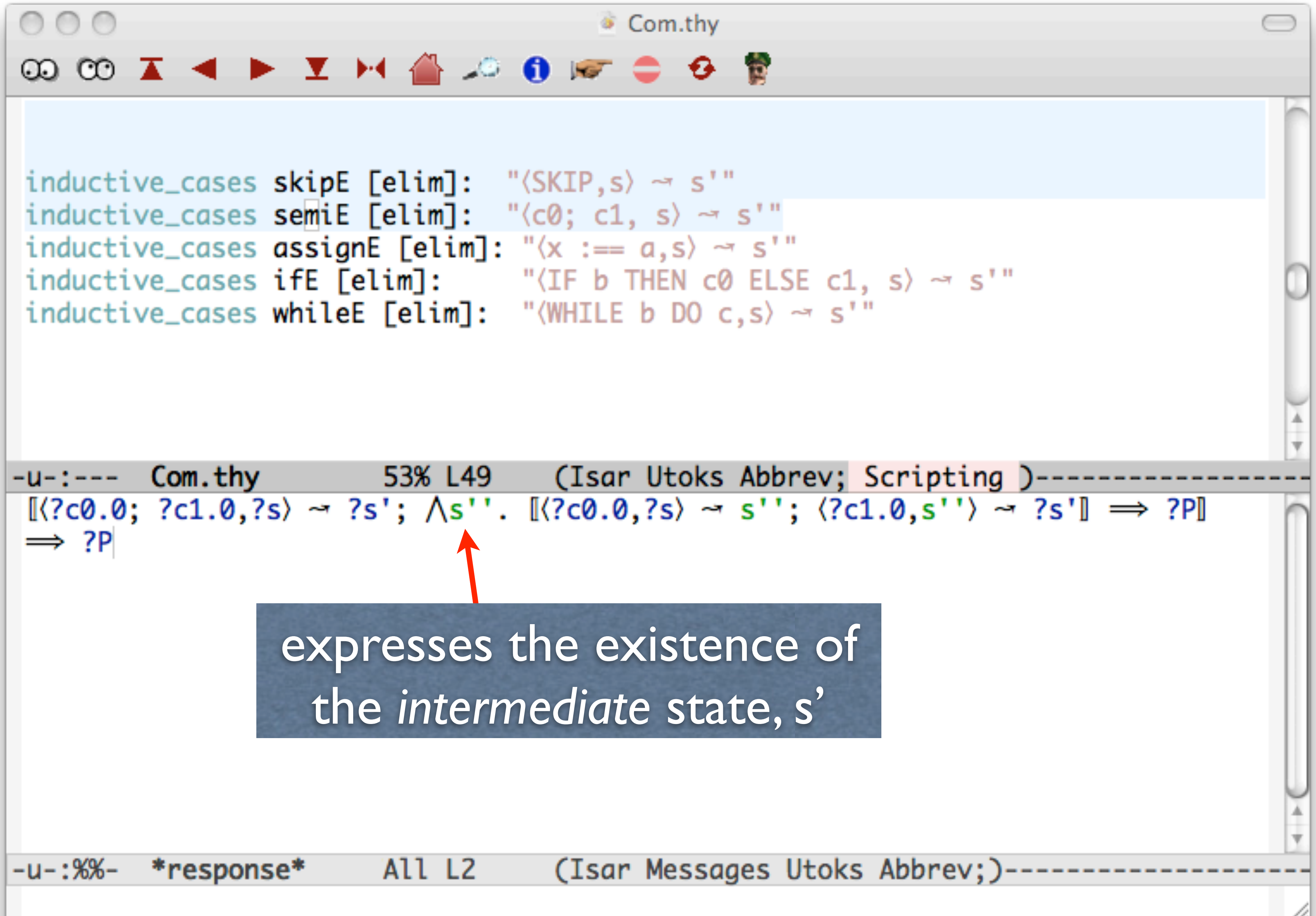
Rule Inversion in Isabelle



The pattern for each rule inversion lemma appears in quotation marks. Isabelle generates a theorem and gives it the name shown. Each theorem is also made available to Isabelle's automatic tools.

It is possible to write elim! rather than just elim; the exclamation mark tells Isabelle to apply the lemma aggressively. However, this must not be done with the theorem `whileE`: it expands an occurrence of ⟨**while** *b* **do** c, s⟩ → s' and generates another formula of essentially the same form, thereby running for ever.

Rule Inversion Again



A Non-Termination Proof

$$\langle \textbf{while true do } c, s\rangle \not\to s'$$

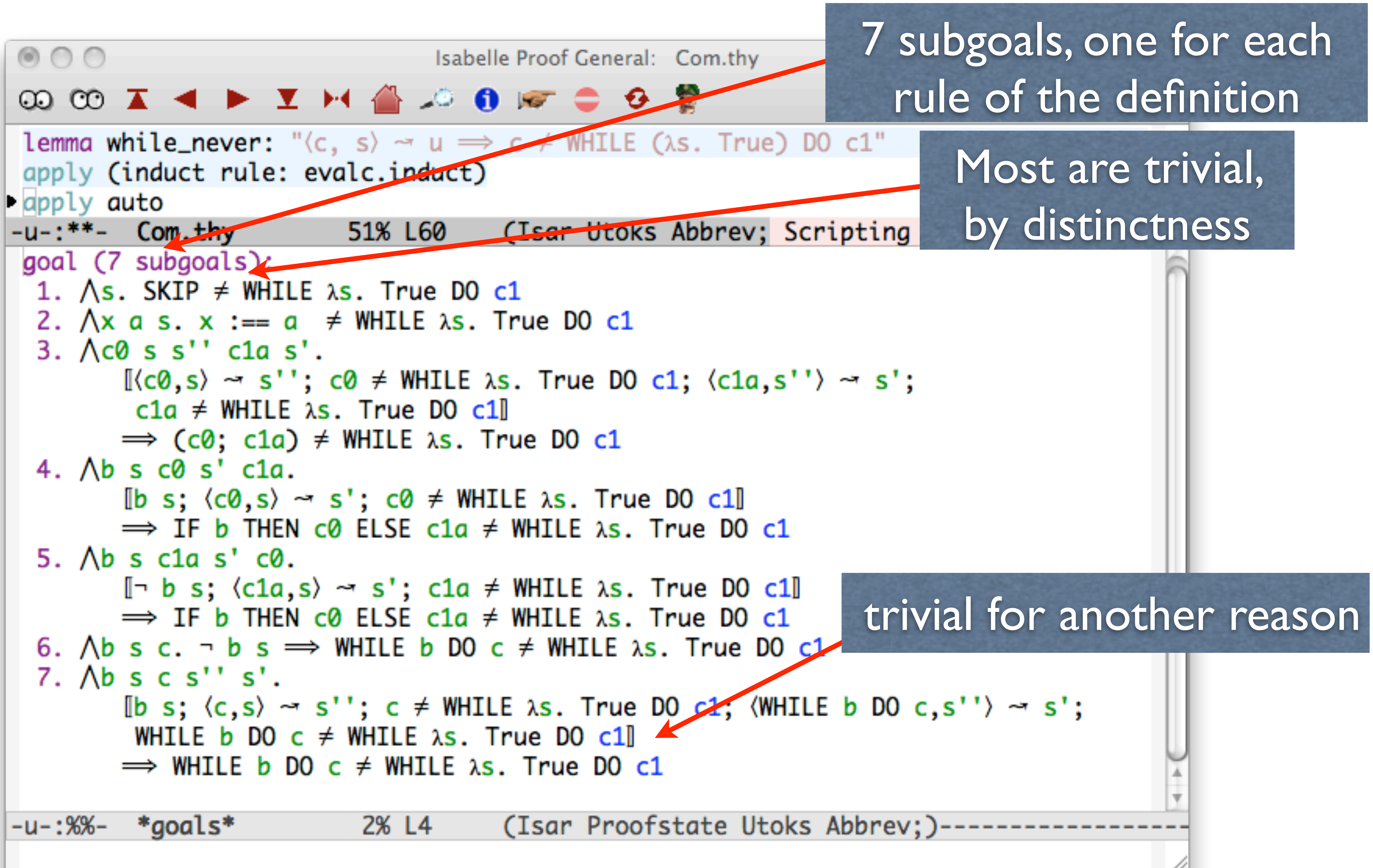
Not provable by induction!

$$\langle c, s\rangle \to s' \implies \forall c'.\, c \neq (\textbf{while true do } c')$$

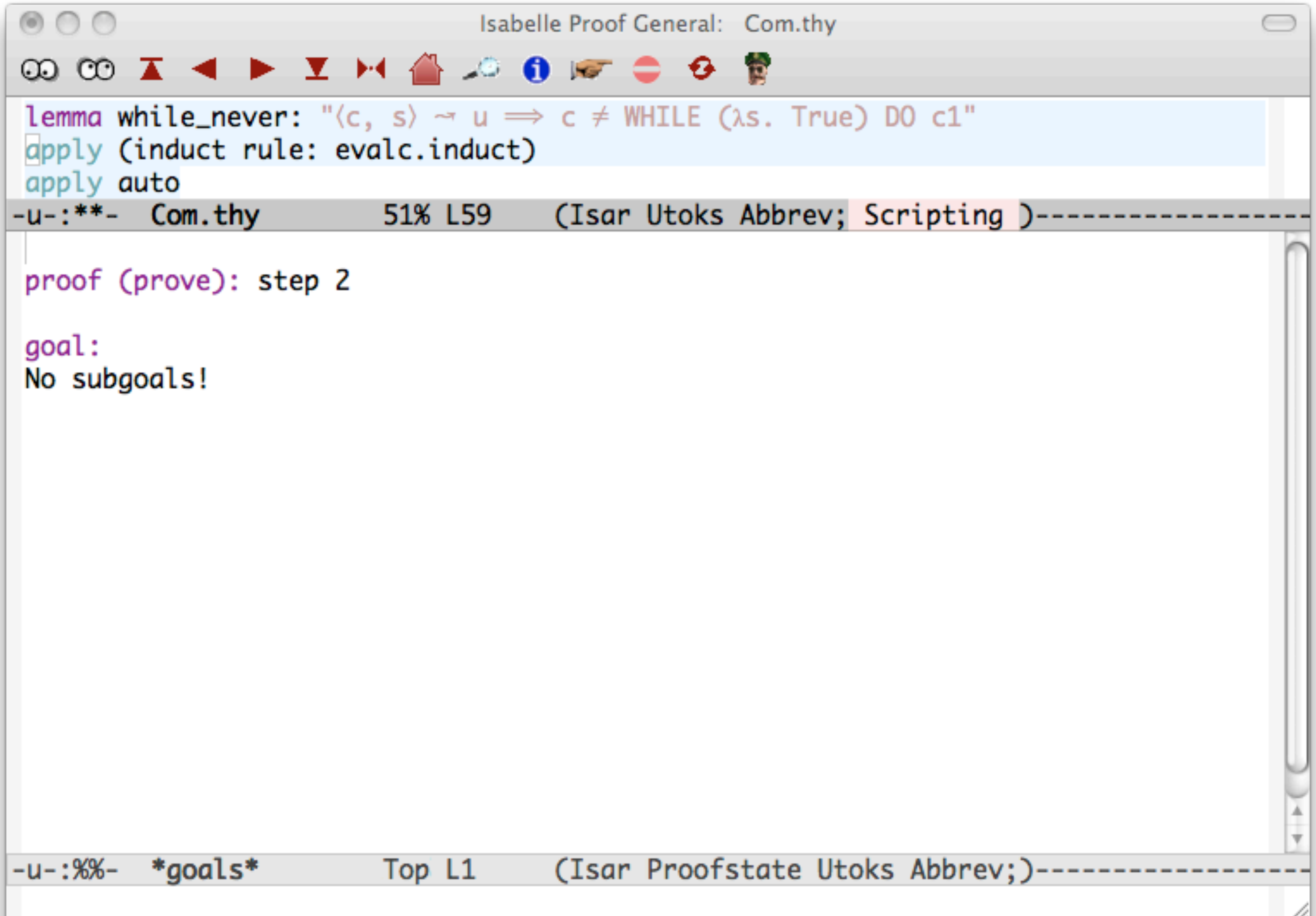
The inductive version considers

all possible commands

Non-Termination in Isabelle



Done!



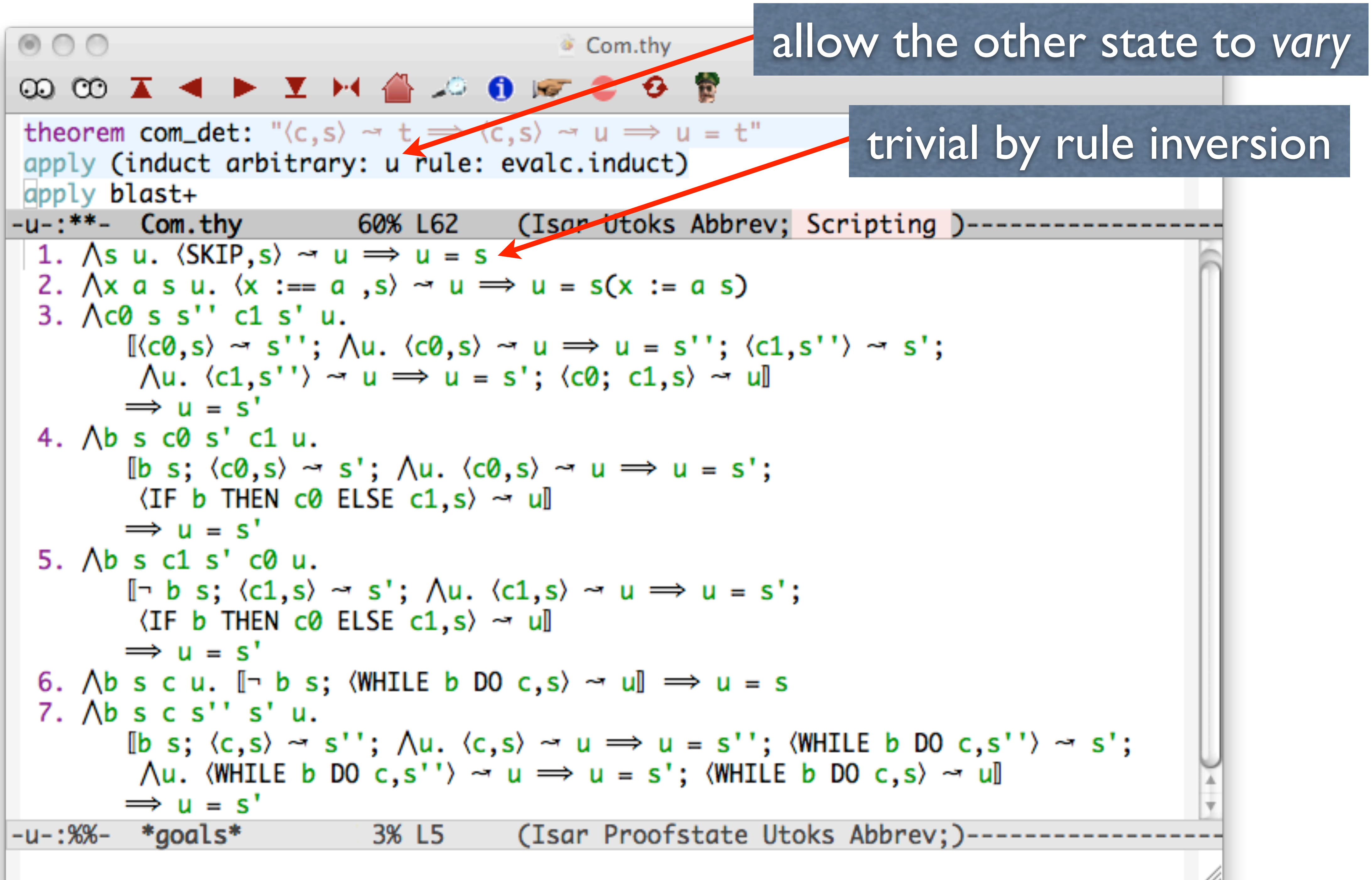
This really is a trivial proof. I timed this call to `auto` and it needed only 6 ms.

Determinacy

$$\frac{\langle c, s\rangle \to t \qquad \langle c, s\rangle \to u}{t = u}$$

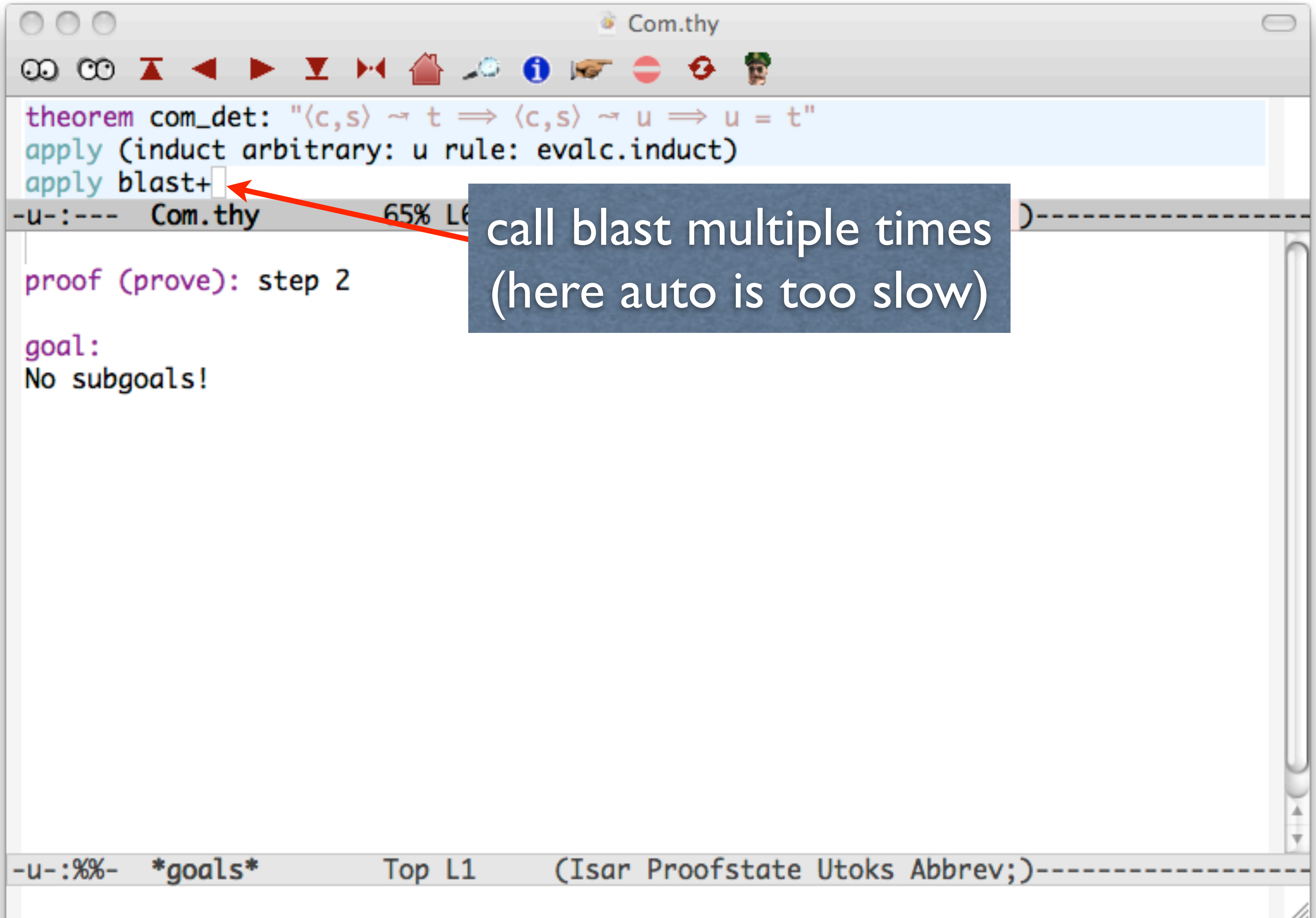
If a command is executed in a given state, and it terminates, then this final state is *unique*.

Determinacy in Isabelle



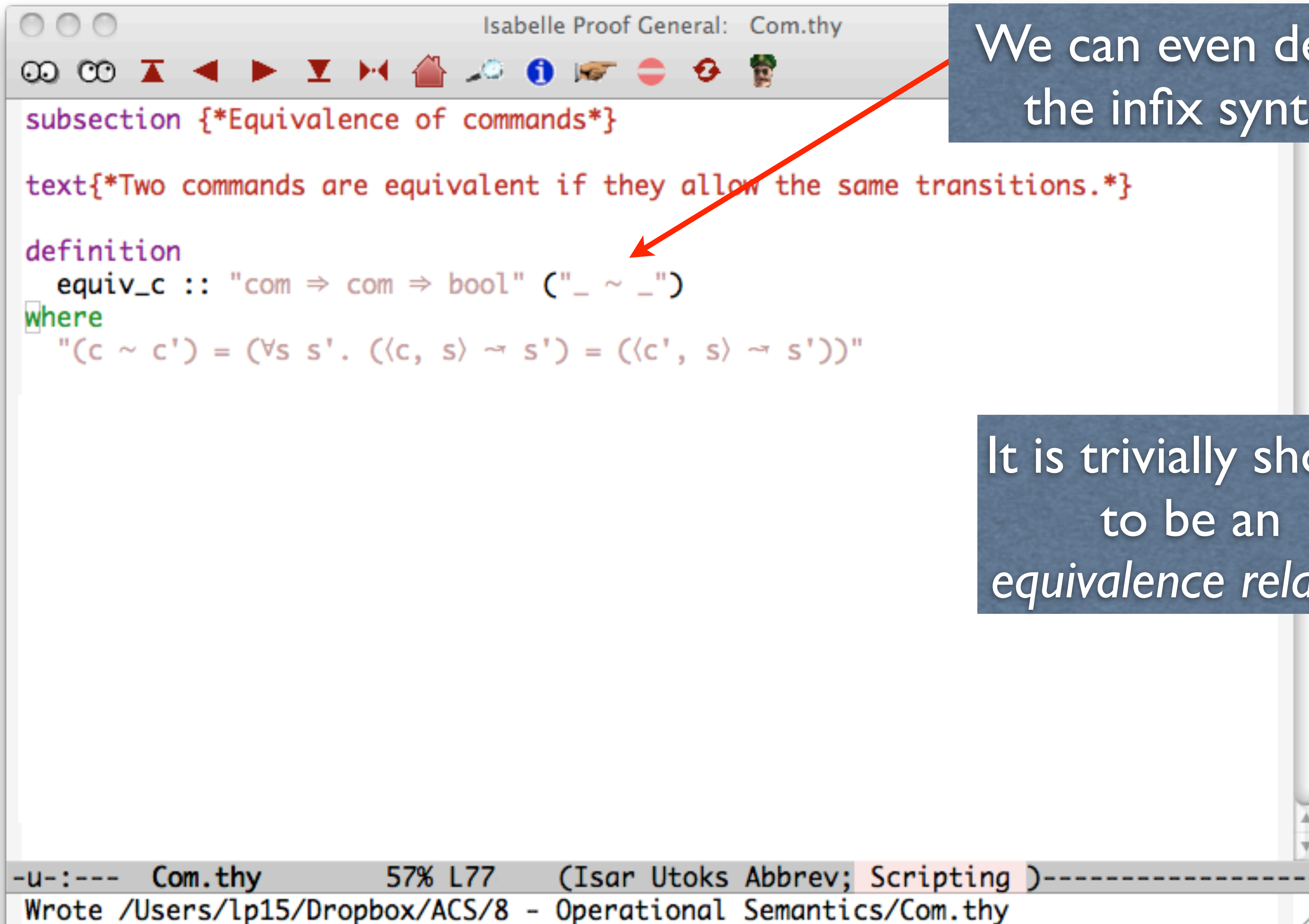
The proof method `blast` uses introduction and elimination rules, combined with powerful search heuristics. It will not terminate until it has solved the goal. Unlike `auto` and `force`, it does not perform simplification (rewriting) or arithmetic reasoning.

Proved by Rule Inversion



The proof involves a long, tedious and detailed series of rule inversions. Apart from its length, the proof is trivial. This proof needed only 32 ms.

Semantic Equivalence



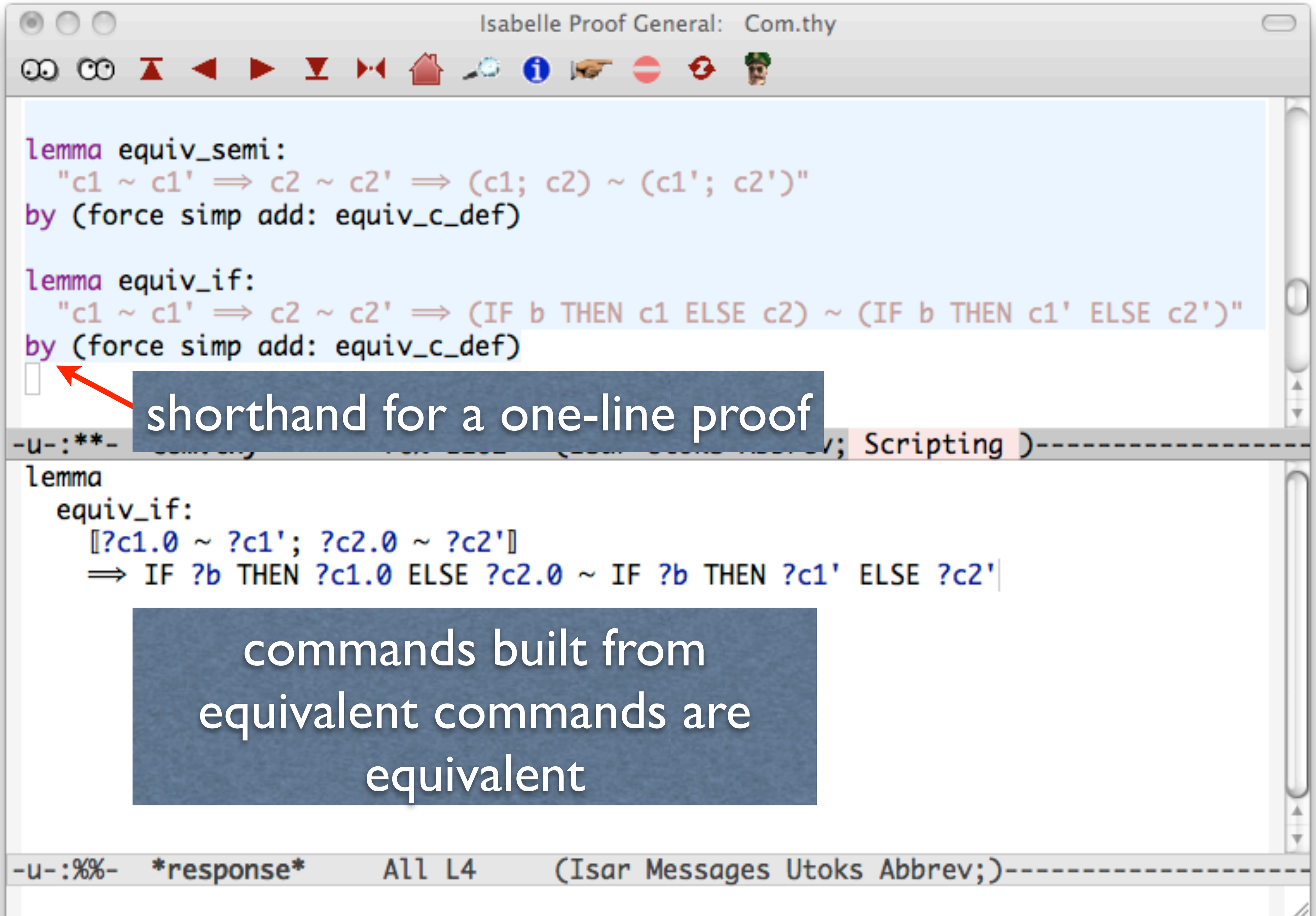
We can even define the infix syntax

It is trivially shown to be an *equivalence relation*

The printed version of these notes does not include the actual proofs, because they are revealed during the presentation. They are reproduced below. It is necessary to unfold the definition of semantic equivalence, `equiv_c`. By default, Isabelle does not unfold nonrecursive definitions.

```
lemma equiv_refl:
  "c ~ c"
by (auto simp add: equiv_c_def)

lemma equiv_sym:
  "c1 ~ c2 ==> c2 ~ c1"
by (auto simp add: equiv_c_def)

lemma equiv_trans:
  "c1 ~ c2 ==> c2 ~ c3 ==> c1 ~ c3"
by (auto simp add: equiv_c_def)
```

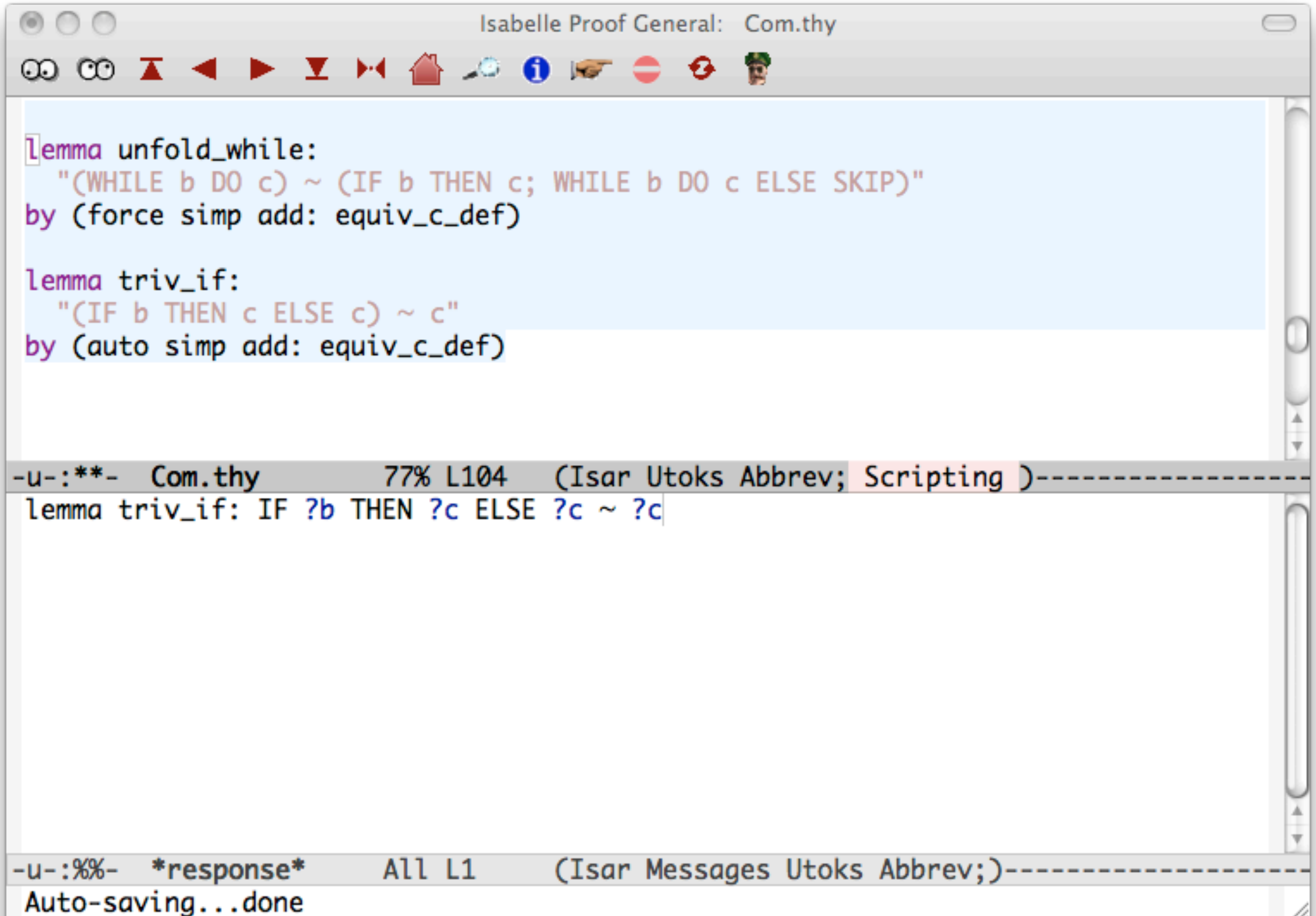
More Semantic Equivalence!



The properties shown here establish that semantic equivalence is a congruence relation with respect to the command constructors `Semi` and `Cond`. The proofs are again trivial, providing we remember to unfold the definition of semantic equivalence, `equiv_c`. Proving the analogous congruence property for While is harder, requiring rule induction with an induction formula similar to that used for another proof about `While` earlier in this lecture.

The proof method `force` is similar to `auto`, but it is more aggressive and it will not terminate until it has proved the subgoal it was applied to. In these examples, `auto` will give up too easily.

And More!!

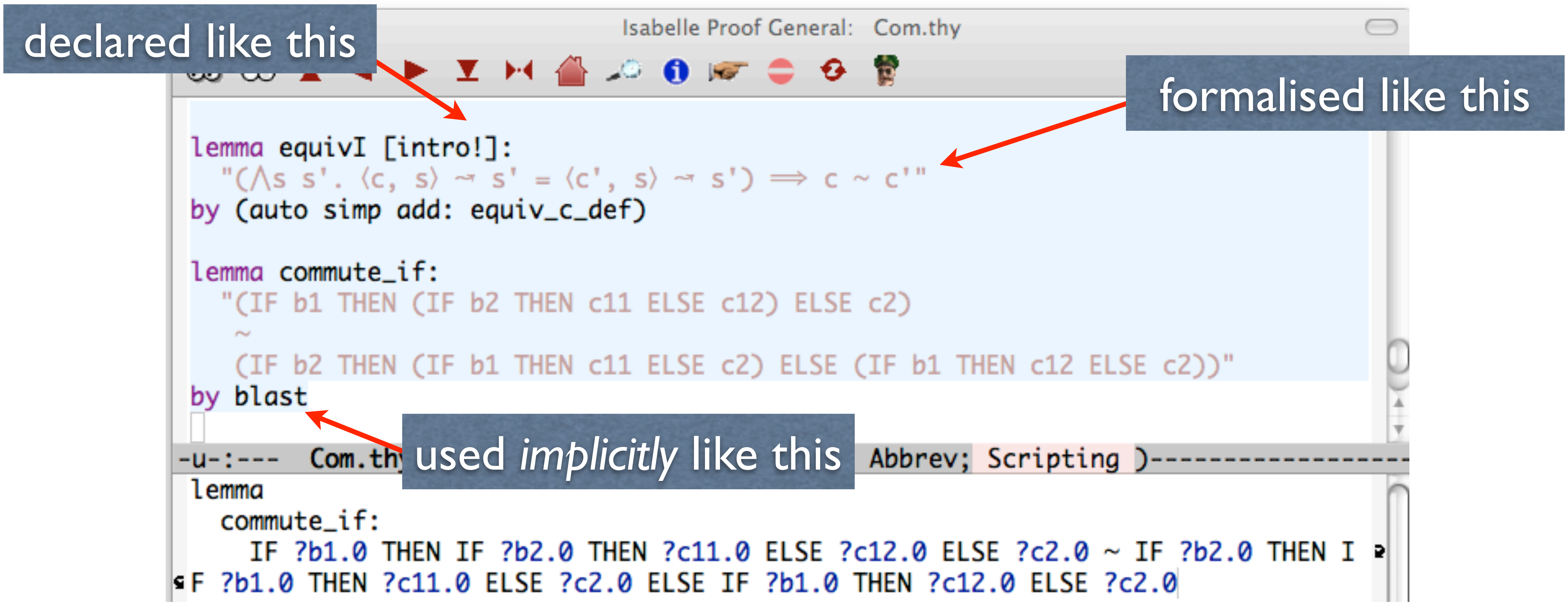


By some fluke, `force` will not solve the second of these. Sometimes you just have to try different things.

Note that a proof consisting of a single proof method can be written using the command “`by`”, which is more concise than writing “`apply`” followed by “`done`”. It is a small matter here, but structured proofs (which we are about to discuss) typically consist of numerous one line proofs expressed using “`by`”.

A New Introduction Rule

$$\frac{\langle c, s\rangle \to s' \iff \langle c', s\rangle \to s'}{c \sim c'} \qquad c \text{ and } c' \text{ not free...}$$



Giving the attribute `intro!` to a theorem informs Isabelle's automatic proof methods, including `auto`, `force` and `blast`, that this theorem should be used as an introduction rule. In other words, it should be used in backward-chaining mode: the conclusion of the rule is unified with the subgoal, continuing the search from that rule's premises. It is now unnecessary to mention this theorem when calling those proof methods. The theorem shown can now be proved using blast alone. We do not need to refer to `equivI` or to the definition of `equiv_c`. The approach used to prove other examples of semantic equivalence in this lecture do not terminate on this problem in a reasonable time. The proof shown only requires 12 ms.

The exclamation mark (`!`) tells Isabelle to apply the rule aggressively. It is appropriate when the premise of the rule is equivalent to the conclusion; equivalently, it is appropriate when applying the rule can never be a mistake. The weaker attribute `intro` should be used for a theorem that is one of many different ways of proving its conclusion.

Final Remarks on Semantics

- Small-step semantics is treated similarly.
- Variable binding is crucial in larger examples, and should be formalised using the *nominal package*.
    - choosing a fresh variable
    - renaming bound variables consistently
- Serious proofs will be complex and difficult!

Documentation on the nominal package can be downloaded from `http://isabelle.in.tum.de/nominal/`

Many examples are distributed with Isabelle. See the directory `HOL/Nominal/Examples`.

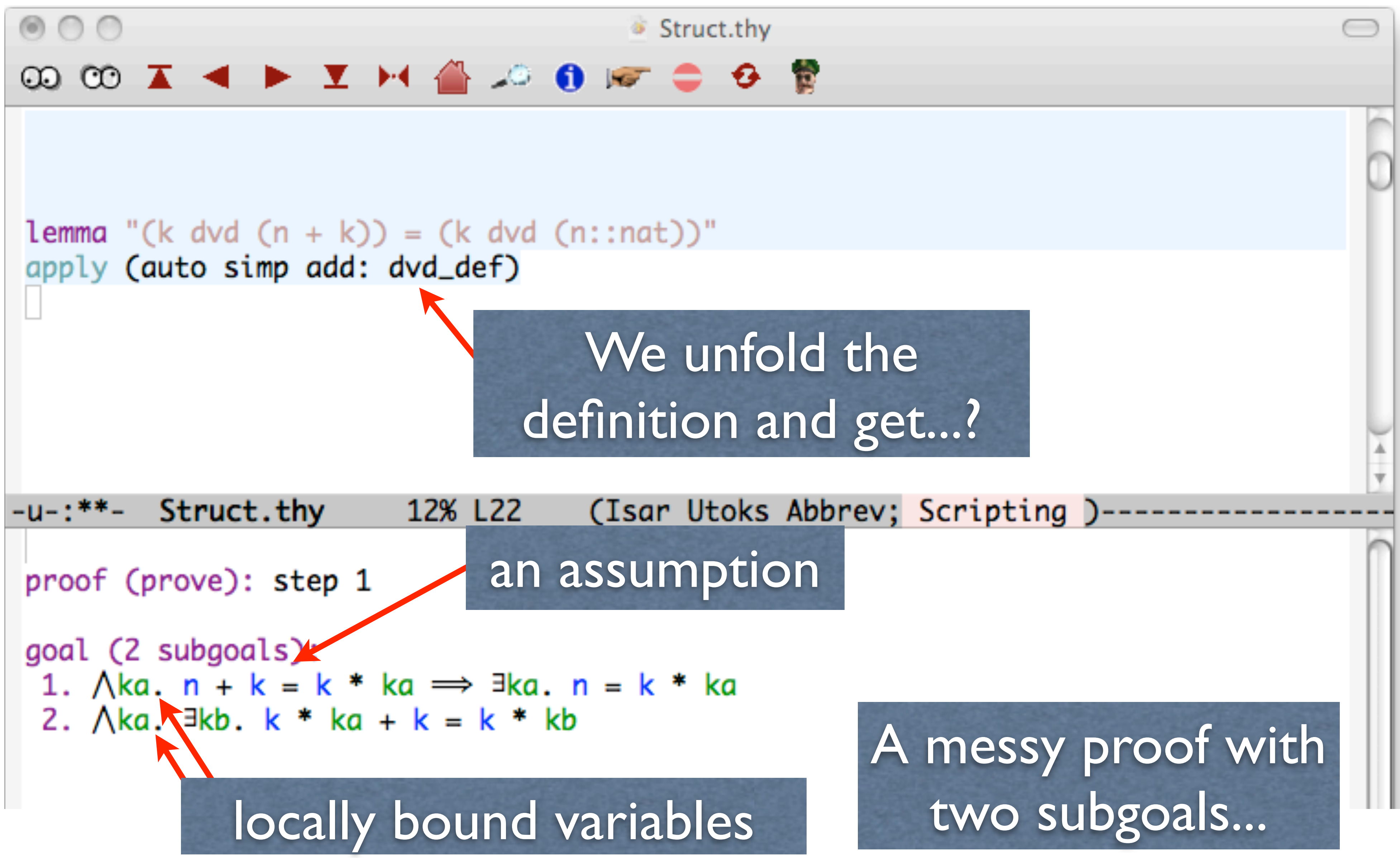
Other relevant publications are available here: `http://www4.in.tum.de/~urbanc/publications.html`

Interactive Formal Verification
9: Structured Proofs

Lawrence C Paulson
Computer Laboratory
University of Cambridge

A Proof about "Divides"

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

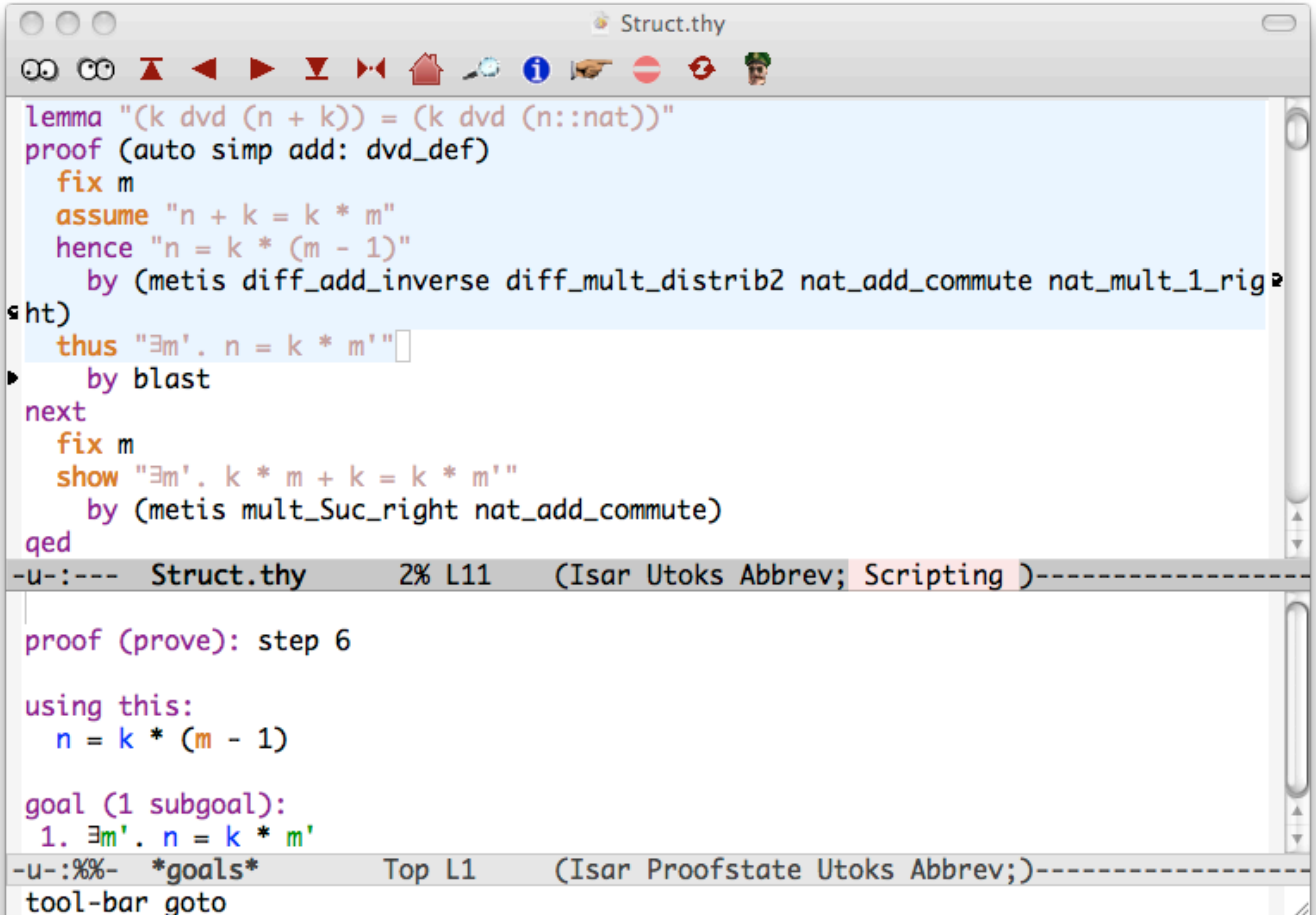


Complex Subgoals

- 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.

The old-fashioned tactics mentioned above, such as rule_tac, are described in the *Tutorial*, particularly from section 5.7 onwards.

A Structured Proof



But how do you write them?

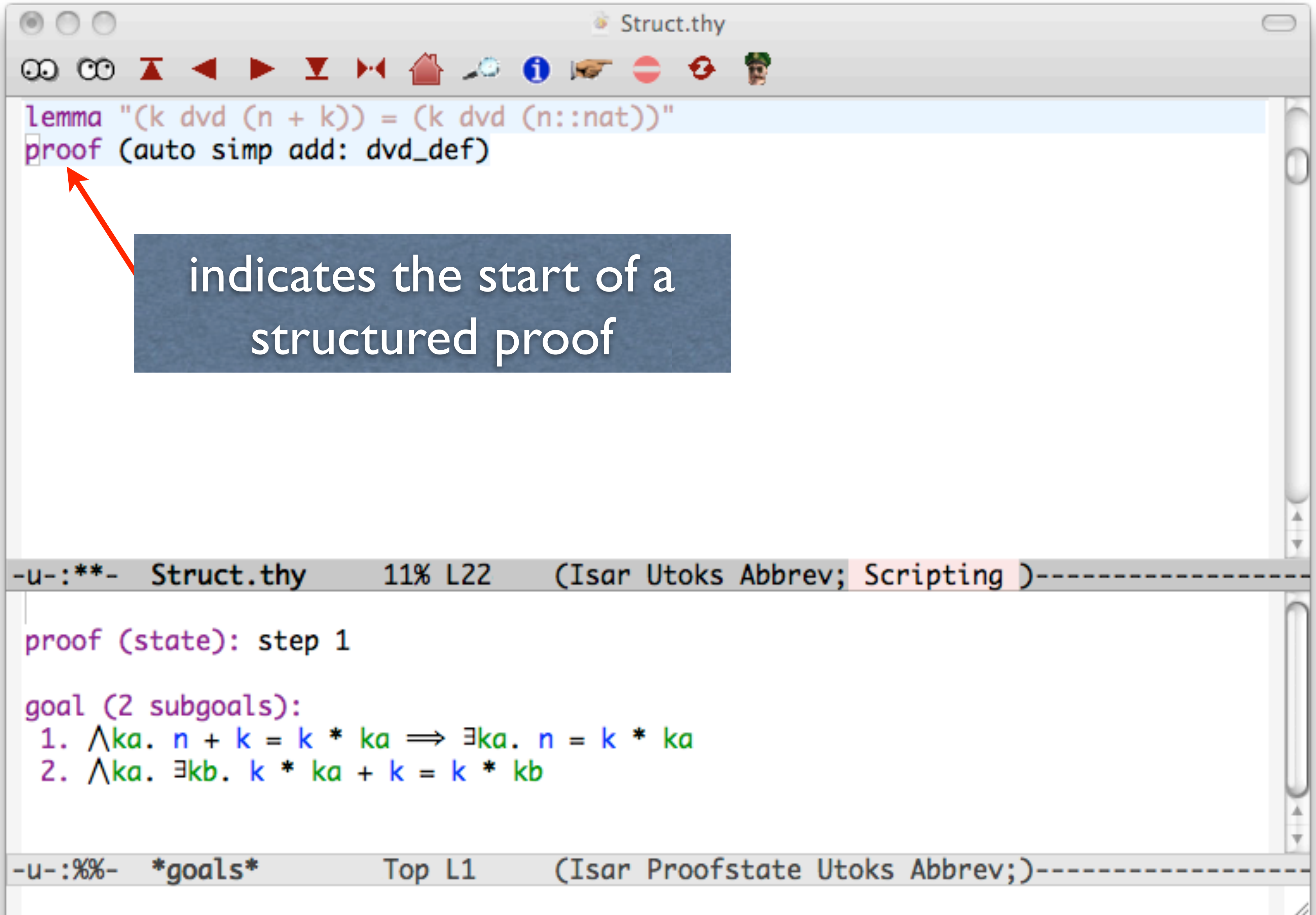
The Elements of Isar

- A *proof context* holds a 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.

The *Tutorial* has little to say about structured proofs. Separate introductions exist, for example, “A Tutorial Introduction to Structured Isar Proofs” by Tobias Nipkow.

Structured proofs can be tricky to write at first. Interaction with proof General is essential: it is virtually impossible to write a structured proof otherwise.

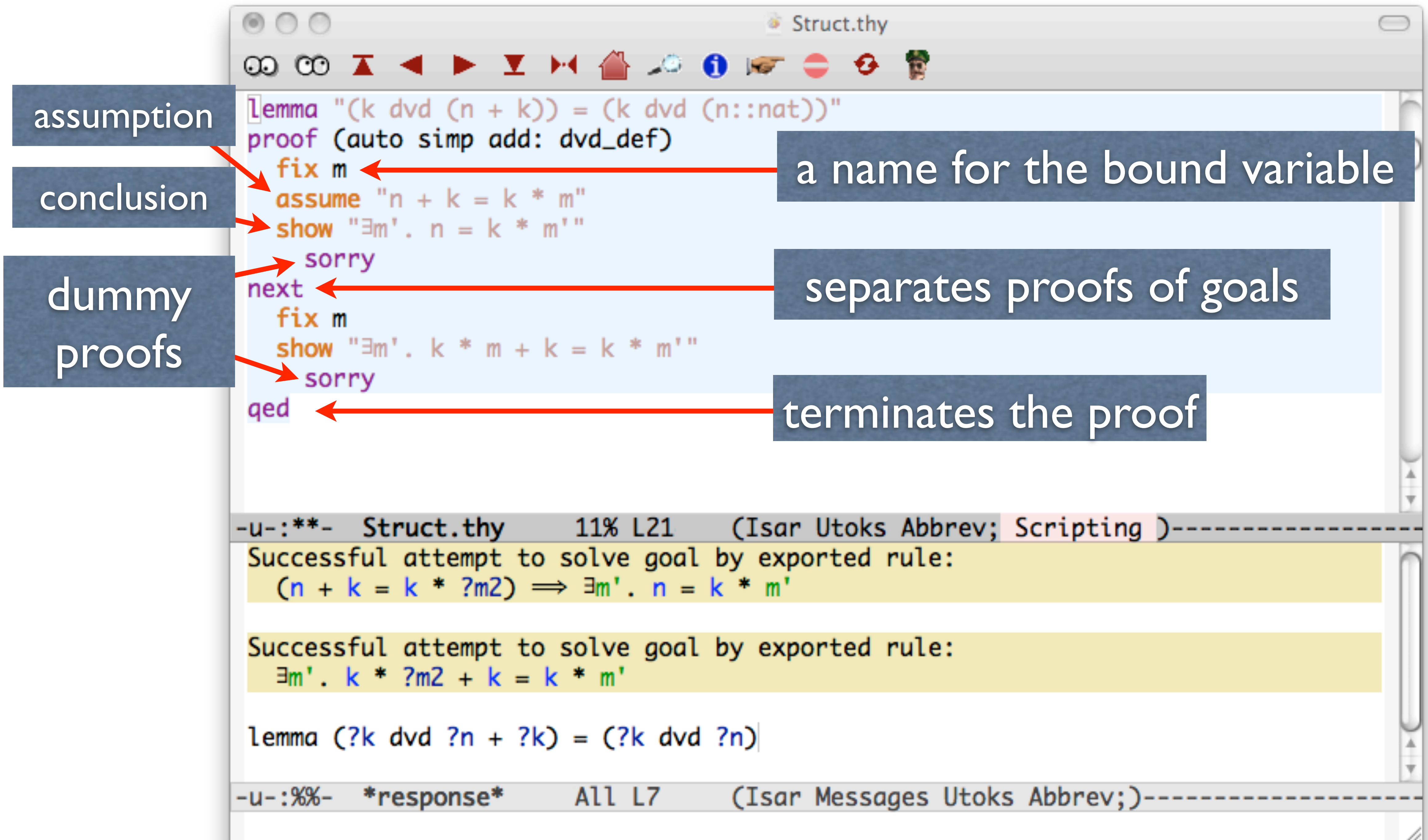
Getting Started



The simplest way to get started is as shown: applying auto with any necessary definitions. The resulting output will then dictate the structure of the final proof.

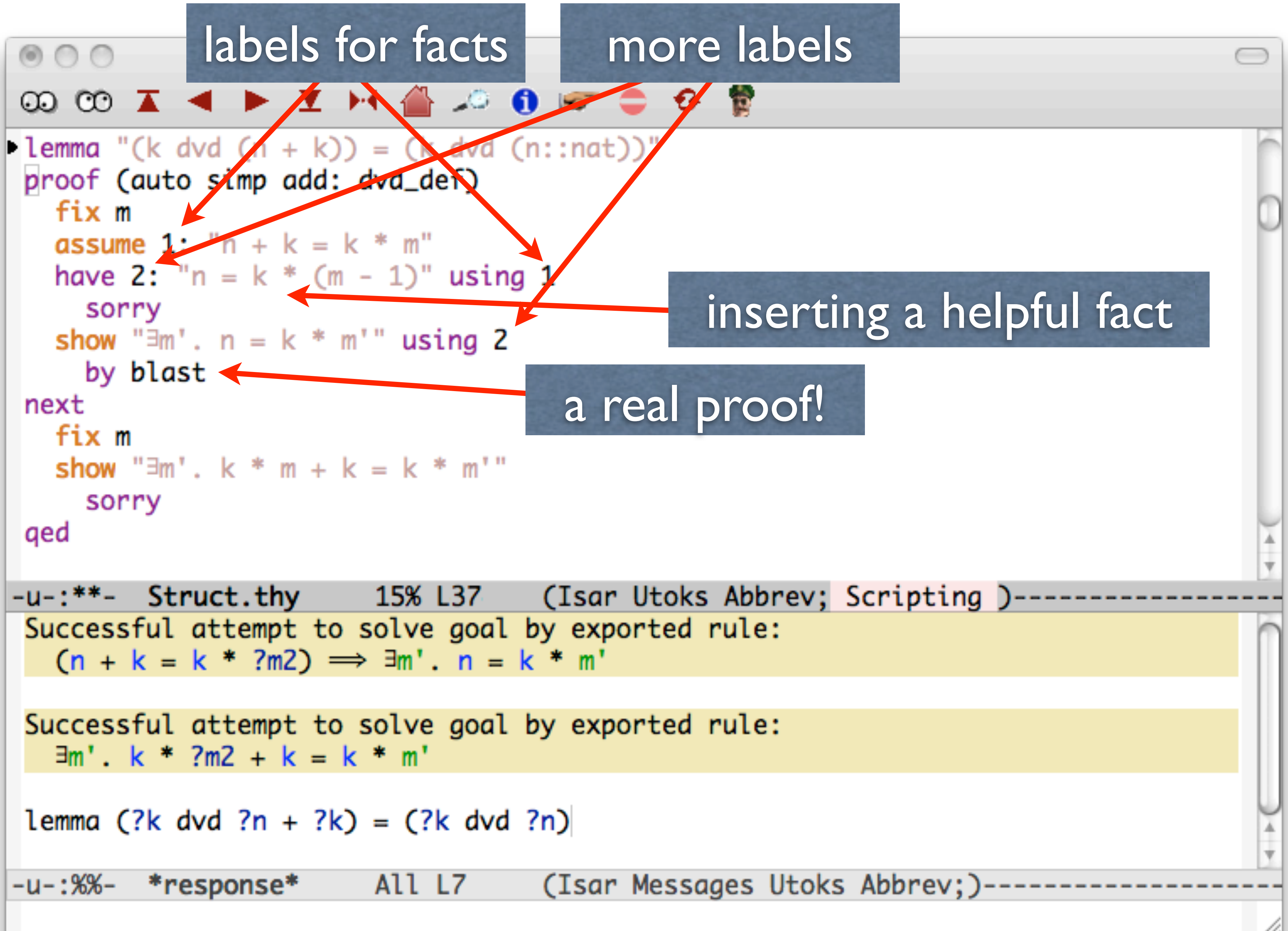
This style is actually rather fragile. Potentially, a change to `auto` could alter its output, causing a proof based around this precise output to fail. There are two ways of reducing this risk. One is to use a proof method less general than `auto` to unfold the definition of the divides relation and to perform basic logical reasoning. The other is to encapsulate the proofs of the two subgoals in local blocks that can be passed to `auto`; this approach requires a rather sophisticated use of Isar. In fact, these concerns appear to be exaggerated: proofs written in this style seldom fail.

The Proof Skeleton



We have used `sorry` to omit the proofs. These dummy proofs allow us to construct the outer shell and confirm that it fits together. We use `show` to state (and eventually prove for real!) the subgoal's conclusion. Since we have renamed the bound variable `ka` to `m`, we must rename it in the assumption and conclusions. The context that we create with `fix`/`assume`, together with the conclusion that we state with `show`, must agree with the original subgoal. Otherwise, Isabelle will generate an error message.

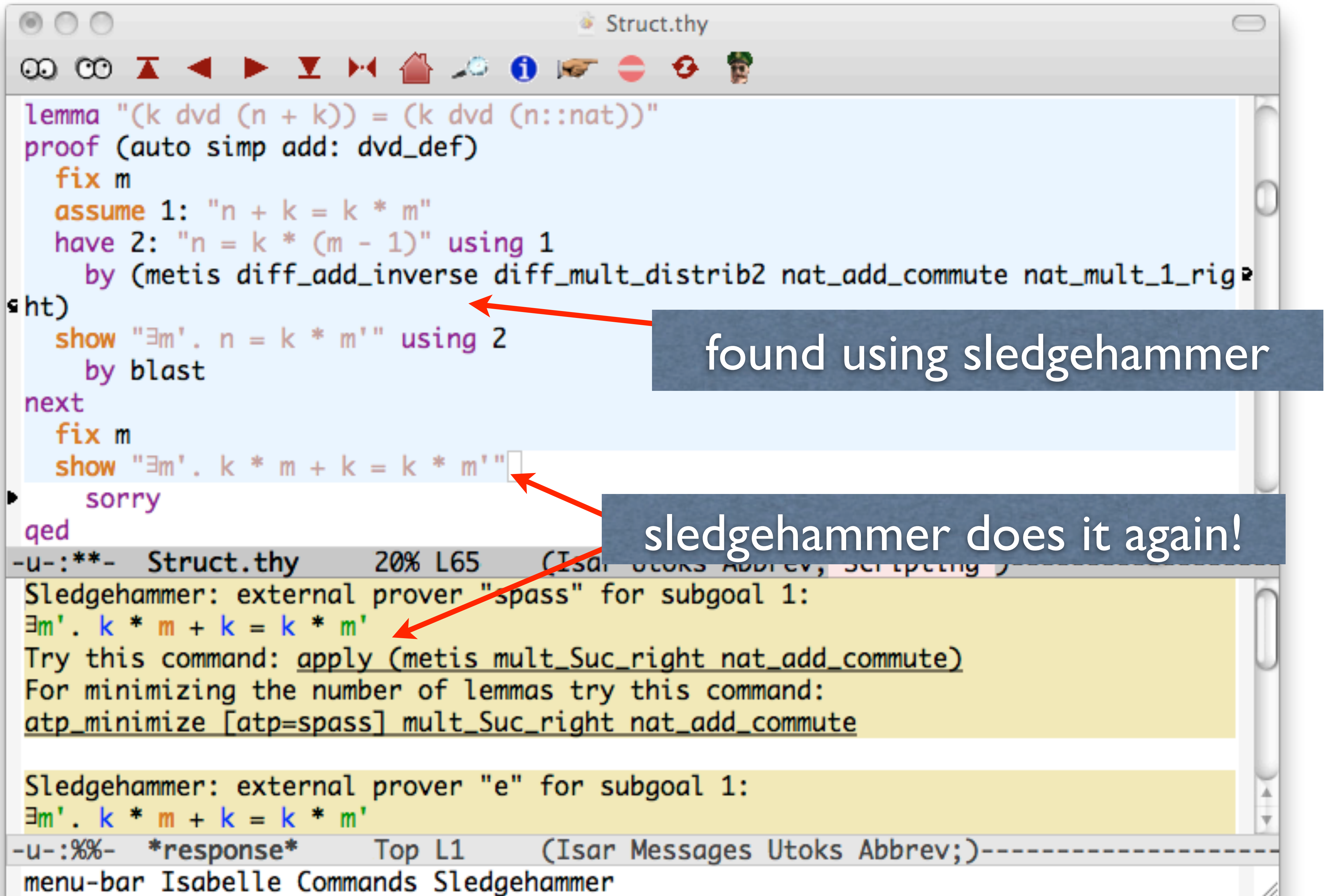
Fleshing Out that Skeleton



Looking at the first subgoal, we see that it would help to transform the assumption to resemble the body of the quantified formula that is the conclusion. Proving that conclusion should then be trivial, because the existential witness (`m-1`) is explicit. We use `sorry` to obtain this intermediate result, then confirm that the conclusion is provable from it using `blast`. Because it is a one line proof, we write it using “by”. It is permissible to insert a string of “`apply`” commands followed by “`done`”, but that looks ugly.

We give labels to the assumption and the intermediate result for easy reference. We can then write “`using 1`”, for example, to indicate that the proof refers to the designated fact. However, referring to the previous result is extremely common, and soon we shall streamline this proof to eliminate the labels. Also, labels do not have to be integers: they can be any Isabelle identifiers.

Completing the Proof



We have narrowed the gaps, and now sledgehammer can fill them. Replacing the last "sorry" completes the proof.

There is of course no need to follow this sort of top-down development. It is one approach that is particularly simple for beginners.

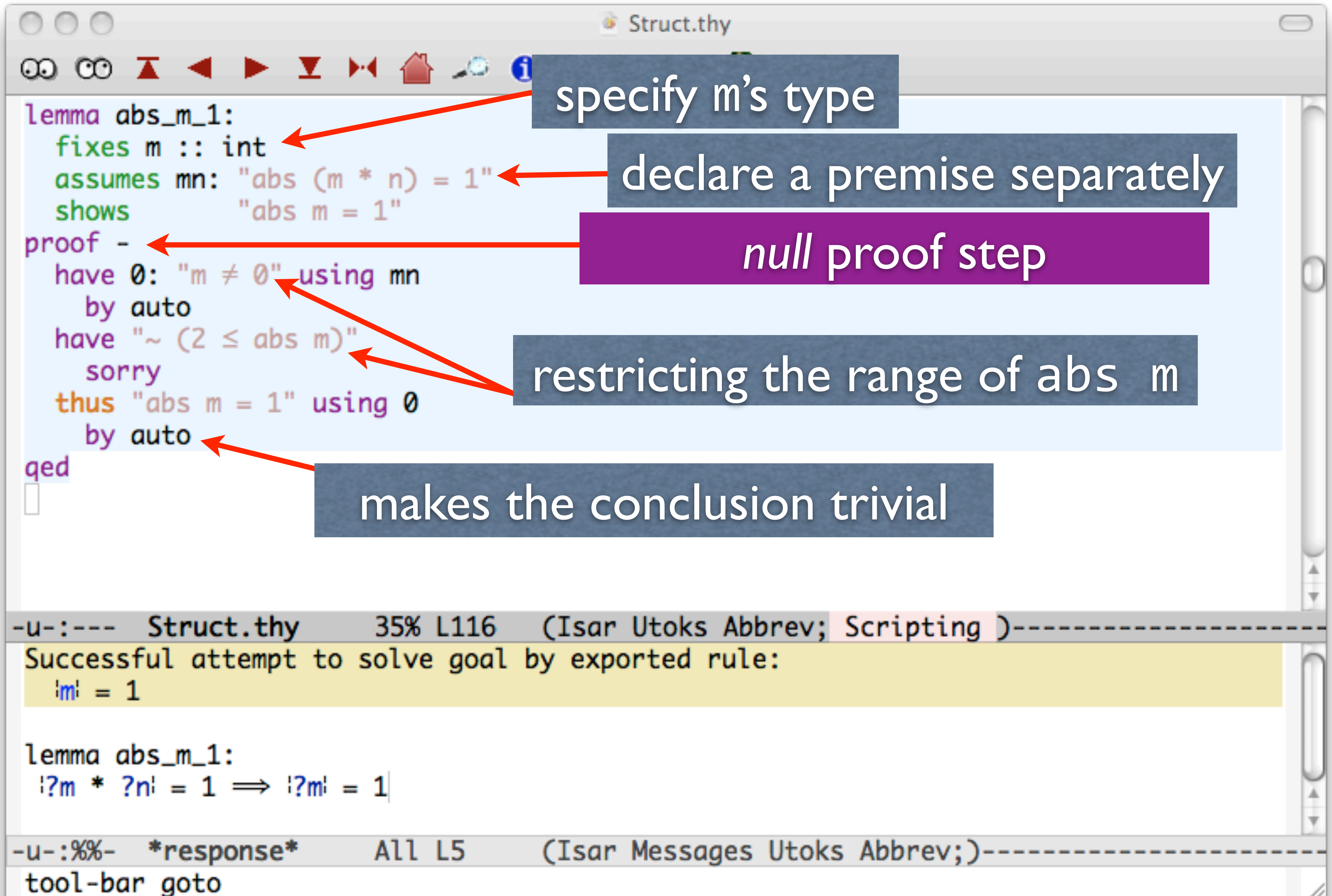
Streamlining the Proof

```
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
- There are numerous other tricks of this sort!

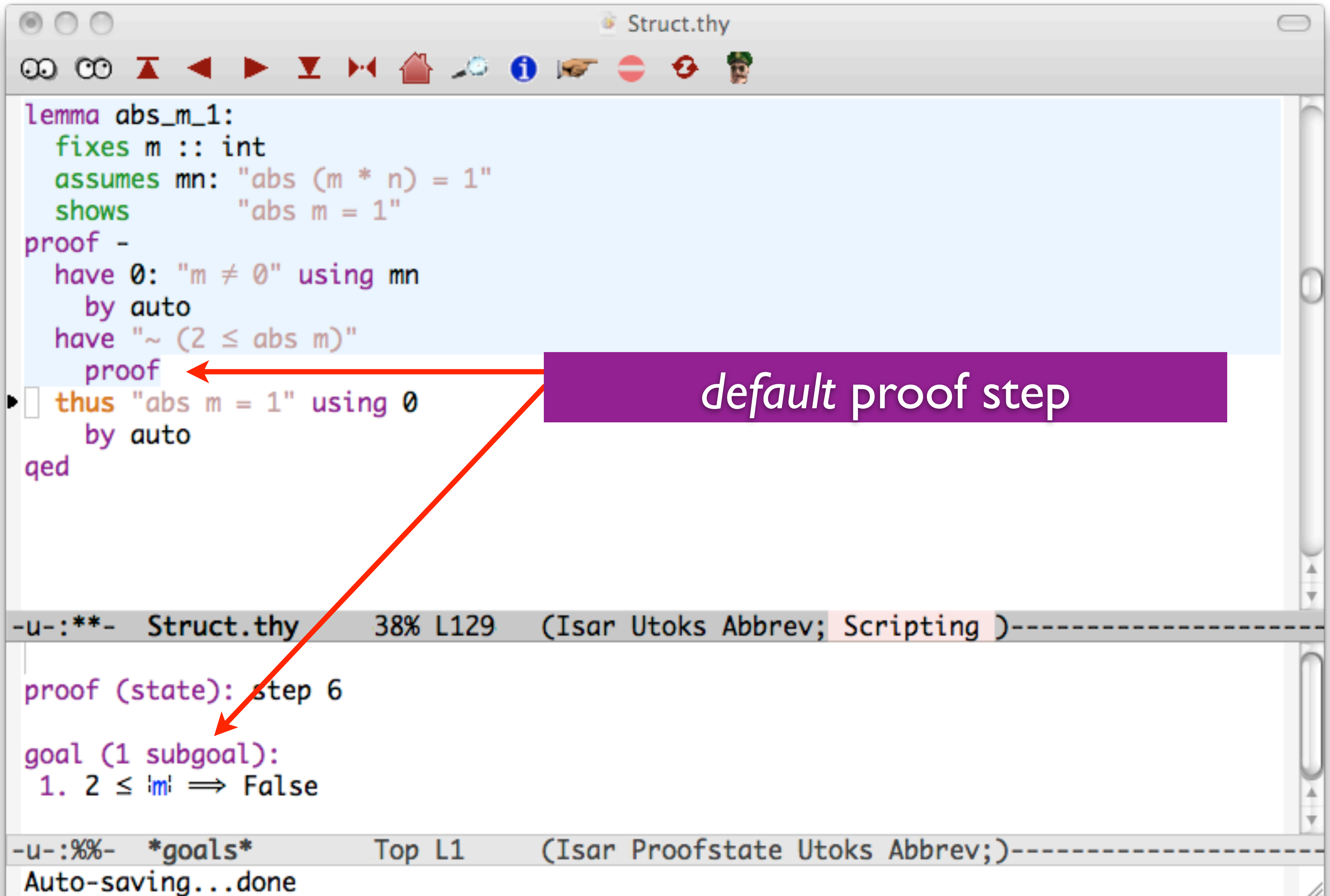
Another Proof Skeleton



This is an example of an obvious fact is proof is not obvious. Clearly m≠0, since otherwise m*n=0. If we can also show that |m|≥2 is impossible, then the only remaining possibility is |m|=1.

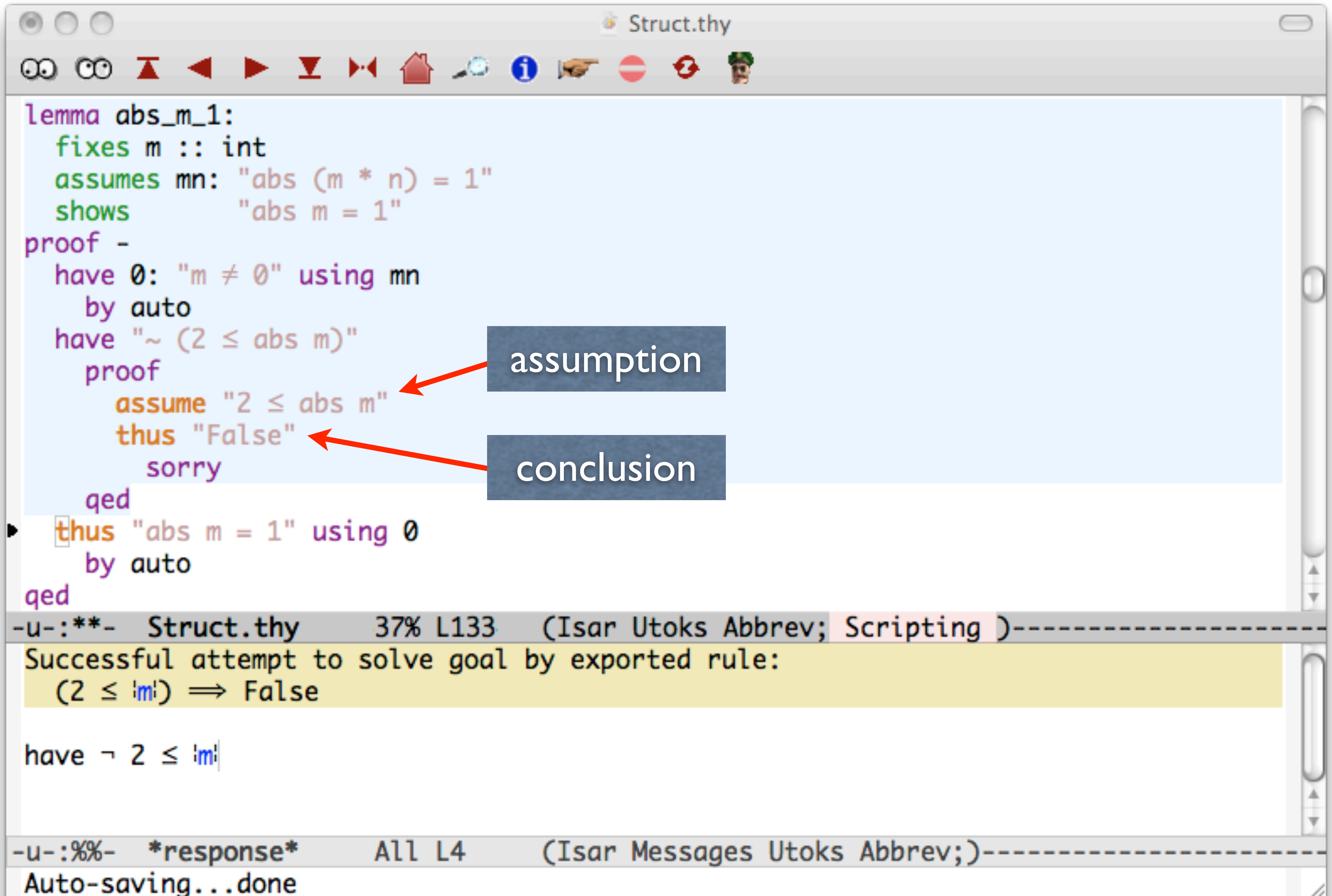
In this example, `auto` can do nothing. No proof steps are obvious from the problem's syntax. So the Isar proof begins with "-", the null proof. This step does nothing but insert any "pending facts" from a previous step (here, there aren't any) into the proof state. It is quite common to begin with "`proof -`".

Starting a Nested Proof



To begin with "`proof`" (not to be confused with "`proof -`") applies a default proof method. In theory, this method should be appropriate for the problem, but in practice, it is often unhelpful. The default method is determined by elementary syntactic criteria. For example, the formula "¬ (2 ≤ abs m)" begins with a negation sign, so the default method applies the corresponding logical inference: it reduces the problem to proving False under the assumption 2 ≤ abs m.

A Nested Proof Skeleton



Proofs can be nested to any depth. The assumptions and conclusions of each nested proof are independent of one another. The usual scoping rules apply, and in particular the facts `mn` and `0` are visible within this inner scope.

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
```

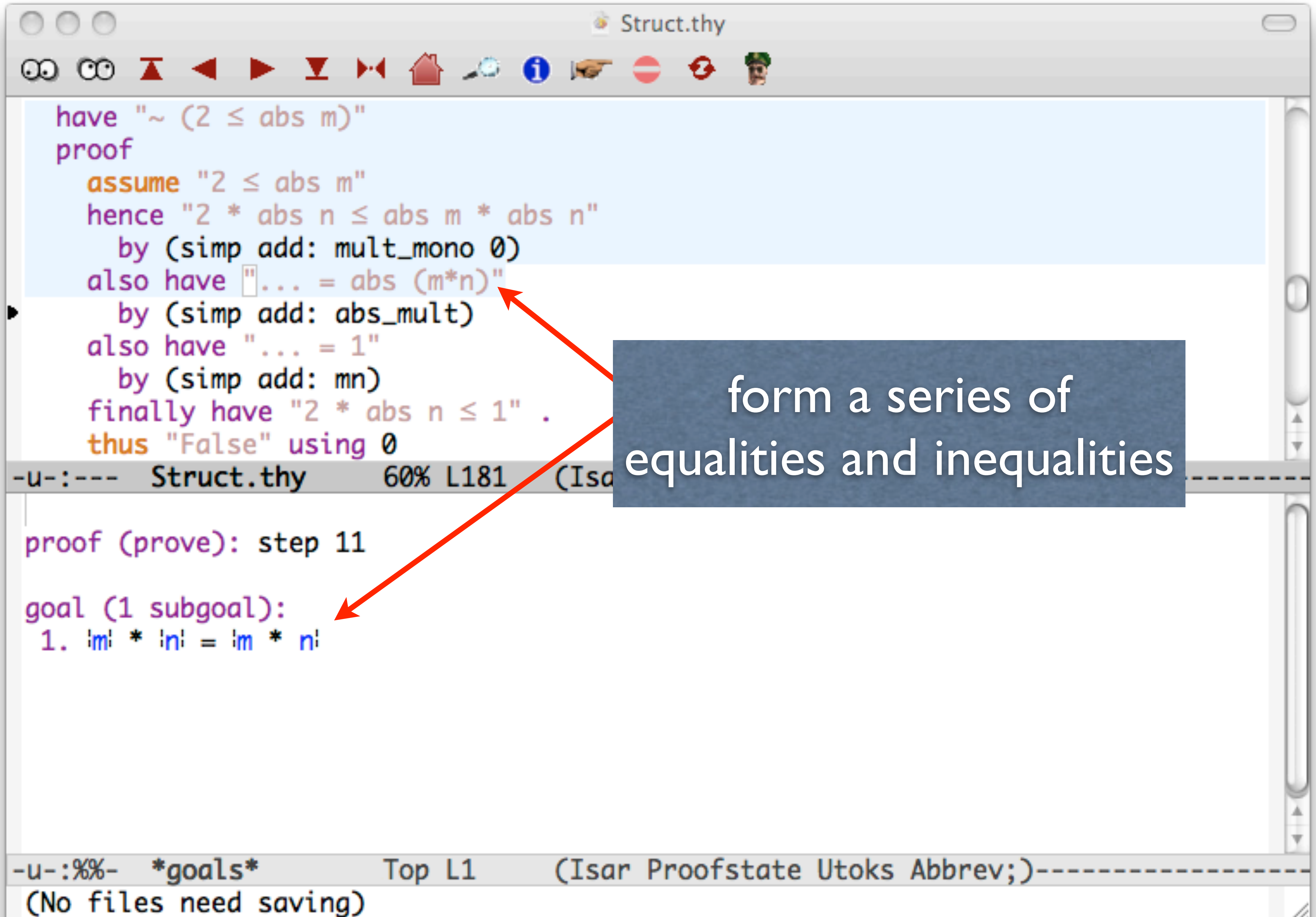
a chain of steps leads to contradiction

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

This example is typical of a structured proof. From the assumption, 2 ≤ abs m, we deduce a chain of consequences that become absurd. We connect one step to the next using “`hence`”, except that we must introduce the conclusion using “`thus`”.

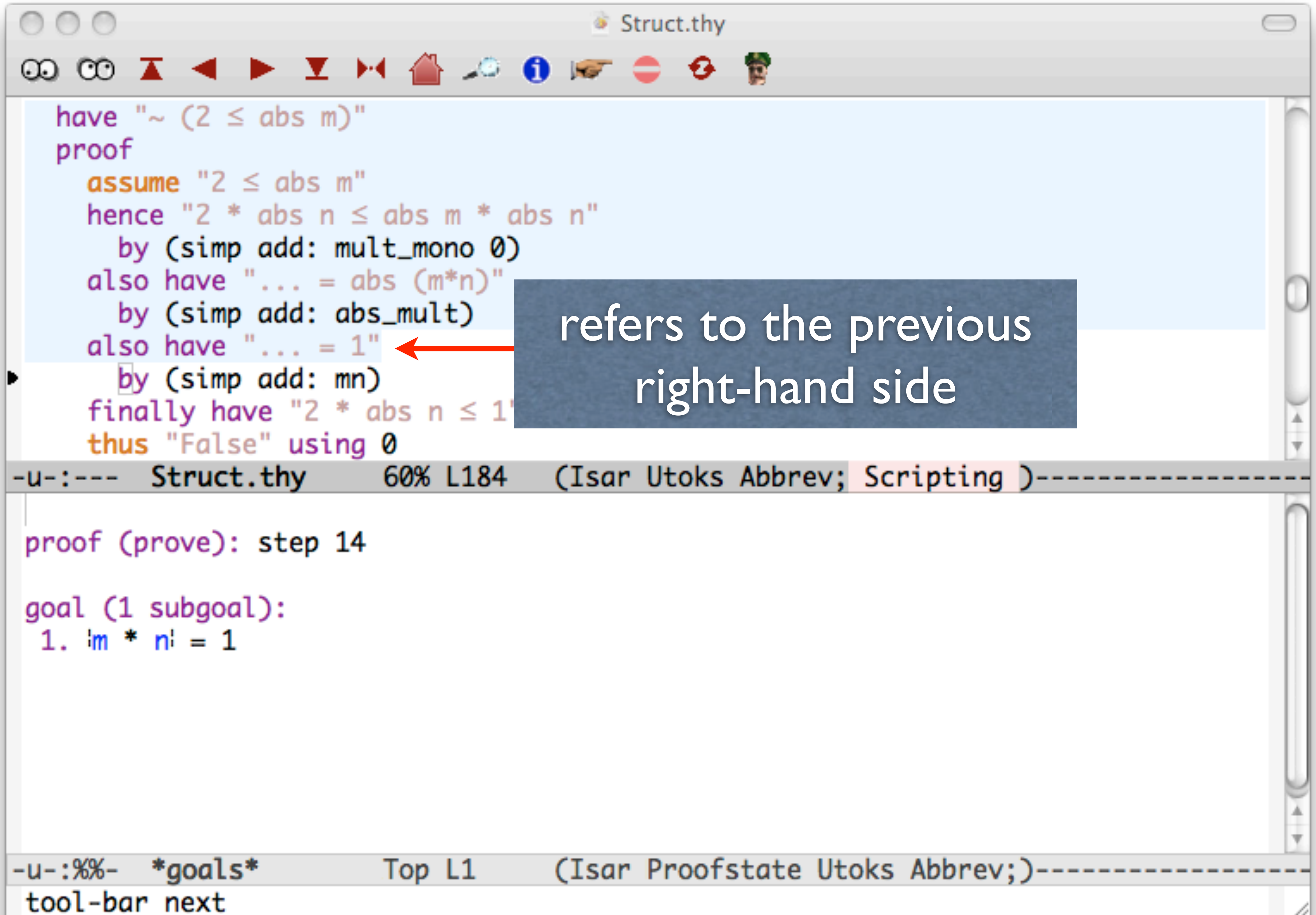
Note that we have beefed up the fact “0” from simply m≠0 to include as well n≠0, which we need to obtain a contradiction from 2 × abs n ≤ 1. In fact, “0” here denotes a list of facts.

Calculational Proofs

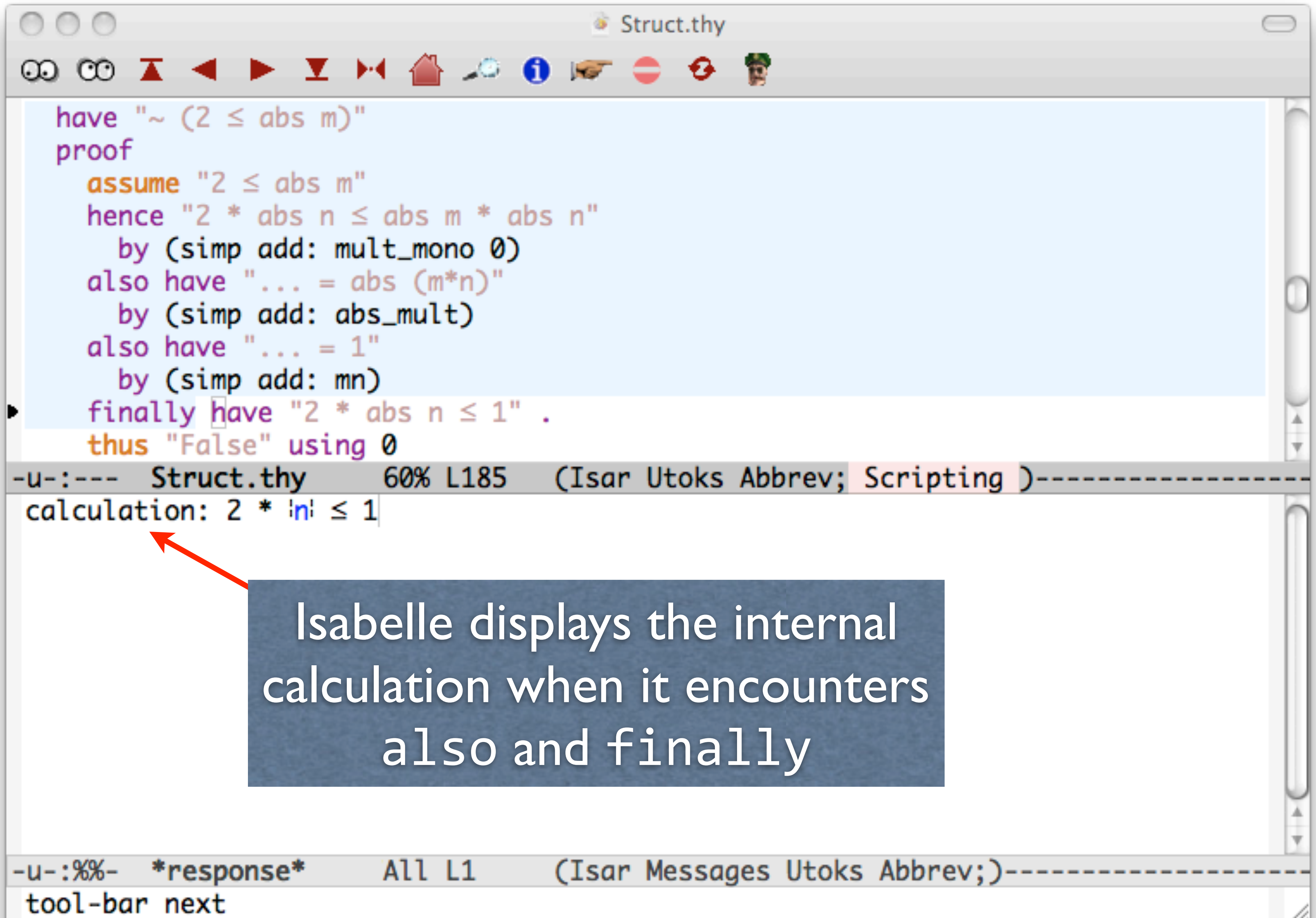


The chain of reasoning in the previous proof holds by transitivity, and in normal mathematical discourse would be written as a chain of inequalities and inequalities. Isar supports this notation.

The Next Step



The Internal Calculation



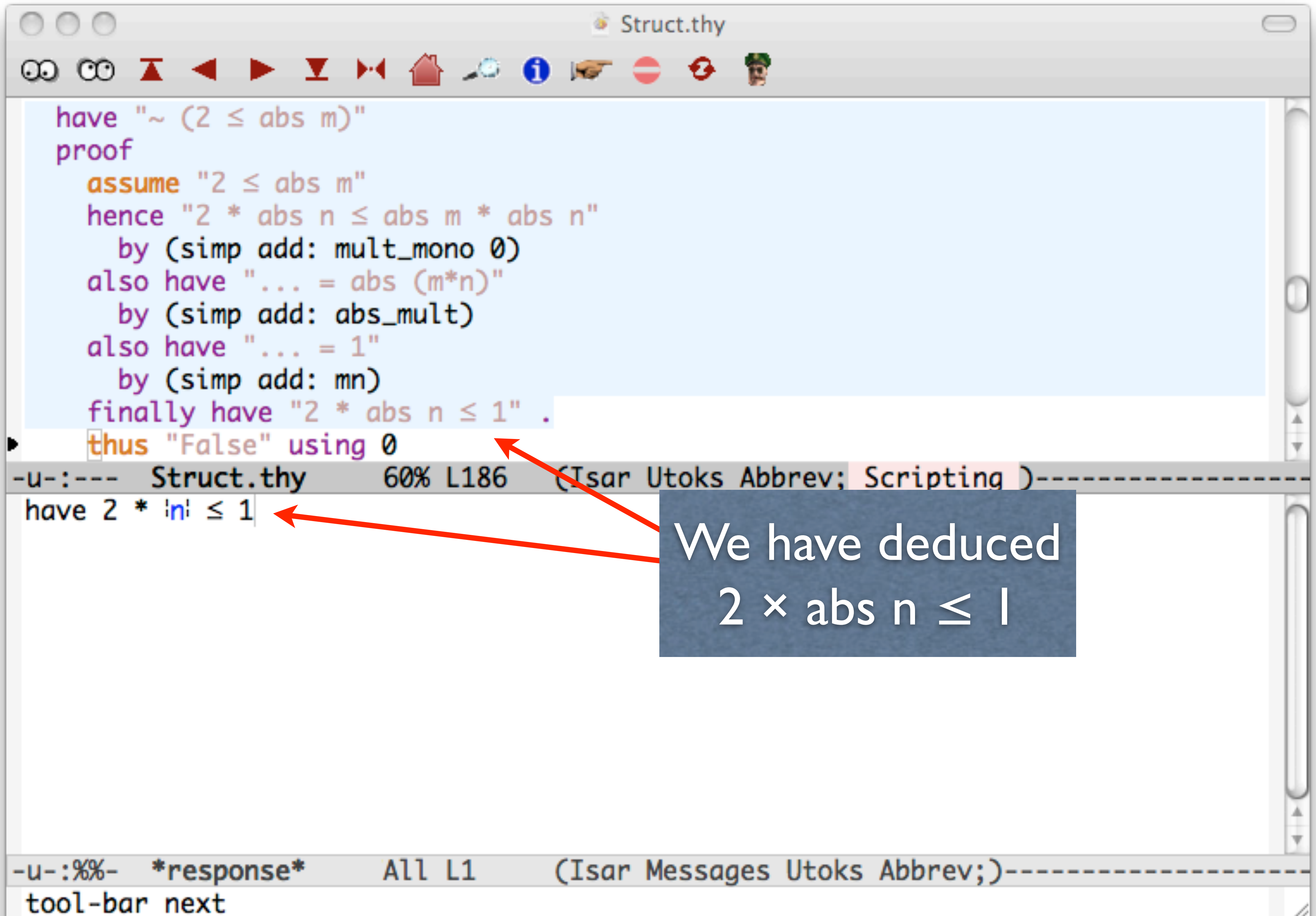
Use “`also`” to attach a new link to the chain, extending the calculation. Use “`finally`” to refer to the calculation itself. It is usual for the proof script merely to repeat explicitly what this calculation should be, as shown above. If this is done, the proof is trivial and is written in Isar as a single dot (`.`).

We could instead avoid that repetition and reach the contradiction directly as follows:

```
    also have "... = 1"
     by (simp add: mn)
  finally show "False" using 0
     by auto
```

Internally, this proof is identical to the previous one. It merely differs in appearance, not bothering to note that 2 × abs n ≤ 1 has been derived.

Ending the Calculation



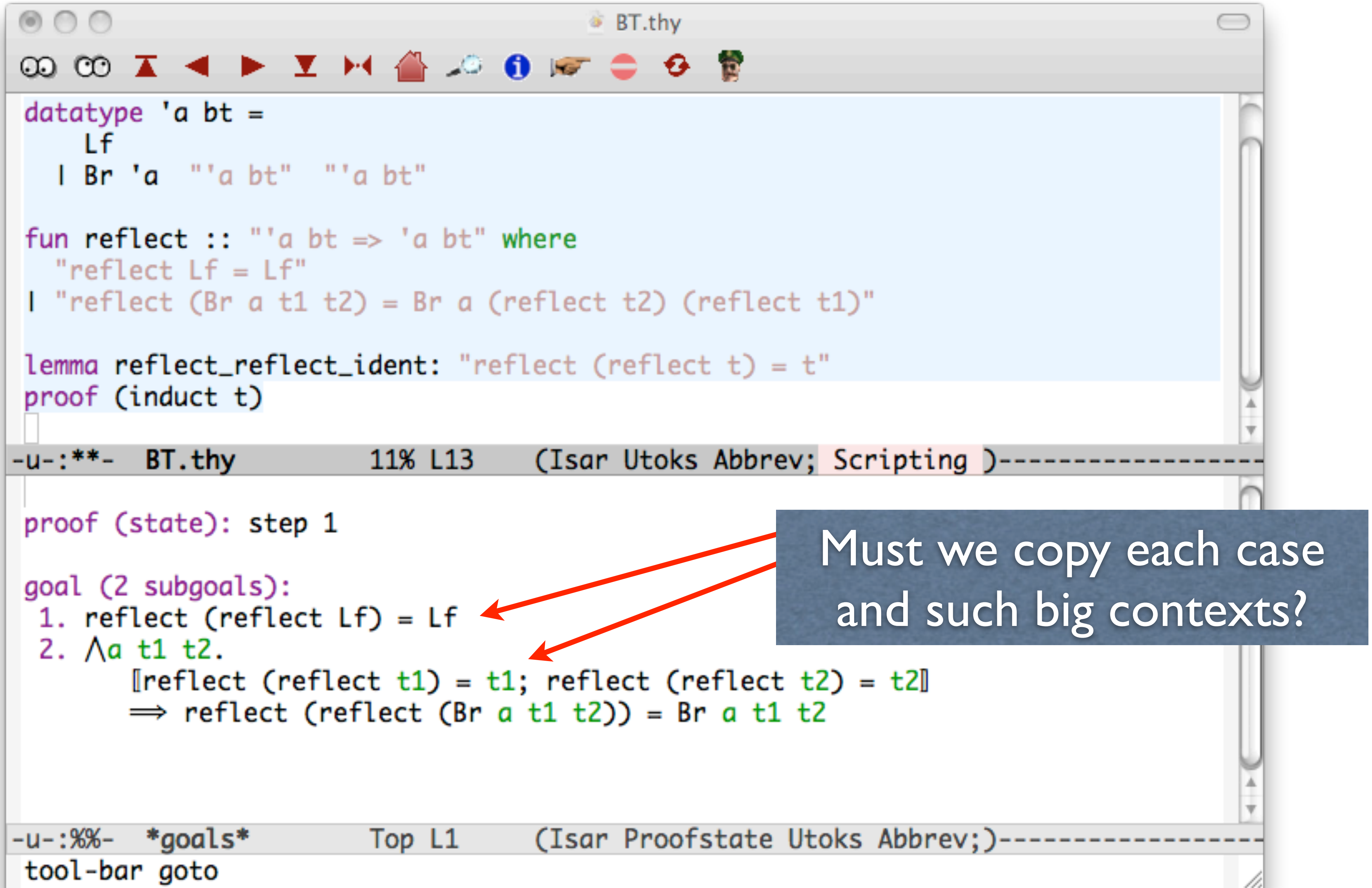
Structure of a Calculation

- 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 `(.)`

Interactive Formal Verification
10: Structured Induction Proofs

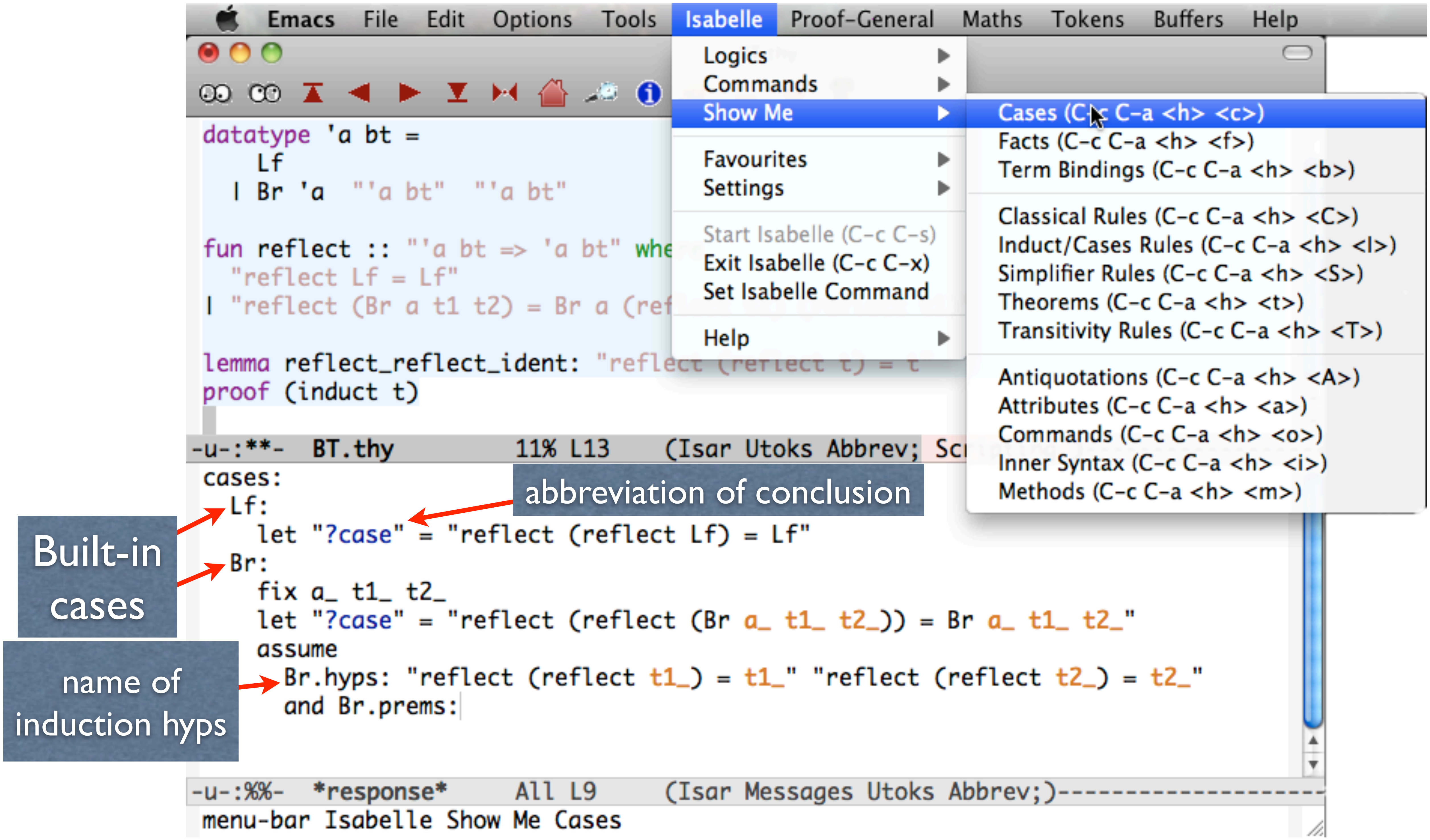
Lawrence C Paulson
Computer Laboratory
University of Cambridge

A Proof about Binary Trees



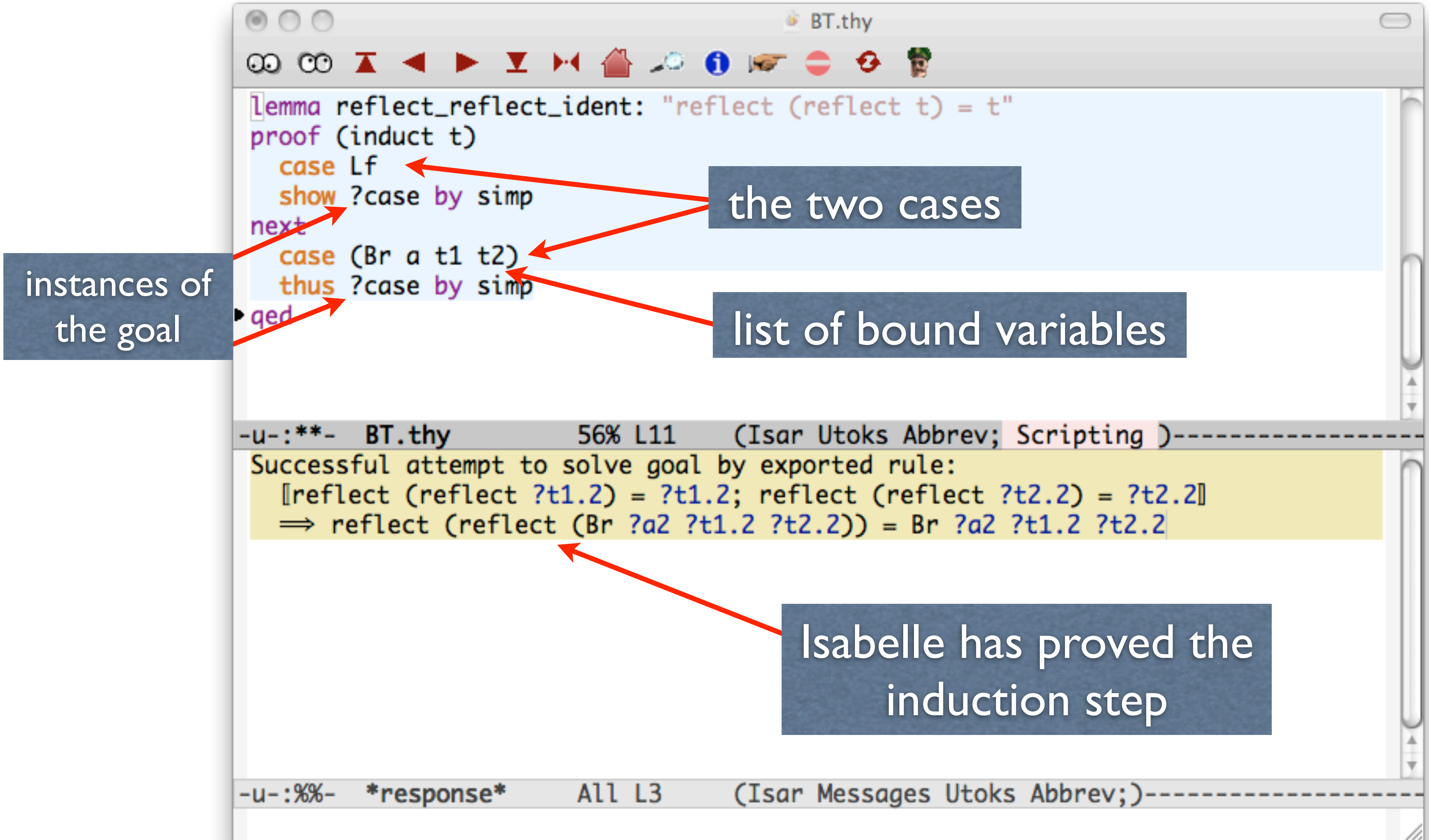
Inductive proofs frequently involve several subgoals, some of them with multiple assumptions and bound variables. Creating an Isar proof skeleton from scratch would be tiresome, and the resulting proof would be quite lengthy.

Finding Predefined Cases



Many induction rules have attached cases designed for use with Isar. By referring to such a case, a proof script implicitly introduces the contexts shown above. There are placeholders for the bound variables (specific names must be given). Identifiers are introduced to denote induction hypotheses and other premises that accompany each case. Also, the identifier `?case` is introduced to abbreviate the required instance of the induction formula.

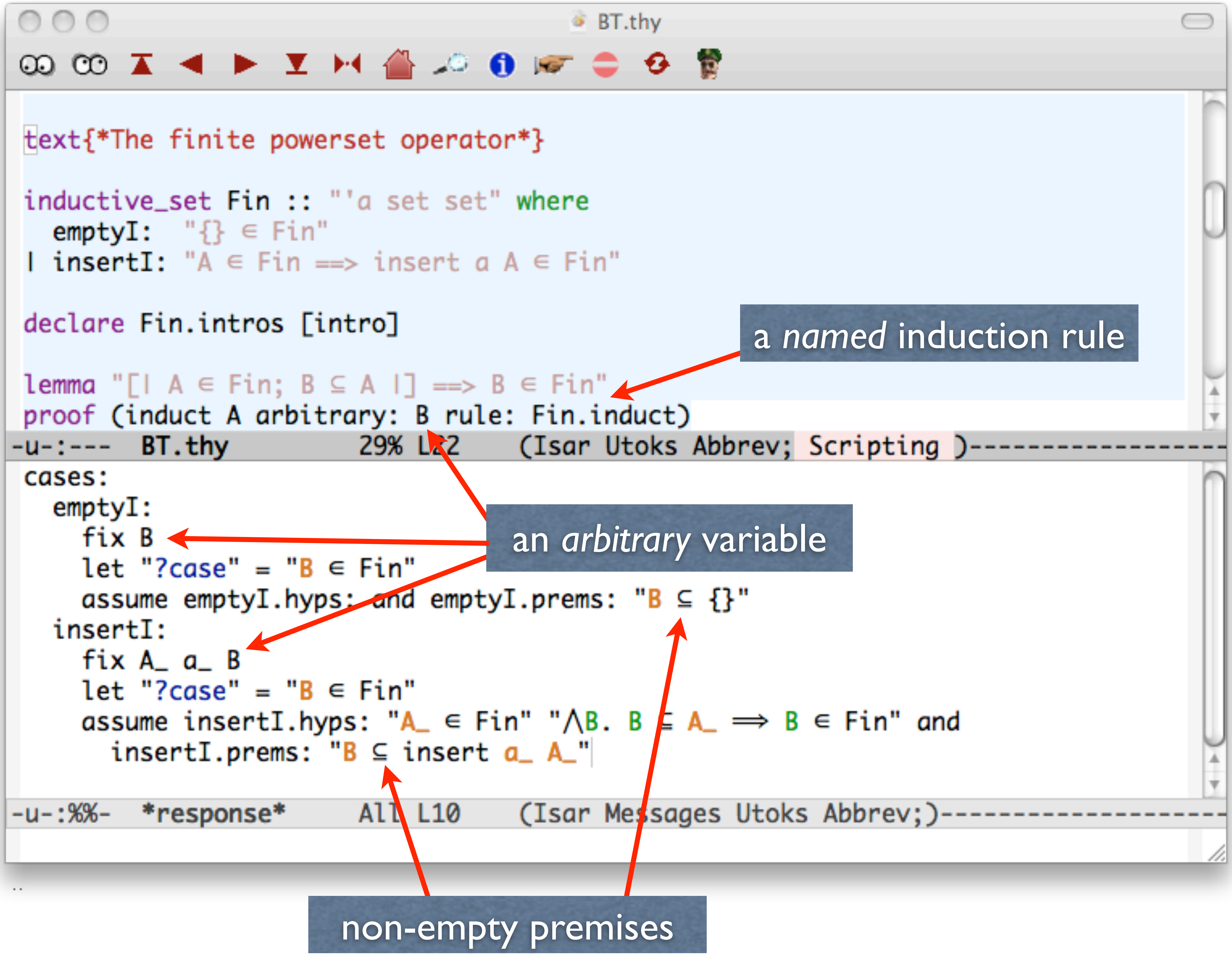
The Finished Proof



With all these abbreviations, the induction formula does not have to be repeated in its various instances. The instances that are to be proved are abbreviated as `?case`; they (and the induction hypotheses) are automatically generated from the supplied list of bound variables.

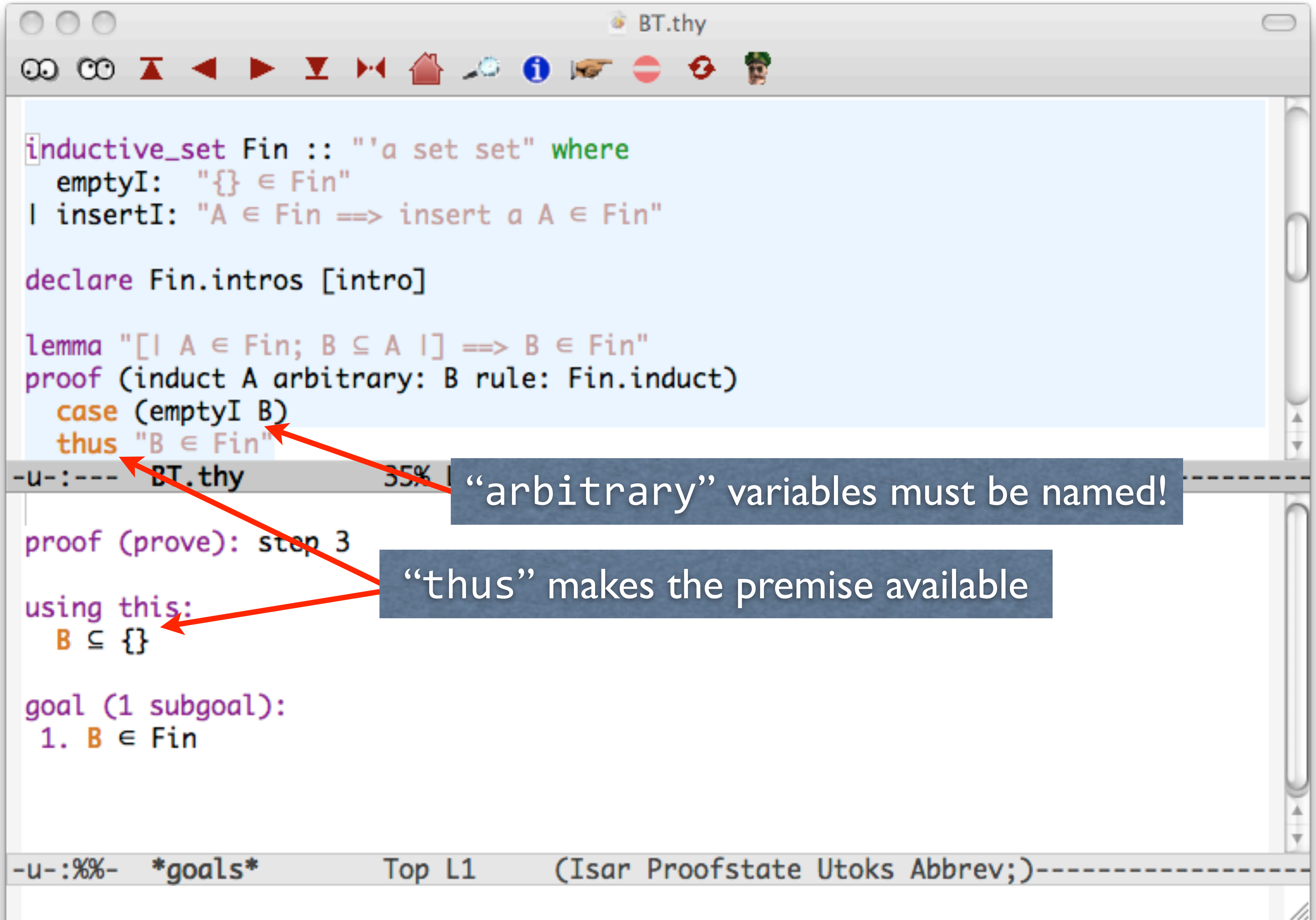
Observe the use of "`thus`" rather than "`show`" in the inductive case, thereby providing the induction hypotheses to the method. In a more complicated proof, these hypotheses can be denoted by the identifier `Br.hyps`.

A More Sophisticated Proof



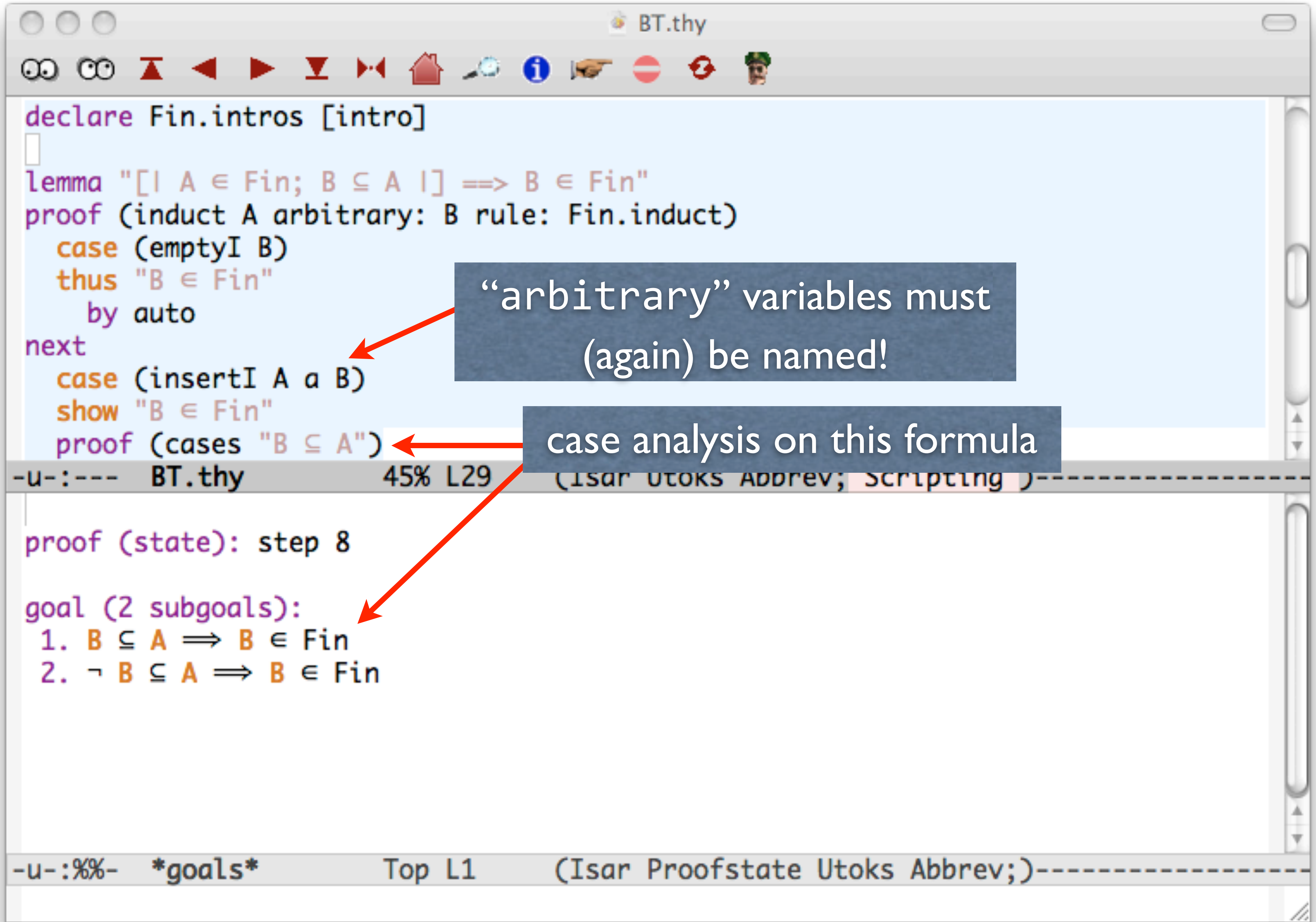
An inductive definition generates an induction rule with one case (correspondingly named) for each introduction rule. This particular proof requires the variable B to be taken as arbitrary, which means, universally quantified: it becomes an additional bound variable in each case. This proof also carries along a further premise, B⊆A, instances of which are attached to both subgoals.

Proving the Base Case



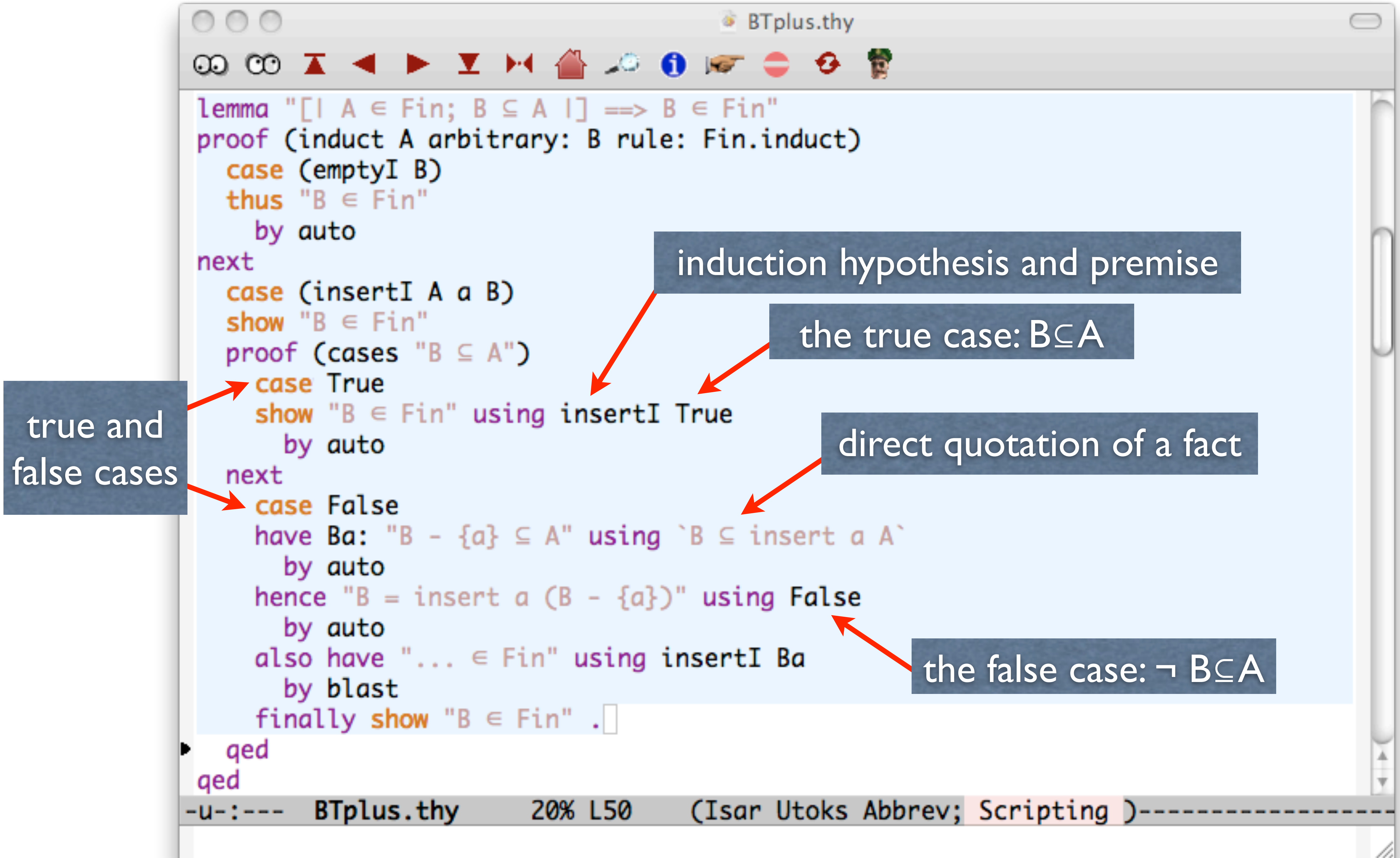
The base case would normally be just `emptyI`. But here, there is an additional bound variable. Note that we could have written, for example, `(emptyI C)` and Isabelle would have adjusted everything to use C instead of B.

A Nested Case Analysis



Here we know B ⊆ insert a A, as it is the inherited premise of this case. But do we in fact know B⊆A?

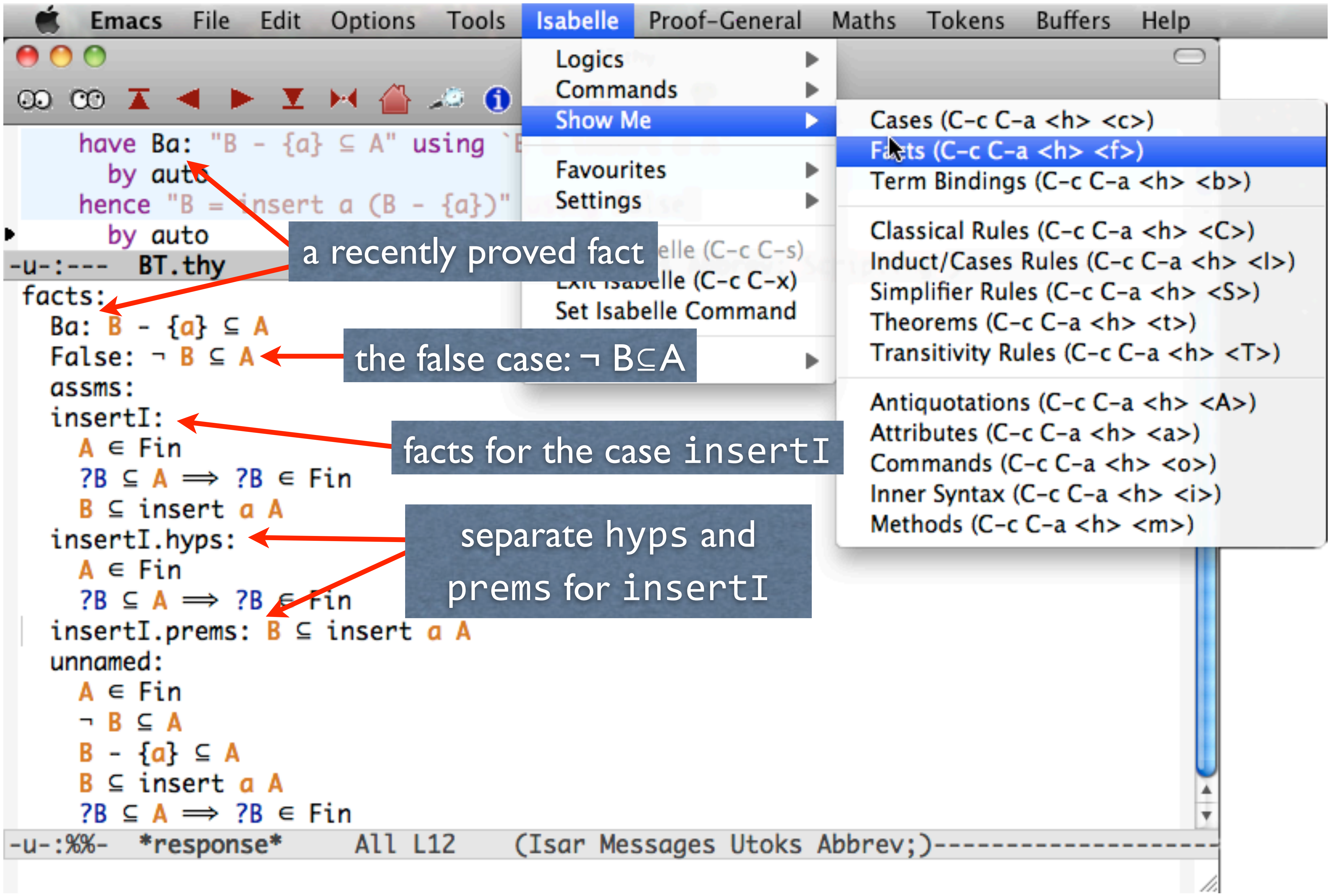
The Complete Proof



Here is an outline of the proof. If B⊆A, then it is trivial, as we can immediately use the induction hypothesis. If not, then we apply the induction hypothesis to the set B-{a}. We deduce that B-{a} ∈ Fin, and therefore B = insert a (B-{a}) ∈ Fin.

This proof script contains many references to facts. The facts attached to the case of an inductive proof or case analysis are denoted by the name of that case, for example, insertI, True or False. We can also refer to a theorem by enclosing the actual theorem statement in backward quotation marks. We see this above in the proof of B-{a} ⊆ A.

Which Theorems are Available?



Existential Claims: "obtain"

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

Frequently, our reasoning involves quantities (such as j above) that are known to satisfy certain properties. Here, the "divides" premise implies the existence of a divisor, j. What Isabelle does internally can be difficult to understand, especially if the proof fails. It proves a theorem having the general form of an elimination rule, which in the premise introduces one or more bound variables: the variables that we "obtain".

Continuing the Proof

BT.thy

```
lemma dvd_mult_cancel:
  fixes k::nat
  assumes dv: "k*m dvd k*n" and "0<k"
  shows "m dvd n"
proof -
  obtain j where "k*n = (k*m)*j" using dv
    by (auto simp add: dvd_def)
  hence "k*n = k*(m*j)"
    by (simp add: mult_ac)
  hence "n = m*j" using `0<k`
    by auto
```

```
-u-:---  BT.thy         62% L62     (Isar Utoks Abbrev; Scripting )----------------
```

```
proof (prove): step 5

using this:
  k * n = k * m * j

goal (1 subgoal):
 1. k * n = k * (m * j)
```

we now have the key property of j

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

The Finished Proof

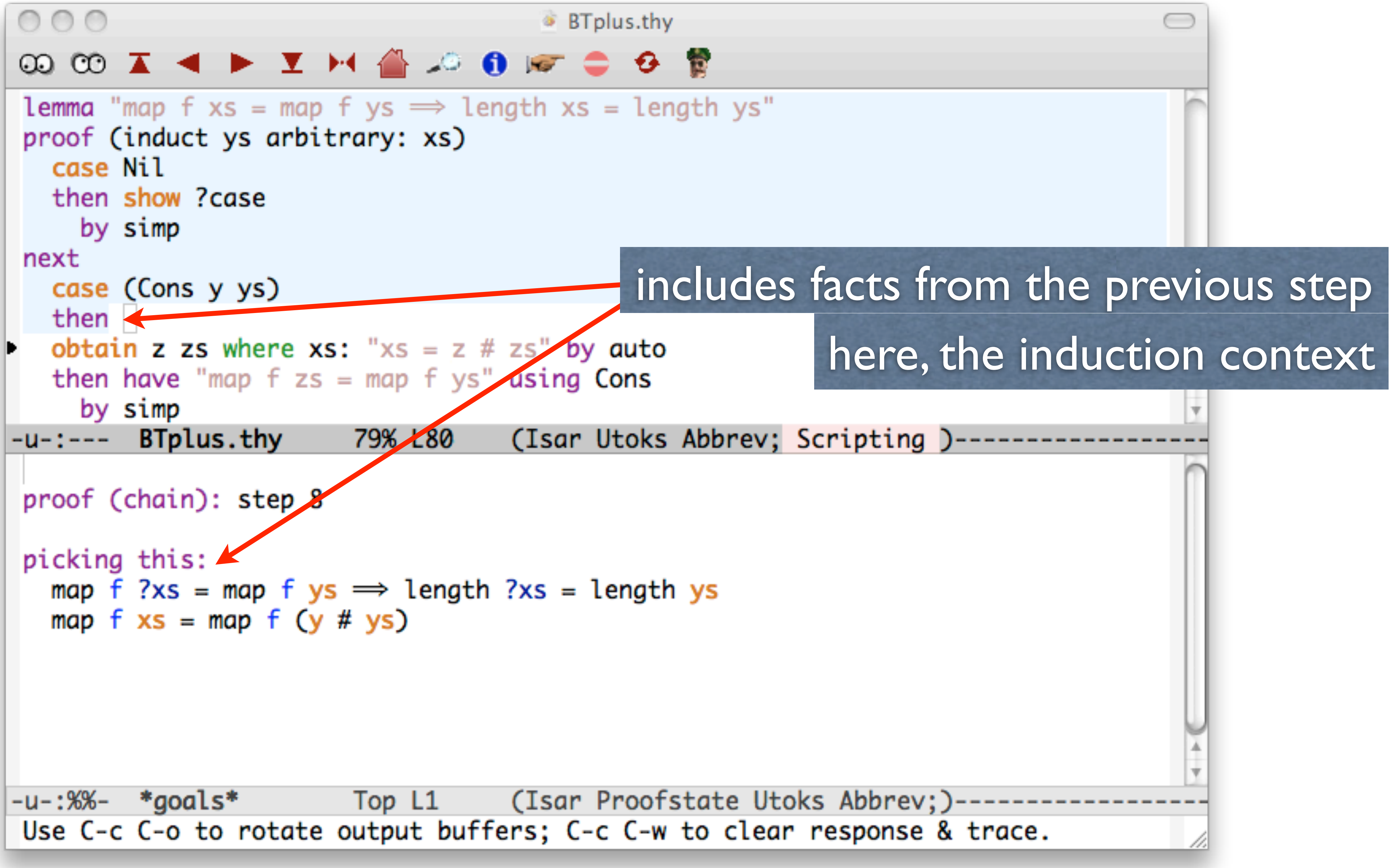
BT.thy

```
lemma dvd_mult_cancel:
  fixes k::nat
  assumes dv: "k*m dvd k*n" and "0<k"
  shows "m dvd n"
proof -
  obtain j where "k*n = (k*m)*j" using dv
    by (auto simp add: dvd_def)
  hence "k*n = k*(m*j)"
    by (simp add: mult_ac)
  hence "n = m*j" using `0<k`
    by auto
  thus "m dvd n"
    by (auto simp add: dvd_def)
qed
-u-:---  BT.thy        62% L54     (Isar Utoks Abbrev; Scripting )----------------
```

removing k

```
proof (state): step 11

this:
  m dvd n

goal:
No subgoals!
-u-:%%-  *goals*       Top L1      (Isar Proofstate Utoks Abbrev;)----------------
```

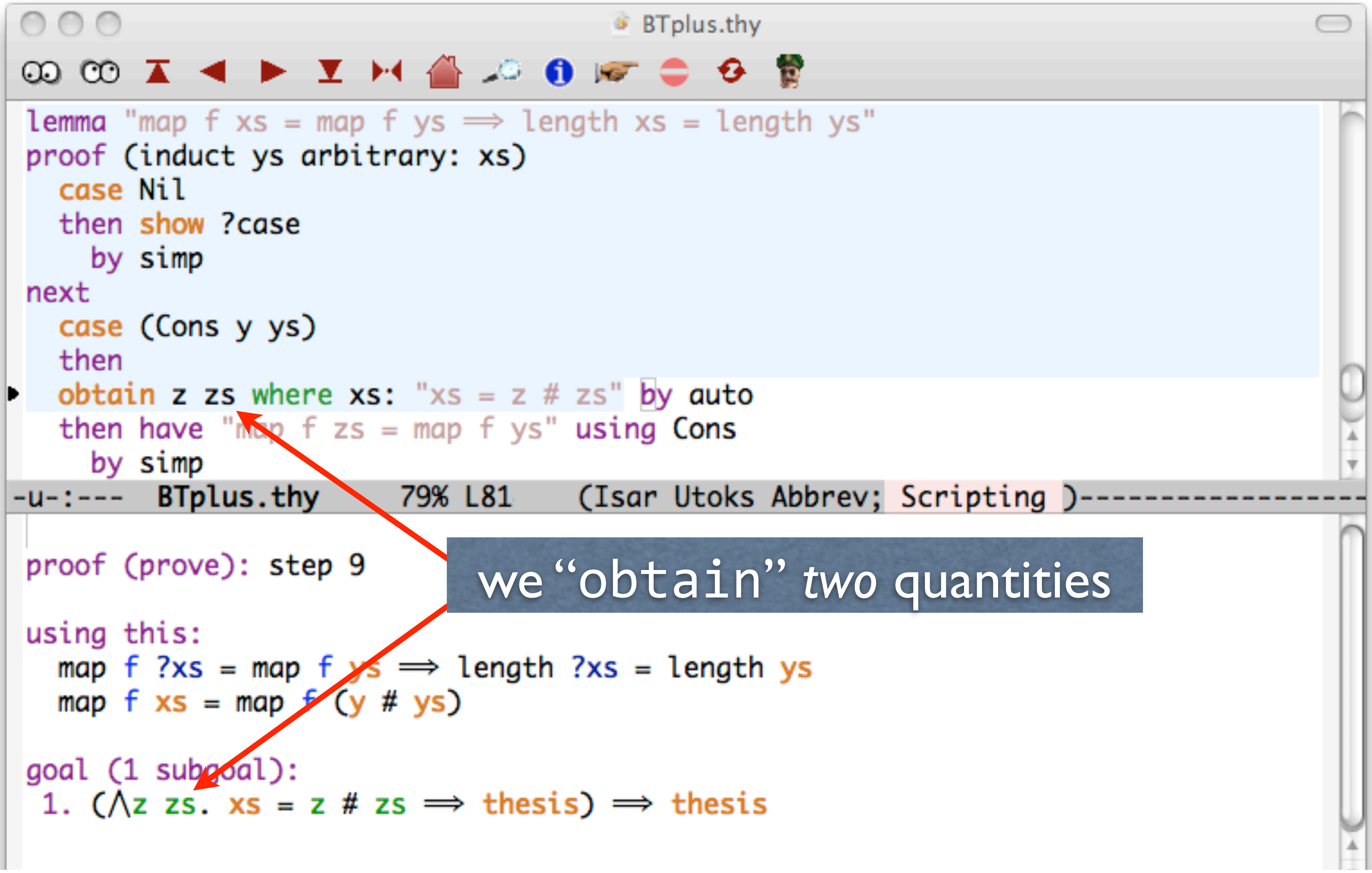
Introducing “then”



Isar proof steps often include facts that are “piped in” (by analogy with UNIX) from previous steps. The use of labels is thereby minimised. Facts so included may be treated specially by the proof method, particularly if the proof method is to apply an elimination rule. The more automatic methods simply add the facts to the subgoal’s assumptions.

The simplest way to include previous facts is by the keyword “then”. Isabelle highlights, as shown above, the fact that have been “picked”.

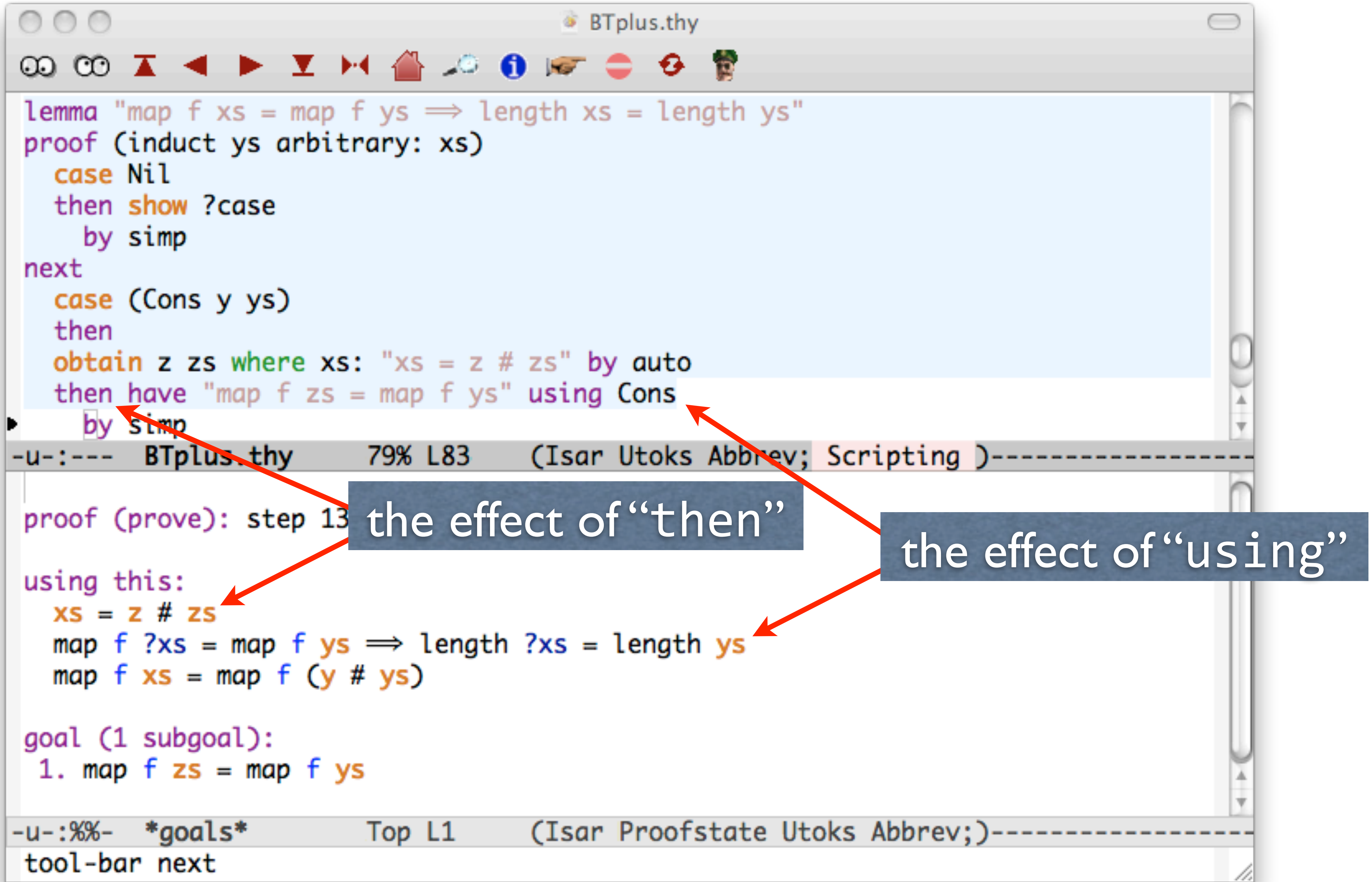
Another Example of “obtain”



$$(\text{map } f\ xs = y\#ys) \leftrightarrow (\exists z\ zs.\ xs = z\#zs\ \&\ f\ z = y\ \&\ \text{map } f\ zs = ys)$$

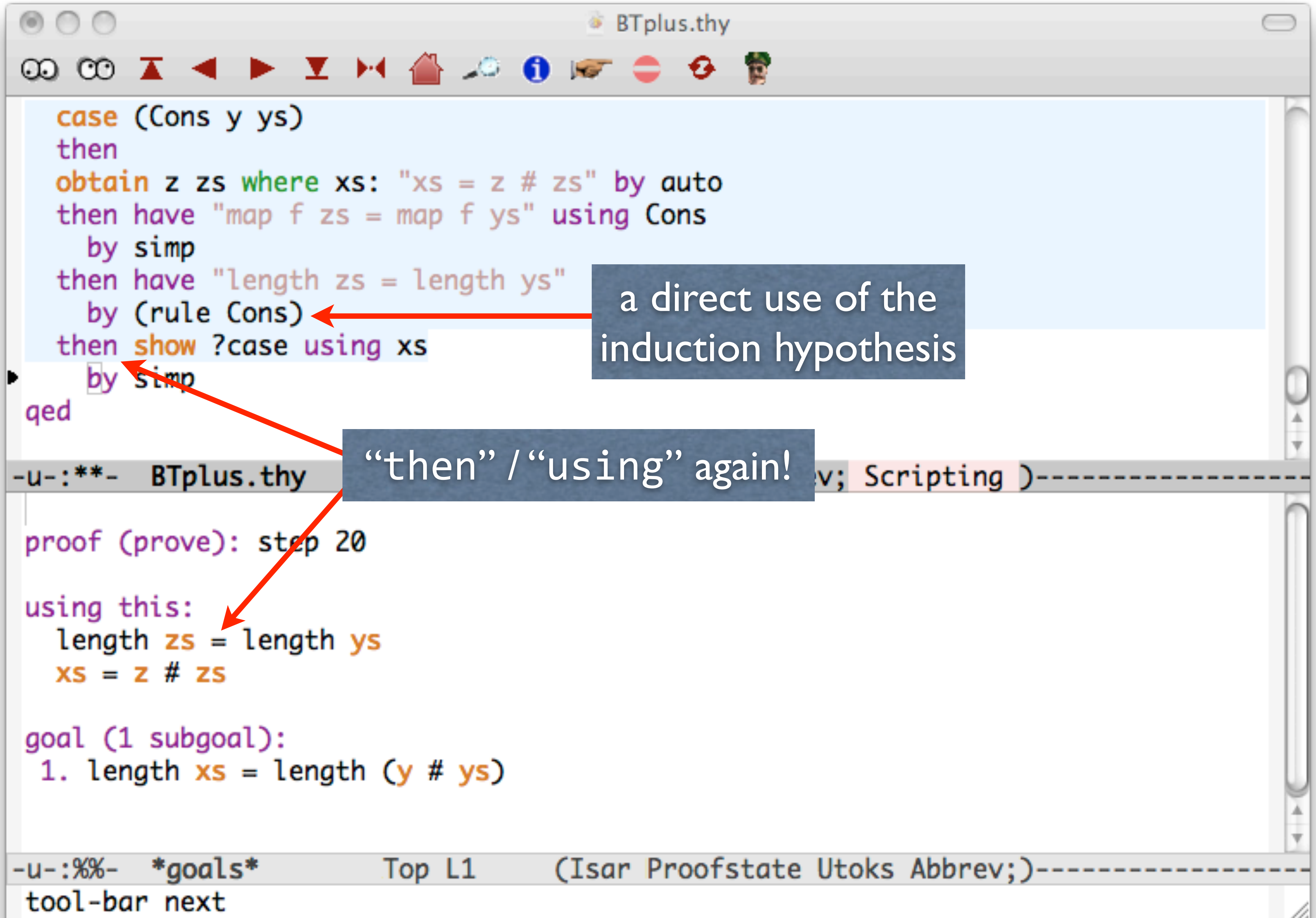
The slightly queer logical equivalence shown above, combined with the assumption `map f xs = map f (y # ys)`, which arises from the induction, implies the existence of `z` and `zs` satisfying a useful equality.

Facts from Two Sources



The ability to introduce facts from multiple sources is both convenient and powerful. It is vital to look at Isabelle's response so that you are aware of what is going on.

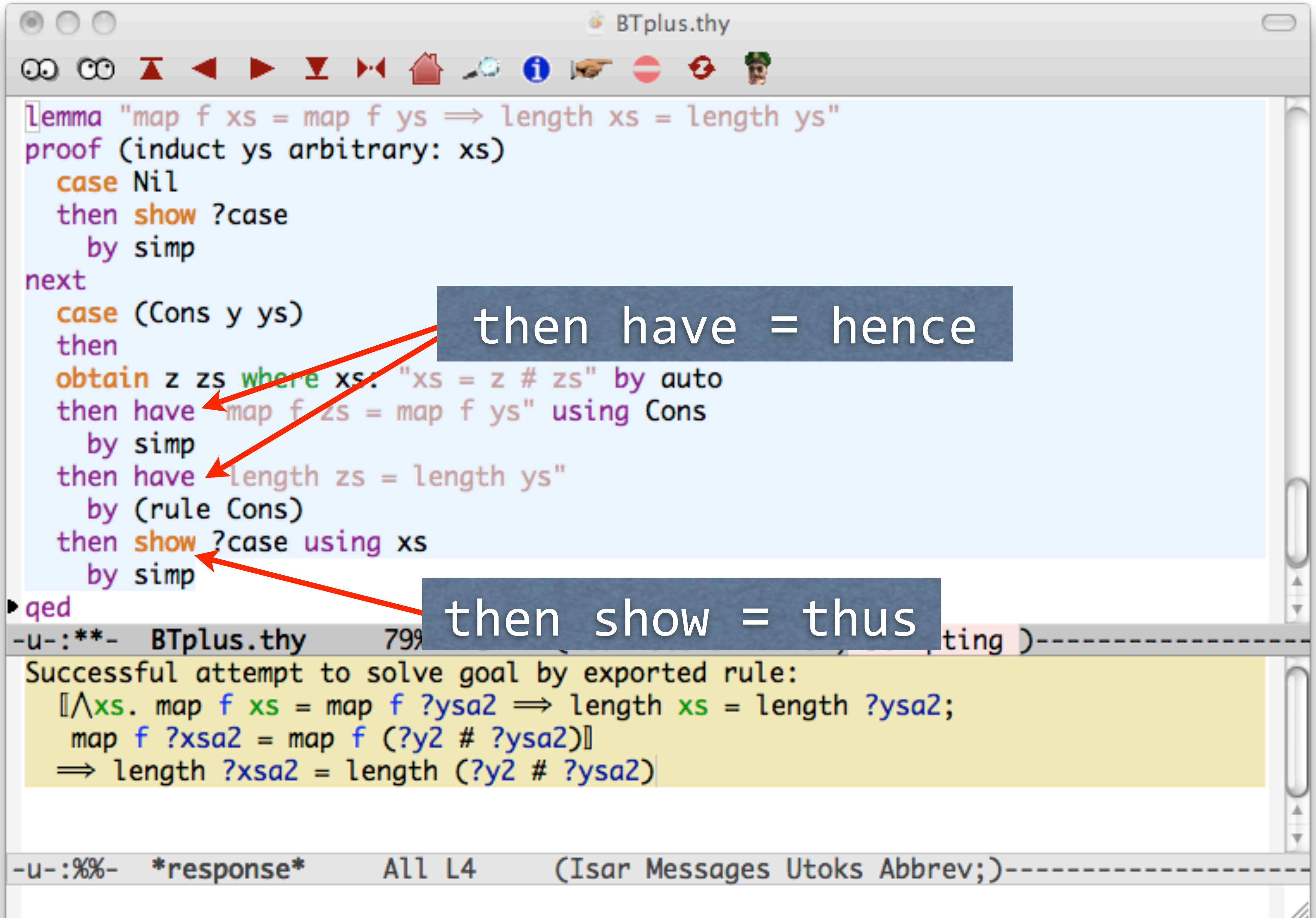
Finishing Up



Unusually, we prove length zs = length ys using the method “`rule`” rather than some automatic method such as “`auto`”. This step needs the induction hypothesis, and we could indeed have included it via “`using Cons`” and then invoked “auto”. But this particular result is simply the conclusion of the induction hypothesis, whose premise was proved in the previous step. Whether to prefer automatic methods or precise steps is a matter of taste, and people argue about which approach is preferable.

Now consider the proof being undertaken at this moment, as shown by Isabelle’s output. The reasoning should be clear: the included facts obviously imply the final goal for this case, written above as “`?case`”.

The Complete Proof



Additional Proof Structures

```
case (insertI A a B)
show "B ∈ Fin"
proof (cases "B ⊆ A")
  case True
  show "B ∈ Fin" using insertI True
    by auto
next
  case False
  have Ba: "B - {a} ⊆ A" using `B ⊆ insert a A`
    by auto
  hence "B = insert a (B - {a})" using False
    by auto
  also have "... ∈ Fin" using insertI Ba
    by blast
  finally show "B ∈ Fin" .
qed
```

```
case (insertI A a B)
show "B ∈ Fin"
proof (cases "B ⊆ A")
  case True
  with insertI show "B ∈ Fin"
    by auto
next
  case False
  have Ba: "B - {a} ⊆ A" using `B ⊆ insert a A`
    by auto
  with False have "B = insert a (B - {a})"
    by auto
  also from insertI Ba have "... ∈ Fin"
    by blast
  finally show "B ∈ Fin" .
qed
```

`from` ⟨*facts*⟩ … = … `using` ⟨*facts*⟩

`with` ⟨*facts*⟩ … = `then from` ⟨*facts*⟩ …

(where … is `have` / `show` / `obtain`)

Full details, probably much more than you want at this stage, can be found in *The Isabelle/Isar Reference Manual* by Makarius Wenzel.

Interactive Formal Verification
11: Modelling Hardware

Lawrence C Paulson
Computer Laboratory
University of Cambridge

Basic Principles of Modelling

- Define *mathematical abstractions* of the objects of interest (systems, hardware, protocols,...).
- Whenever possible, use *definitions* — not axioms!
- Ensure that the abstractions capture enough detail.
  - Unrealistic models have unrealistic properties.
  - Inconsistent models will satisfy *all* properties.

All models involving the real world are *approximate*!

Constructing models using definitions exclusively is called the definitional approach. A purely definitional theory is guaranteed to be consistent. Axioms are occasionally necessary in abstract models, where the behaviour is too complex to be captured by definitions. However, a system of axioms can easily be inconsistent, which means that they imply every theorem. The most famous example of an inconsistent theory is Frege's, which was refuted by Russell's paradox. A surprising number of Frege's constructions survived this catastrophe. Nevertheless, an inconsistent theory is almost worthless.

Useful models are abstract, eliminating unnecessary details in order to focus on the crucial points. The frictionless surfaces and pulleys found in school physics problems are a well-known example of abstraction. Needless to say, the real world is not frictionless and this particular model is useless for understanding everyday physics such as walking. But even models that introduce friction use abstractions, such as the assumption that the force of friction is linear, which cannot account for such phenomena as slipping on ice. Abstraction is always necessary in models of the real world, with its unimaginable complexity; it is often necessary even in a purely mathematical context if the subject material is complicated.

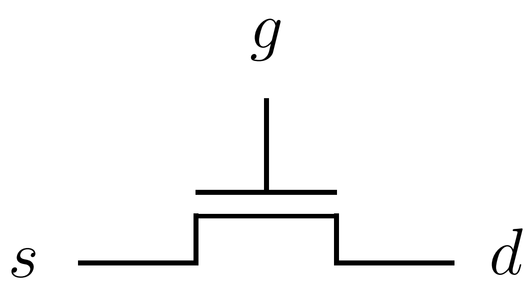
Hardware Verification

- Pioneered by M. J. C. Gordon and his students, using successive versions of the HOL system.
- Used to model substantial hardware designs, including the ARM4 processor.
- Works *hierarchically* from arithmetic units and memories right down to flip-flops and transistors.
- Crucially uses *higher-order logic*, modelling signals as boolean-valued functions over time.

The material in this lecture is based on Prof Gordon's lecture notes for *Specification and Verification II*, which are available on the Internet at http://www.cl.cam.ac.uk/~mjcg/Teaching/SpecVer2/

Devices as Relations

A *relation* in a, b, c, d



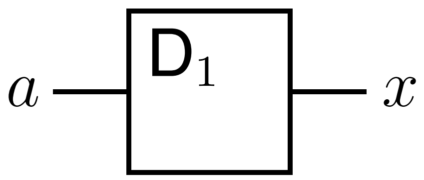
$$g \rightarrow s = d$$

The relation describes the possible combinations of values on the ports.

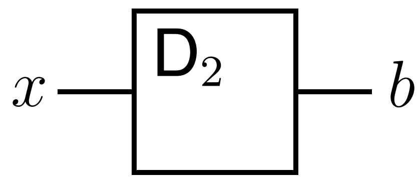
Values could be bits, words, signals (functions from time to bits), etc

The second device on the slide above is an N-type field effect transistor, which can be conceived as a switch: when the gate goes high, the source and drain are connected. The logical implication shown next to the transistor formalises this behaviour. Note that the connection between the source and drain is *bidirectional*, with no suggestion that information flows from one port to the other.

Relational Composition

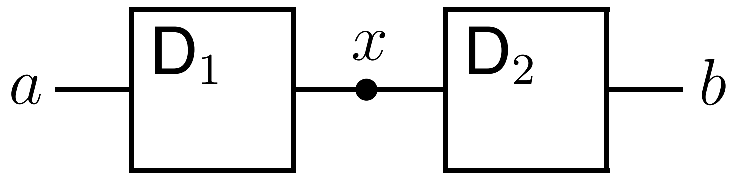


$\mathrm{S}_1[a,x]$



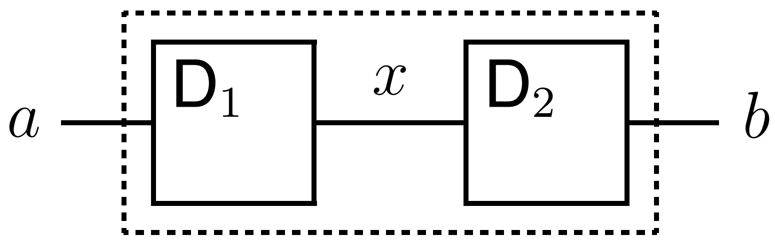
$\mathrm{S}_2[x,b]$

two devices modelled by two formulas



$\mathrm{S}_1[a,x] \wedge \mathrm{S}_2[x,b]$

the connected ports have the *same* value



$\exists x.\, \mathrm{S}_1[a,x] \wedge \mathrm{S}_2[x,b]$

the connected ports have *some* value

The diagrams are taken from Prof Gordon's lecture notes.

Because we model devices by relations, connecting devices together must be modelled by relational composition. Syntactically, we specify circuits by logical terms that denote relations and we express relational composition using the existential quantifier. The quantifier creates a local scope, thereby hiding the internal wire.

Specifications and Correctness

- The *implementation* of a device in terms of other devices can be expressed by composition.
- The *specification* of the device's intended behaviour can be given by an abstract formula.
- Sometimes the implementation and specification can be proved *equivalent*: *Imp*⇔*Spec*.
- The property *Imp*⇒*Spec* ensures that every possible behaviour of the *Imp* is permitted by *Spec*.

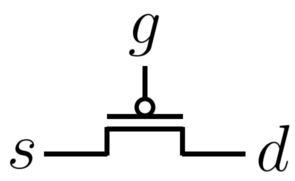
Impossible implementations satisfy *all* specifications!

The implementation describes a circuit, while the specification should be based on mathematical definitions that were established prior to the implementation. A limitation of this approach is that impossible implementations can be expressed: in the most extreme case, implementations that identify the values true and false. In hardware, this represents a short circuit connecting power to ground, possibly a short circuit that only occurs when a particular combination of values appears on other wires, activating an unfortunate series of transistors. In the real world, short circuits have catastrophic effects, while in logic, identifying true with false allows anything to be proved. Therefore, absence of short circuits needs to be established somehow if this relational approach is to be used safely.

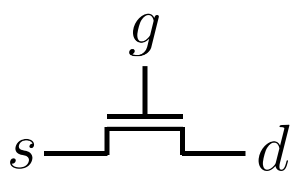
For combinational circuits (those without time), both the implementation and the specification express truth tables with no concept of a "don't care" entry, so logical equivalence should be provable. Sequential circuits involve time, and frequently the specification samples the clock only a specific intervals, ignoring the situation otherwise. Specifications can involve many other forms of abstraction. In general, we cannot expect to prove logical equivalence.

Proving the logical equivalence of the implementation with the specification does not prove the absence of short circuits, but it does prove that the short circuits coincide with inconsistencies in the specification itself. Needless to say, a correct specification should be free of inconsistencies, but there is no way in general to guarantee this. How then do we benefit from using logic? Specifications tend to be much simpler than implementations and they are less likely to contain errors. Moreover, the attempt to prove properties relating specifications and implementations frequently identifies errors, even if we cannot promise all embracing guarantees.

The *Switch Model* of CMOS



$$\mathsf{Ptran}(g, s, d) = (\neg g \Rightarrow (d = s))$$



$$\mathsf{Ntran}(g, s, d) = (g \Rightarrow (d = s))$$

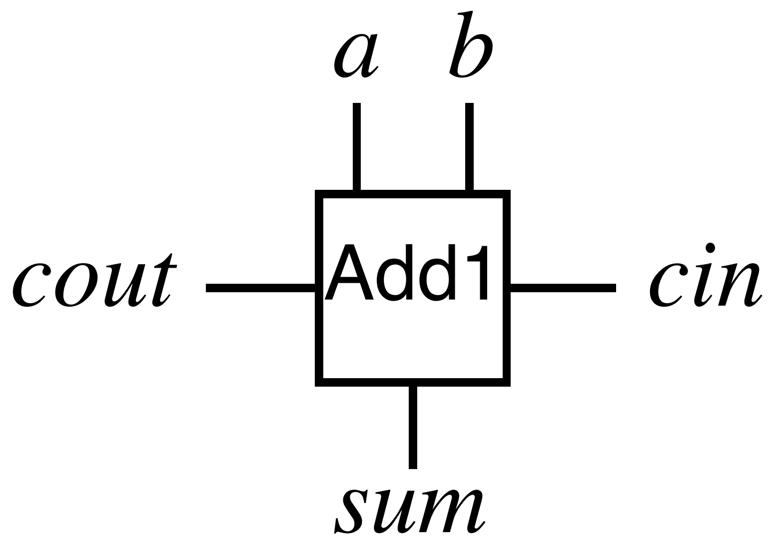
$$\mathsf{Gnd}\ g = (g = \mathsf{F})$$

$$\mathsf{Pwr}\ p = (p = \mathsf{T})$$

```
subsection{* Specification of CMOS primitives *}

text{* P and N transistors *}
definition "Ptran = (λ(g,a,b). (~g ⟶ a = b))"
definition "Ntran = (λ(g,a,b). (g ⟶ a = b))"

text{* Power and Ground*}
definition "Pwr p = (p = True)"
definition "Gnd p = (p = False)"
```

CMOS (complementary metal oxide semiconductor) technology combines P- and N-type transistors on a chip to make gates and other devices. The slide shows primitive concepts: the two types of transistors, ground (modelled by the value False) and power (model by the value True). The corresponding Isabelle definitions are easily expressed. Lambda-notation is a convenient way to express a function is argument is a triple.

Full Adder: Specification

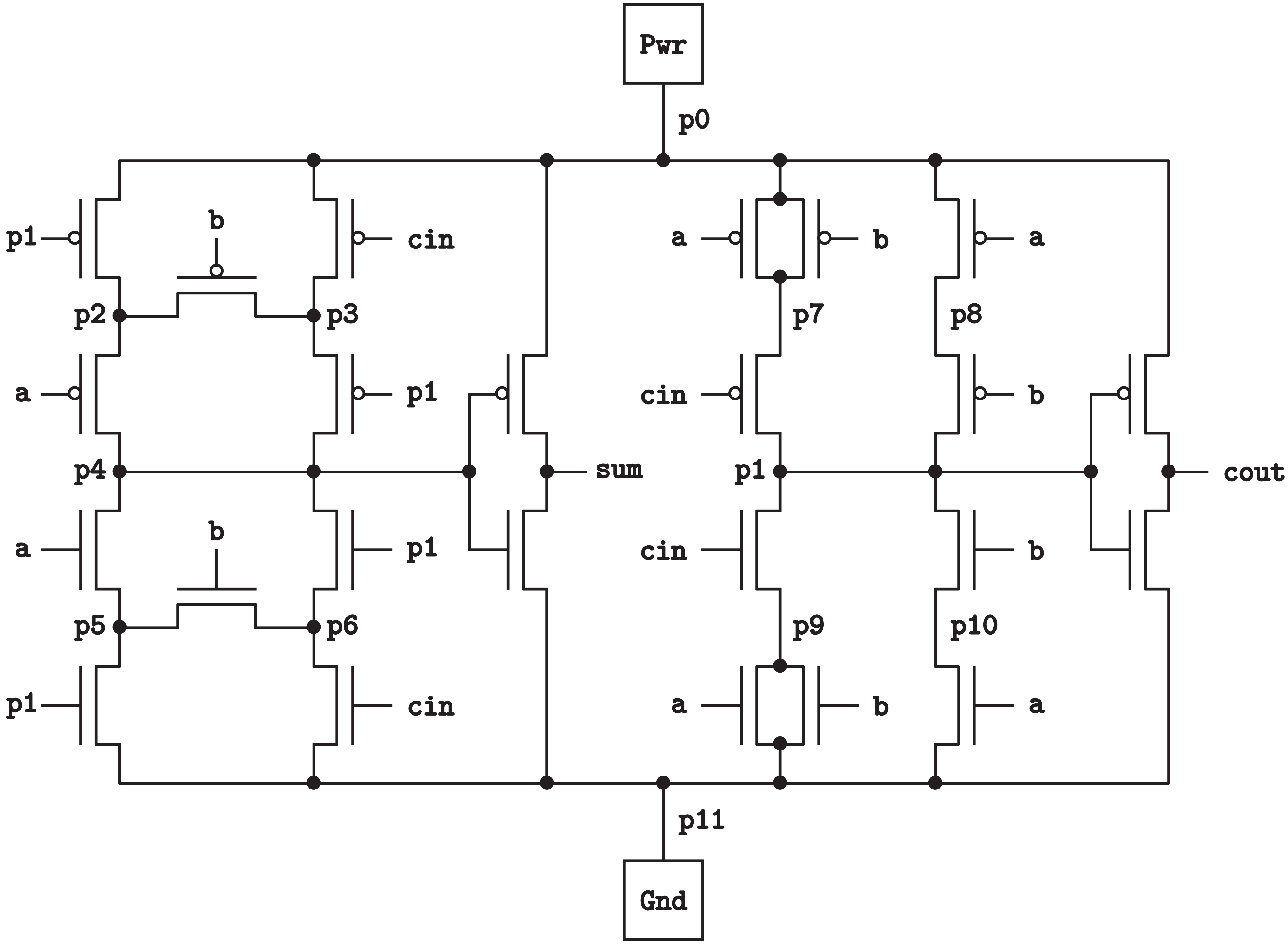


$$2 \times cout + sum = a + b + cin$$

```
text{* 1-bit full adder specification *}

text{* Convert boolean to number (0 or 1) *}
definition bit_val :: "bool ⇒ nat"  where
  "bit_val p = (if p then 1 else 0)"

definition "Add1Spec = (λ(a,b,cin,sum,cout).
              2*(bit_val cout) + bit_val sum =
              bit_val a + bit_val b + bit_val cin)"
```

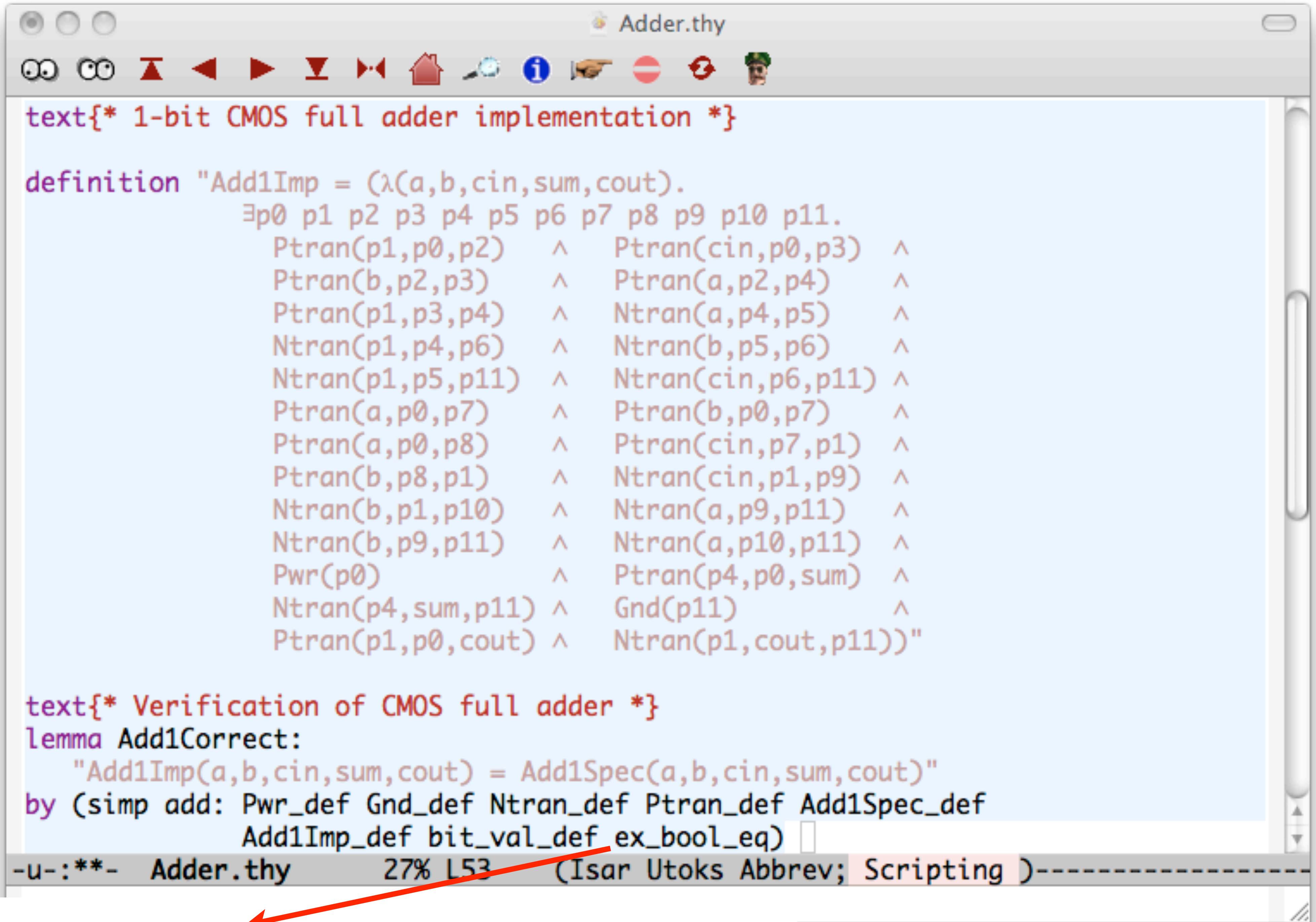
A full adder forms the sum of three one-bit inputs, yielding a two-bit result. The higher-order output bit is called “carry out”, and it will typically be connected to the “carry in” of the next stage. Because we typically use True and False to designate hardware bit values, the obvious conversion to 1 and 0 is necessary in order to express arithmetic properties. Even with this small step, expressing the specification in higher-order logic is trivial. The identifier denotes the abstract relation satisfied by a full adder, namely the legal combinations of values on the various ports.

Full Adder: Implementation



A full adder is easily expressed at the gate level in terms of exclusive-OR (to compute the `sum`) and other simple gating to compute the carry. The diagram above, again from Prof Gordon's notes, expresses a full adder as would be implemented directly in terms of transistors.

Full Adder in Isabelle

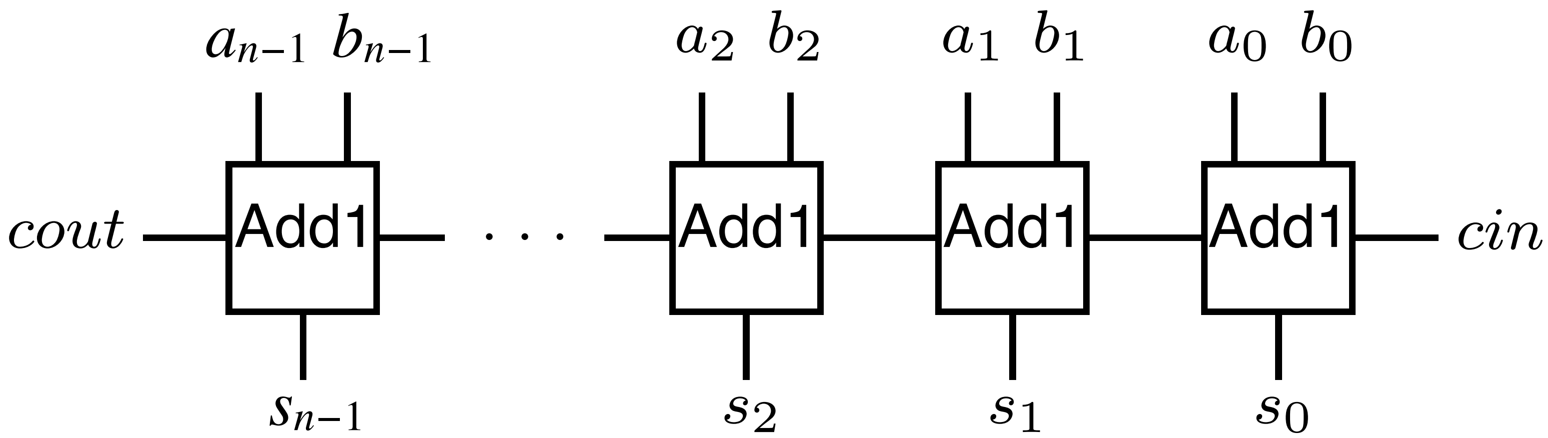


$$(\exists b.\ P\ b) = (P\ \mathrm{True} \lor P\ \mathrm{False})$$

The logical formula above is a direct translation of the diagram on the previous slide. Needless to say, the translation from diagram to formula should ideally be automatic, and better still, driven by the same tools that fabricate the actual chip.

The theorem expresses the logical equivalence between the implementation (in terms of transistors) and the specification (in terms of arithmetic). This type of proof is trivial for reasoning tools based on BDDs or SAT solvers. Isabelle is not ideal for such proofs, and this one requires over four seconds of CPU time. In the simplifier call, the last theorem named is crucial, because it forces a case split on every existentially quantified wire.

An *n*-bit Ripple-Carry Adder



$$(2^n \times cout) + s = a + b + cin$$

- Cascading several full adders yields an *n*-bit adder.
- The implementation is expressed recursively.
- The specification is obvious mathematics.

Adder Specification

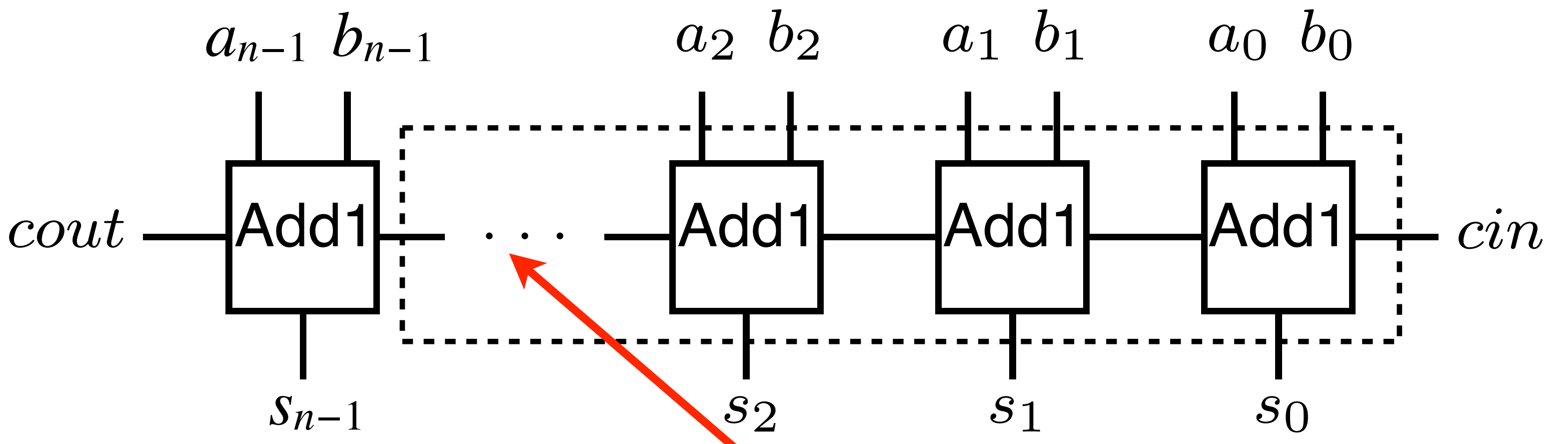
$$(2^n \times cout) + s = a + b + cin$$

values of n-bit words

```
text{* Unsigned number denoted by bitstring f(n-1)...f(0) *}

fun bits_val where
  "bits_val f 0       = 0"
| "bits_val f (Suc n) = 2^n * bit_val(f n) + bits_val f n"

text{* Specification of an n-bit adder *}

definition
  "AdderSpec n = (λ(a, b, cin, sum, cout).
    2^n * bit_val cout + bits_val sum n =
    bits_val a n + bits_val b n + bit_val cin)"
```

The function `bits_val` converts a binary numeral (supplied in the form of a boolean valued function, `f`) to a non-negative integer.The specification of the adder then follows the obvious arithmetic specification closely. When n=0, the specification merely requires cin=cout.

Adder Implementation



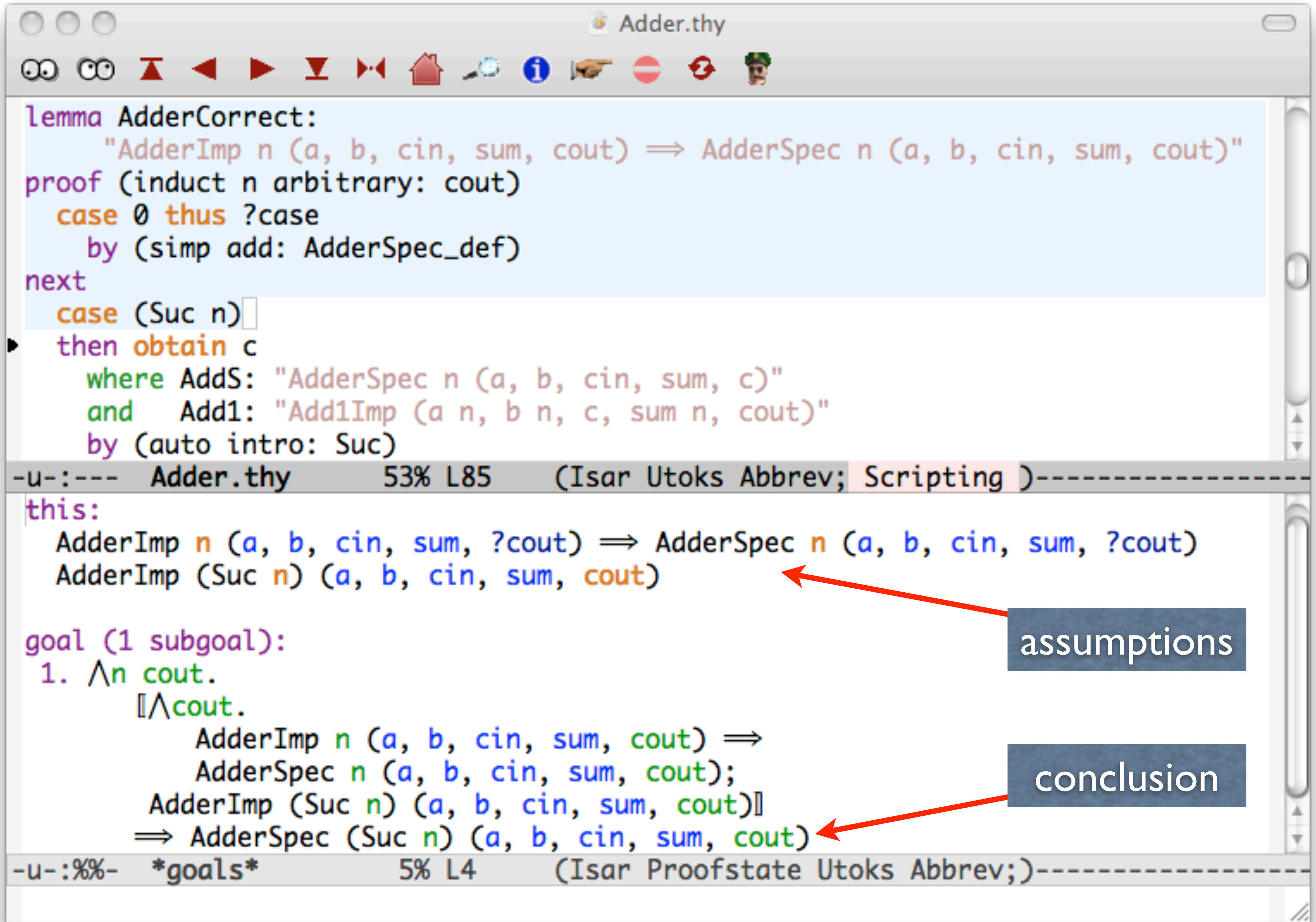
internal wire, to be hidden

```
text{* Implementation of an n-bit ripple-carry adder*}

fun AdderImp where
  "AdderImp 0 (a, b, cin, sum, cout) = (cout = cin)"
| "AdderImp (Suc n) (a, b, cin, sum, cout) =
     (∃c. AdderImp n (a, b, cin, sum, c) ∧
          Add1Imp (a n, b n, c, sum n, cout))"
```

a zero-bit adder simply connects the carry lines!

An (n+1)-bit adder consists of a full adder connected to an n-bit adder. Note that `AdderImp n` specifies an n-bit adder, and in particular, a 0-bit adder is nothing but a wire connecting carry in to carry out.

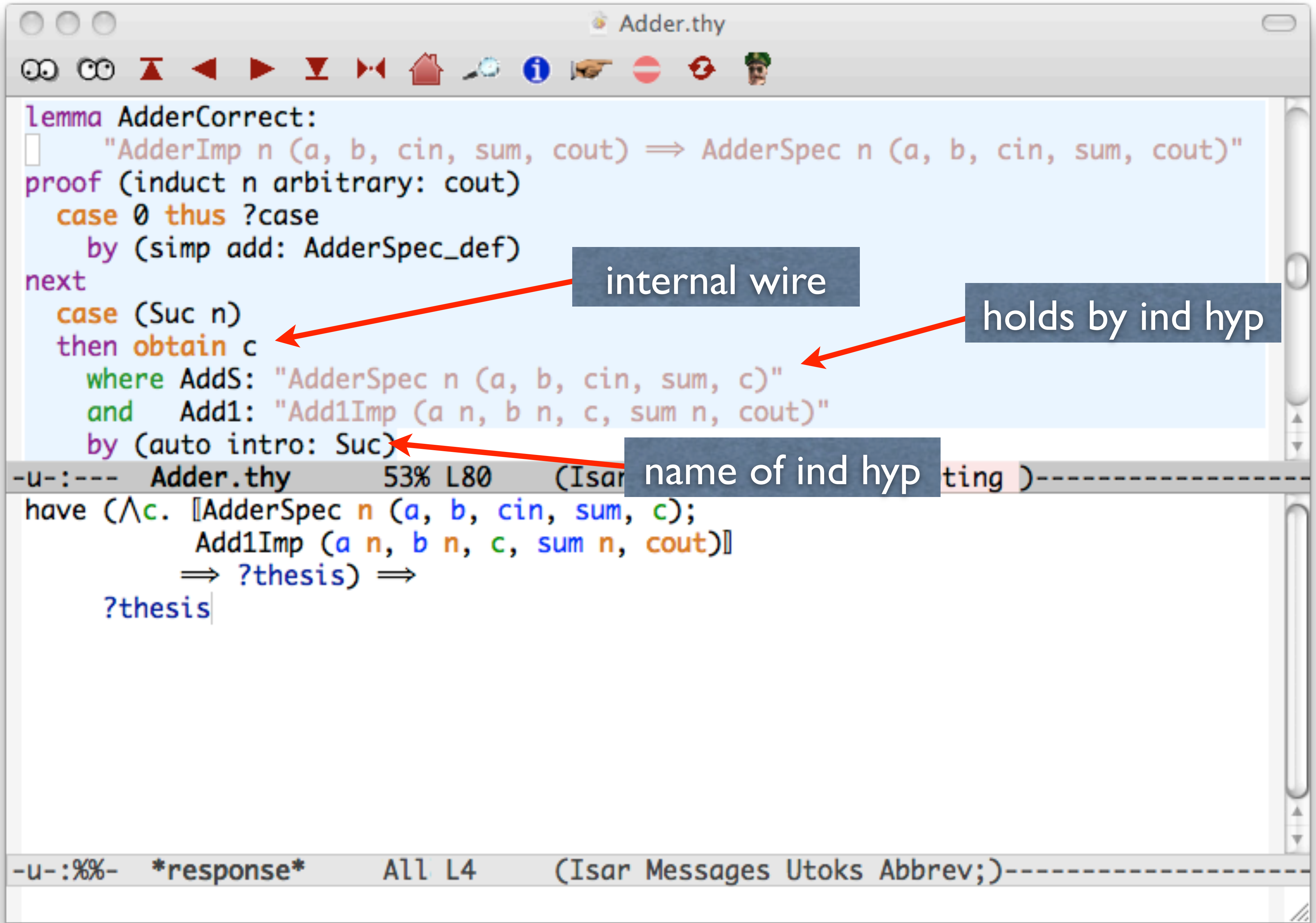
Partial Correctness Proof



We are proving *partial correctness* only: that the implementation implies the specification. The term “partial correctness” here refers to a limitation of the approach, namely that an inconsistent implementation (one with short circuits) can imply any specification. Termination, obviously, plays no role in this circuit.

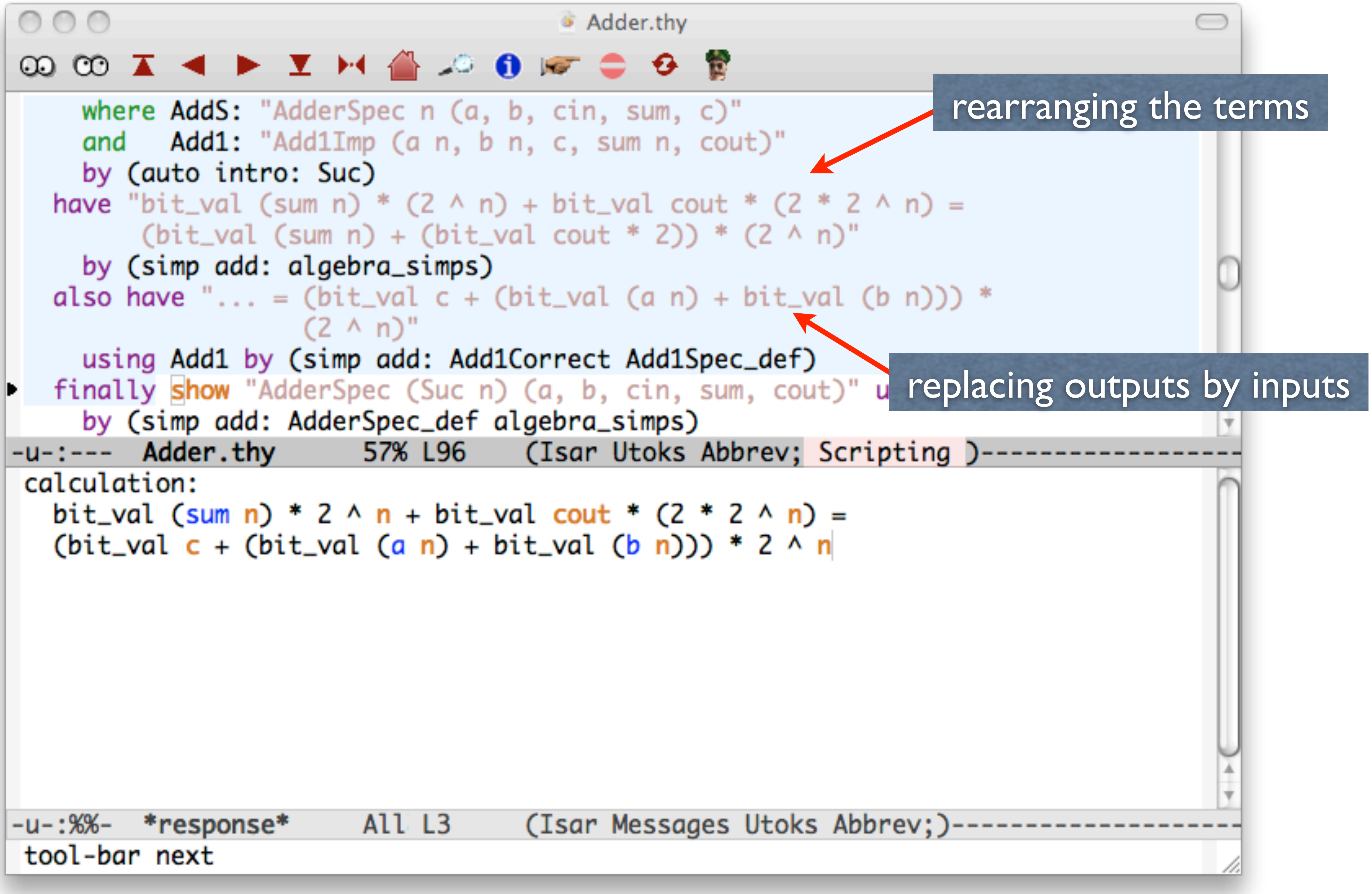
The base case is trivial. Our task in the induction step Is shown on the slide. It is expressed in terms of predicates for the implementation and specification. The induction hypothesis asserts that the implementation implies the specification for n. We now assume the implementation for n+1 and must prove the corresponding specification.

Using the Induction Hypothesis



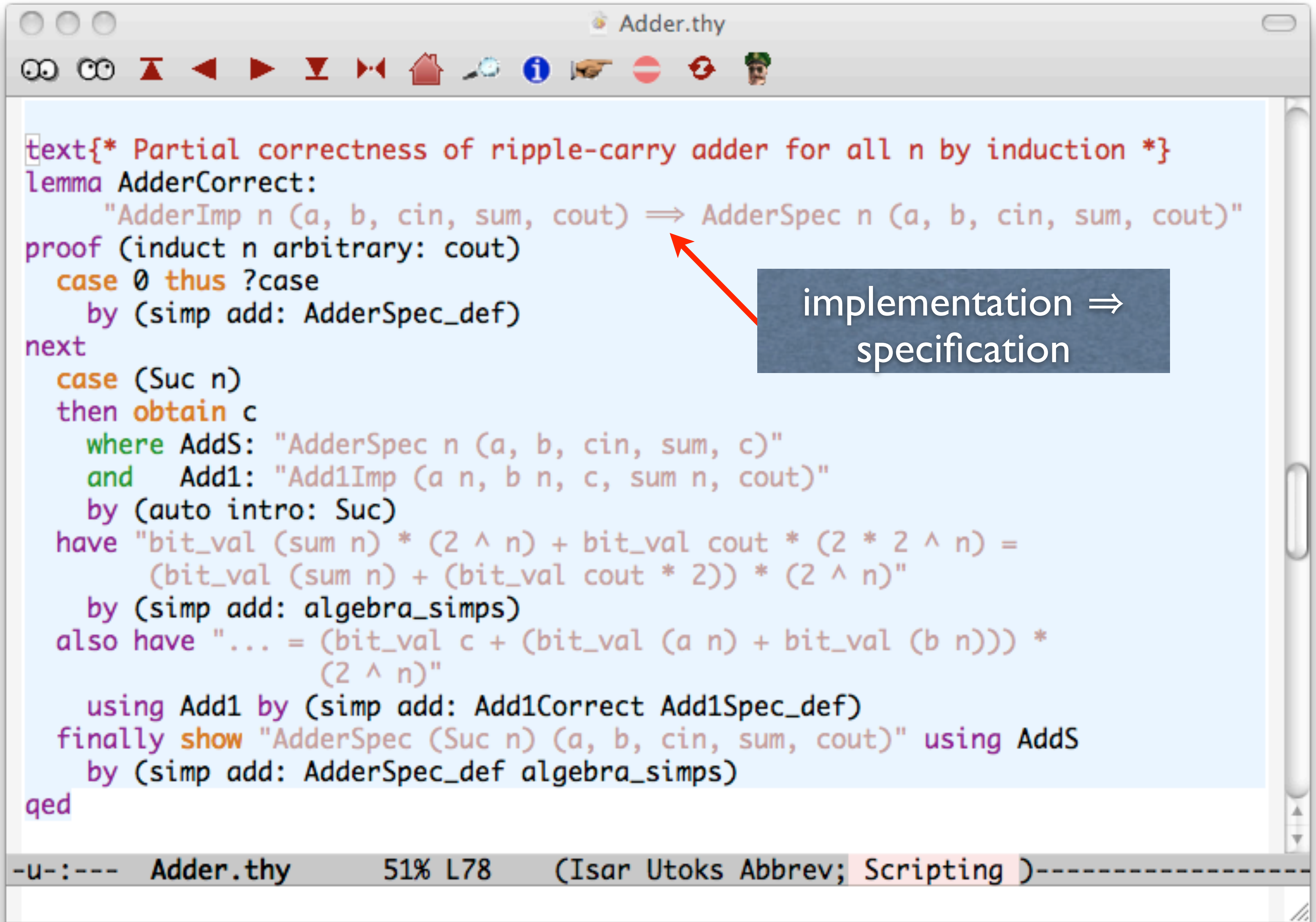
By assumption, we have `AdderImp(Suc n)` and therefore both `AdderImp n` and `Add1Imp`. The simplest use of “`obtain`” would derive those assumptions, but we can skip a step and go directly to `AdderSpec n` by referring to the induction hypothesis.

A Tiresome Calculation



This equation is suggested by earlier attempts to prove the induction step directly. The proof involves using the correctness of a full adder to replace `Add1Imp` by `Add1Spec`, then unfolding the latter to get the sum c + a n + b n. The precise form of the left-hand side has been chosen to match a term that will appear in the main proof. This kind of reasoning is tedious even with the help of Isar. Better support for arithmetic could make this proof almost automatic.

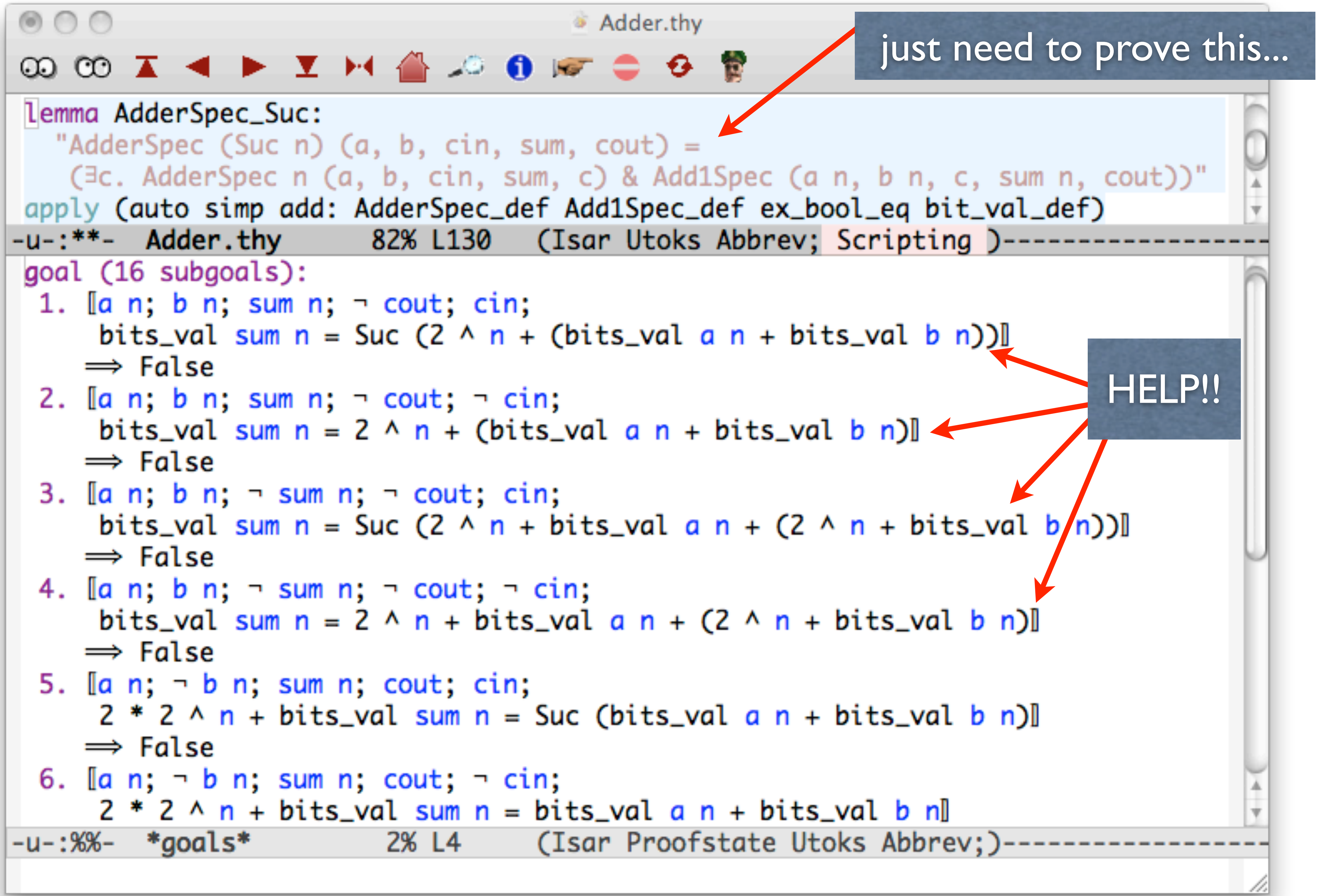
The Finished Proof



We end up with a fairly simple structure. Note that we could have used it `Add1Correct` earlier in the proof, obtaining Add1: "`Add1Spec` ..." directly.

To repeat: we have proved that every possible configuration involving the connectors to our circuit satisfies the specification of an n-bit adder. Tools based on BDDs or SAT solvers can prove instances of this result for fixed values of n, but not in the general case.

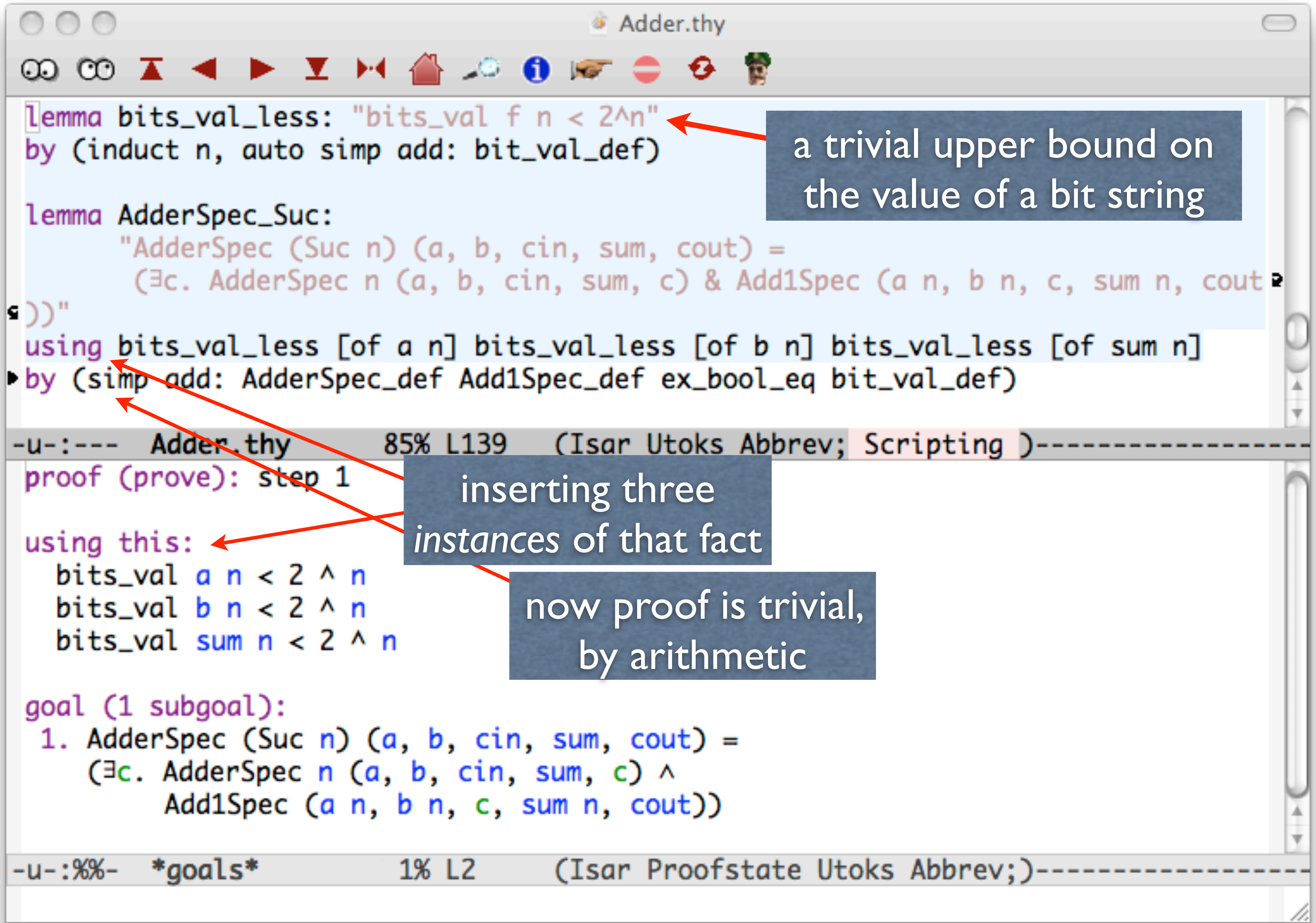
Proving Equivalence



To prove that the specification implies the implementation would yield their exact equivalence. It would also guarantee the lack of short circuits in the implementation, as the specification is obviously correct.

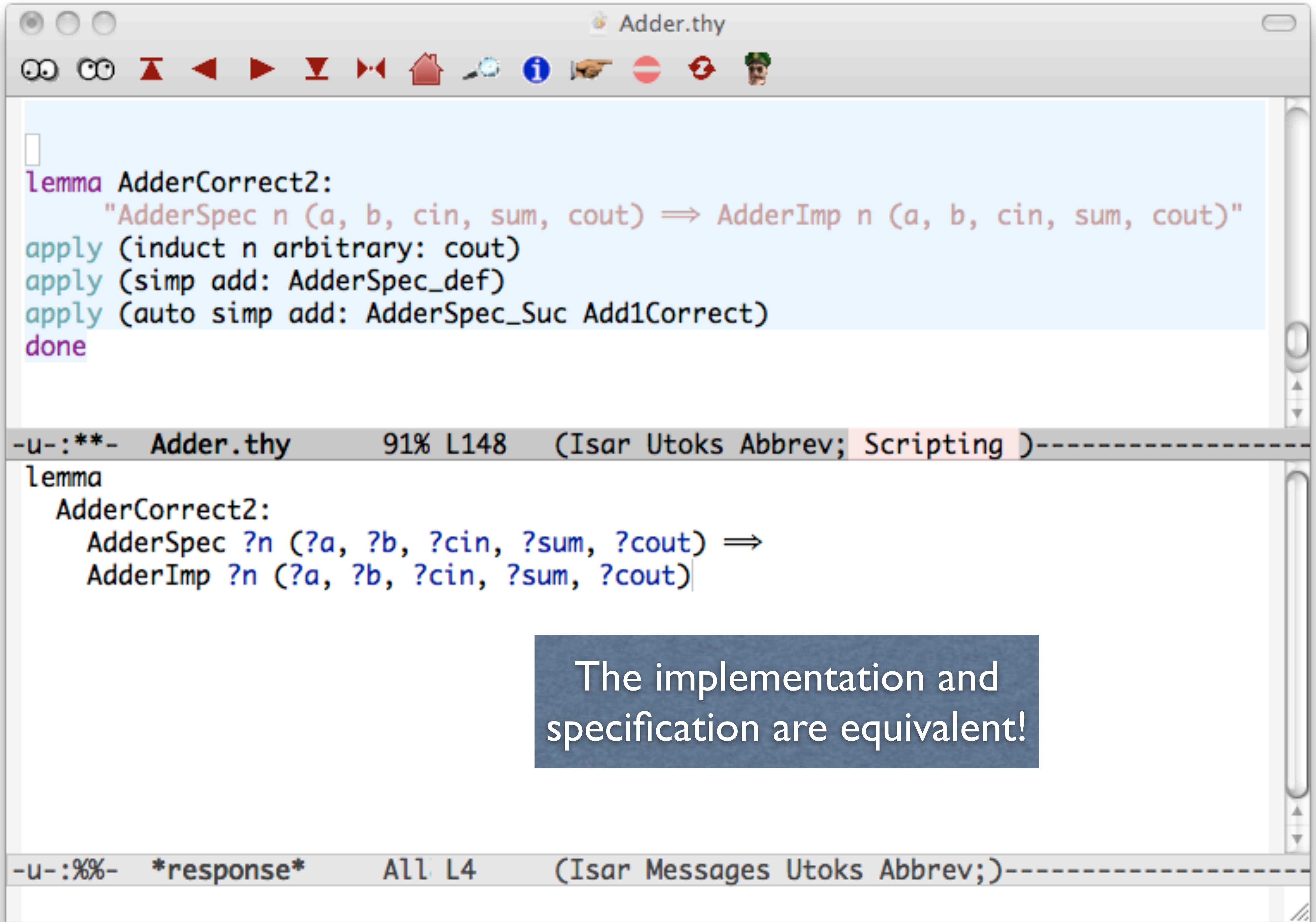
The verification requires the lemma shown above, which resembles the recursive case of `AdderImp`. We might expect its proof to be straightforward. Unfortunately, the obvious proof attempt leaves us with 16 subgoals. A bit of thought informs us that these cases represent impossible combinations of bits. These arithmetic equations cannot hold. But how can we prove this theorem with reasonable effort?

A Crucial Lemma



The crucial insight is that all of the impossible cases involve bit strings that have impossibly high values. It is trivial to prove the obvious upper bound on an n-bit string. Less obvious is that Isabelle's arithmetic decision procedures can dispose of the impossible cases with the help of that upper bound. We use a couple of tricks. One is that "`using`" can be inserted before the "`apply`" command, where it makes the given theorems available. The other trick is the keyword "`of`", which is described below.

The Opposite Implication



With the help of `AdderSpec_Suc`, the opposite direction of the logical equivalence is a trivial induction.

Making Instances of Theorems

- *thm* `[of` $a\ \ b\ \ c$`]`
  replaces variables by terms from left to right
- *thm* `[where` x=a`]`
  replaces the variable x by the term a
- *thm* `[OF` $thm_1\ \ thm_2\ \ thm_3$`]`
  discharges premises from left to right
- *thm* `[simplified]`
  applies the simplifier to *thm*
- *thm* `[`$attr_1$`,` $attr_2$`,` $attr_3$`]`
  applying multiple attributes

We proved `AdderSpec_Suc` with the help of "`using`", which inserted a crucial lemma into the proof. We needed specific instances of the lemma because Isabelle's arithmetic decision procedures cannot make use of the general formula. Fortunately, we needed only three instances and could express them using the keyword "`of`". This type of keyword is called an *attribute*. Attributes modify theorems and sometimes declare them: we have already seen attributes like `[simp]` and `[intro]` many times.

The most useful attributes are shown on the slide. Replacing variables in a theorem by terms (which must be enclosed in quotation marks unless they are atomic) can also be done using "`where`", which replaces a named variable. in the left to right list of terms or theorems, use an underscore (_) to leave the corresponding item unspecified. An example is `bits_val_less [of _ n]`, which denotes `bits_val ?f n < 2 ^ n`.

Joining theorems conclusion to premise can be done in two different ways. An alternative to `OF` is `THEN`: thm_1 [`THEN` thm_2] joins the conclusion of thm1 to the premise of thm2. Thus it is equivalent to thm_2 `[THEN` thm_1`]`. The result of such combinations can often be `simplified`. Finally, we often want to apply several attributes one after another to a theorem.

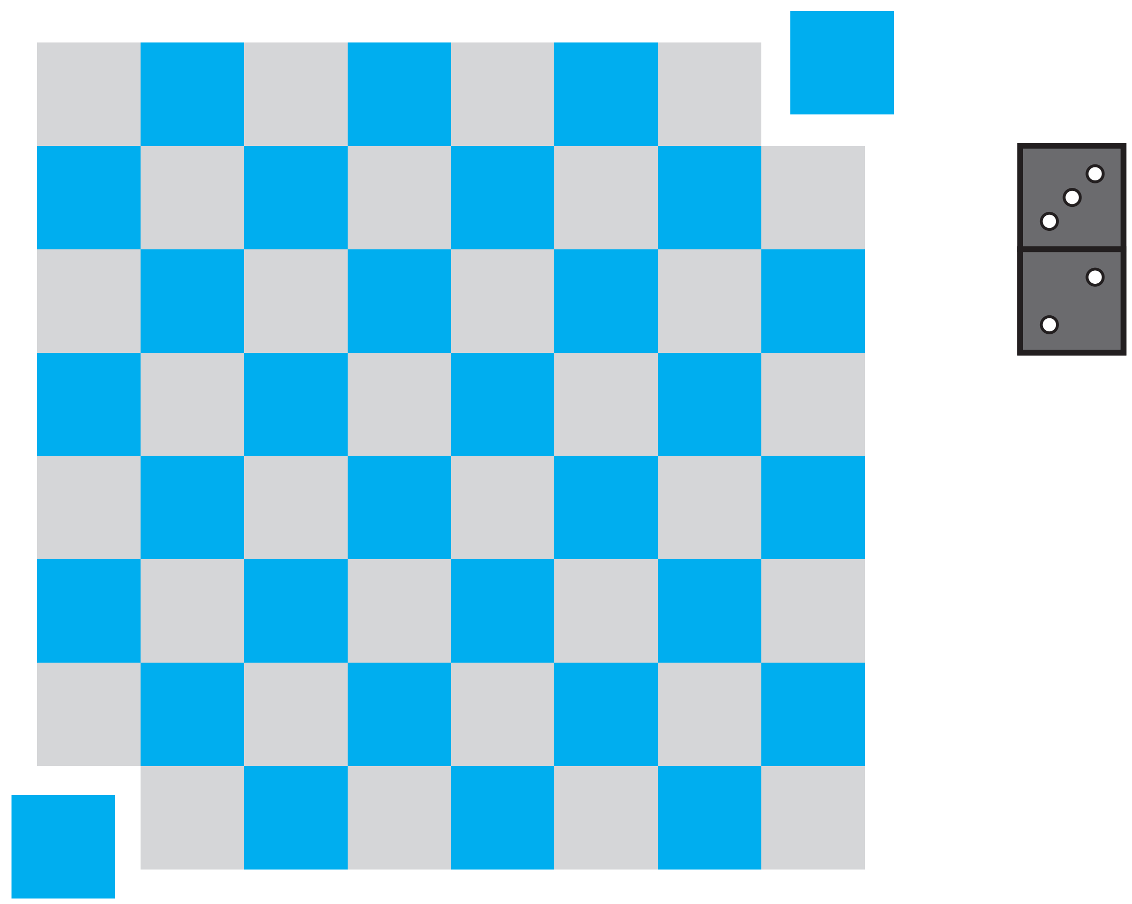
See the *Tutorial*, section **5.15 Forward Proof: Transforming Theorems**.

Interactive Formal Verification
12: The Mutilated Chess Board

Lawrence C Paulson
Computer Laboratory
University of Cambridge

The Mutilated Chess Board

Can this damaged board be tiled using dominoes?



A clear proof requires an *abstract* model.

An earlier version of this formalisation is described in the paper referenced below. Comparing that version of the proof with the present one gives an indication of the progress made by Isabelle developers, especially as regards structured proof.

L. C. Paulson.
A simple formalization and proof for the mutilated chess board.
Logic J. of the IGPL **9** 3 (2001), 499–509.

`http://jigpal.oxfordjournals.org/cgi/reprint/9/3/475`

Proof Outline

- Every *row* of length $2n$ can be tiled with dominoes.
- Every *board* of size $m \times 2n$ can be tiled.
- Every tiled area has the same number of black and white squares.
- Removing some white squares from a tiled area leaves an area that cannot be tiled.
- No mutilated $2m \times 2n$ board can be tiled.

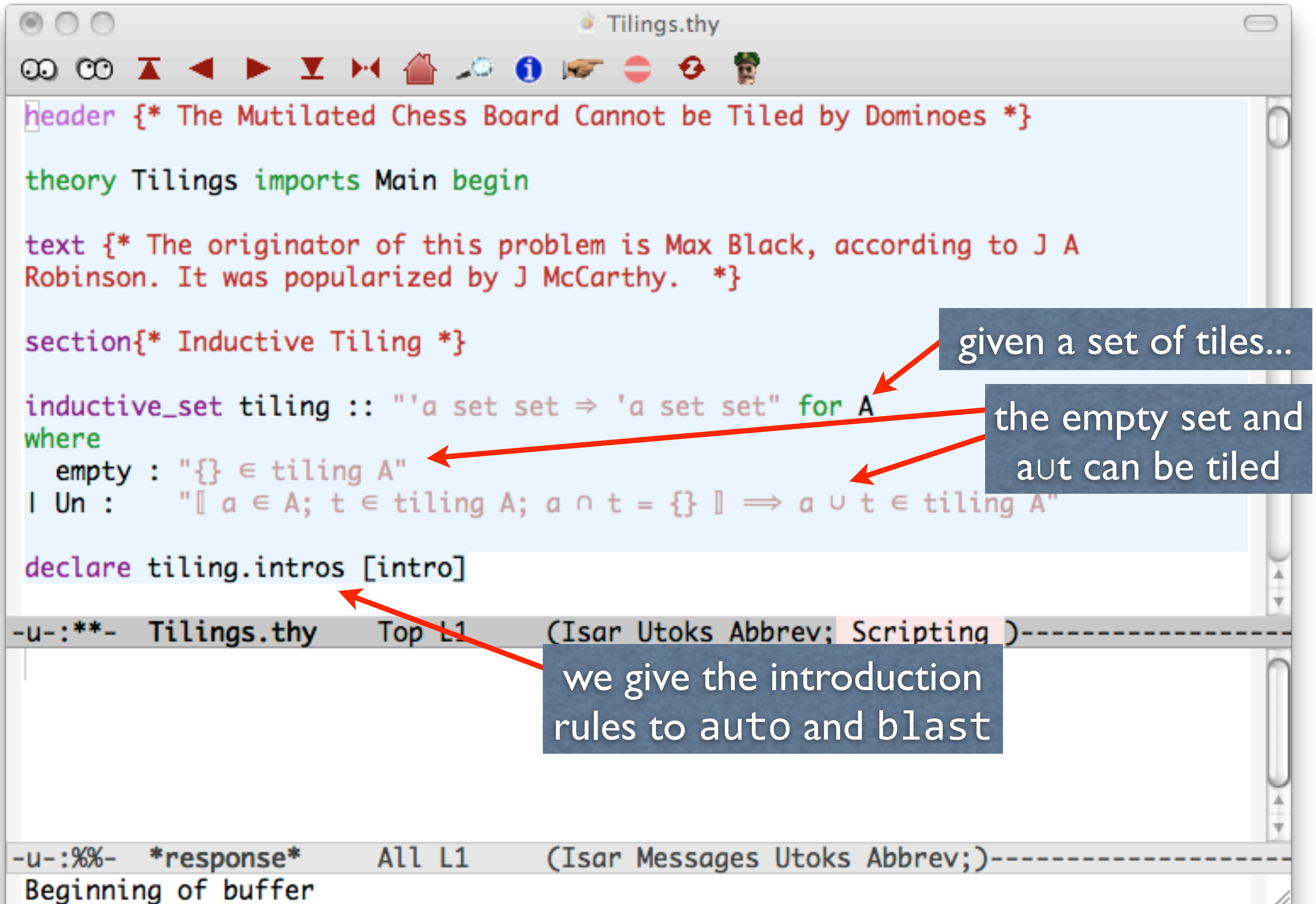
The diagram is compelling with no reasoning at all. By comparison, even the five steps shown above are more complicated than we would like. However, the Isabelle formalisation is simpler and shorter than the others that I am aware of.

An Abstract Notion of Tiling

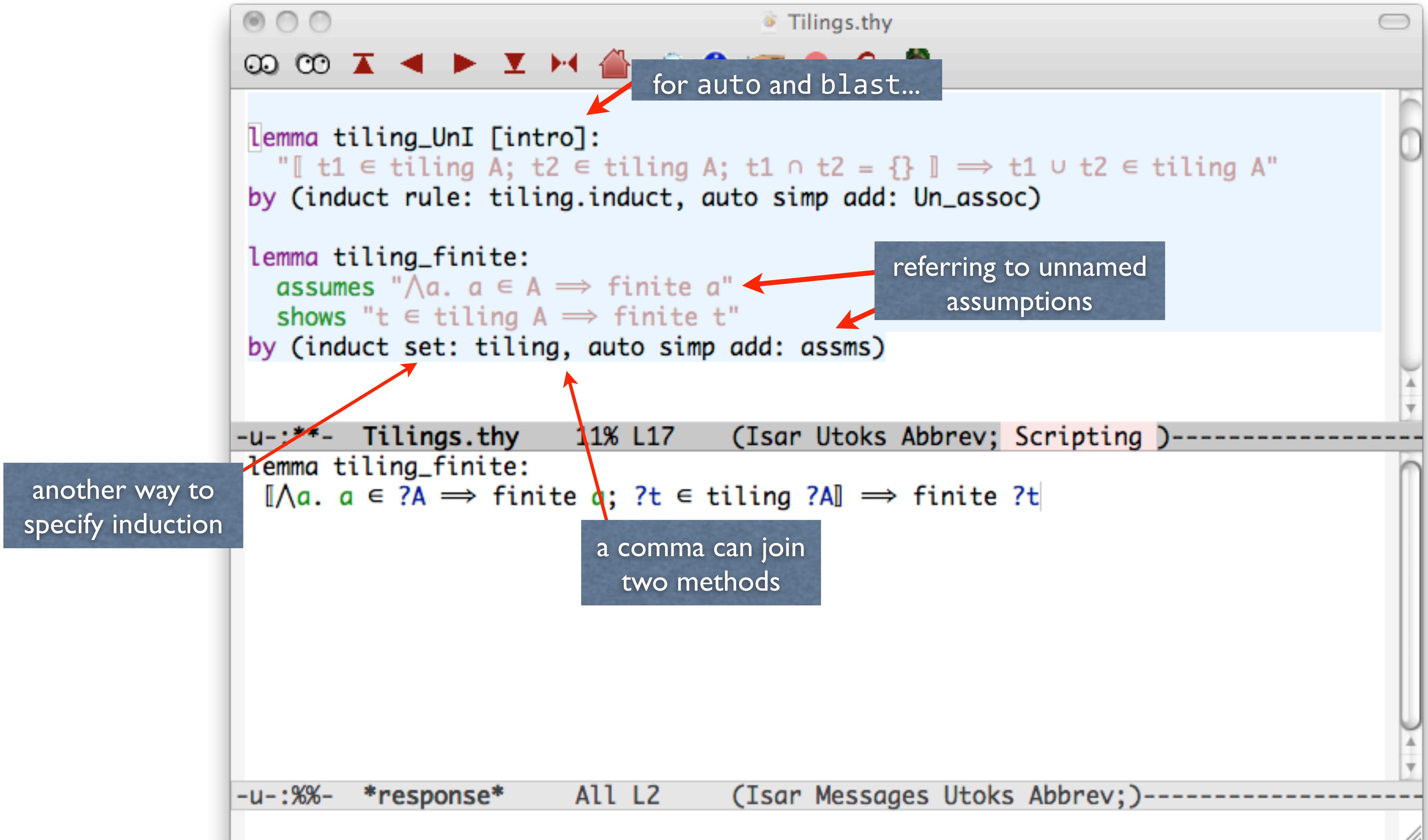
- A *tile* is a set of *points* (such as squares).
- Given a set of tiles (such as dominoes),
  - the empty set can be tiled,
  - and so can $a \cup t$ provided
    - t can be tiled, and
    - a is a tile disjoint from t (no overlaps!)

Instead of formalising chess boards concretely, we look more abstractly at the question of covering a set by non-overlapping tiles.

Tilings Defined Inductively



Simple Proofs about Tilings



Two disjoint tilings can be combined by taking their union, yielding another tiling. The induction is trivial, using the associativity of union. Section 4 of the paper “A simple formalization and proof for the mutilated chess board” explains the proof in more detail.

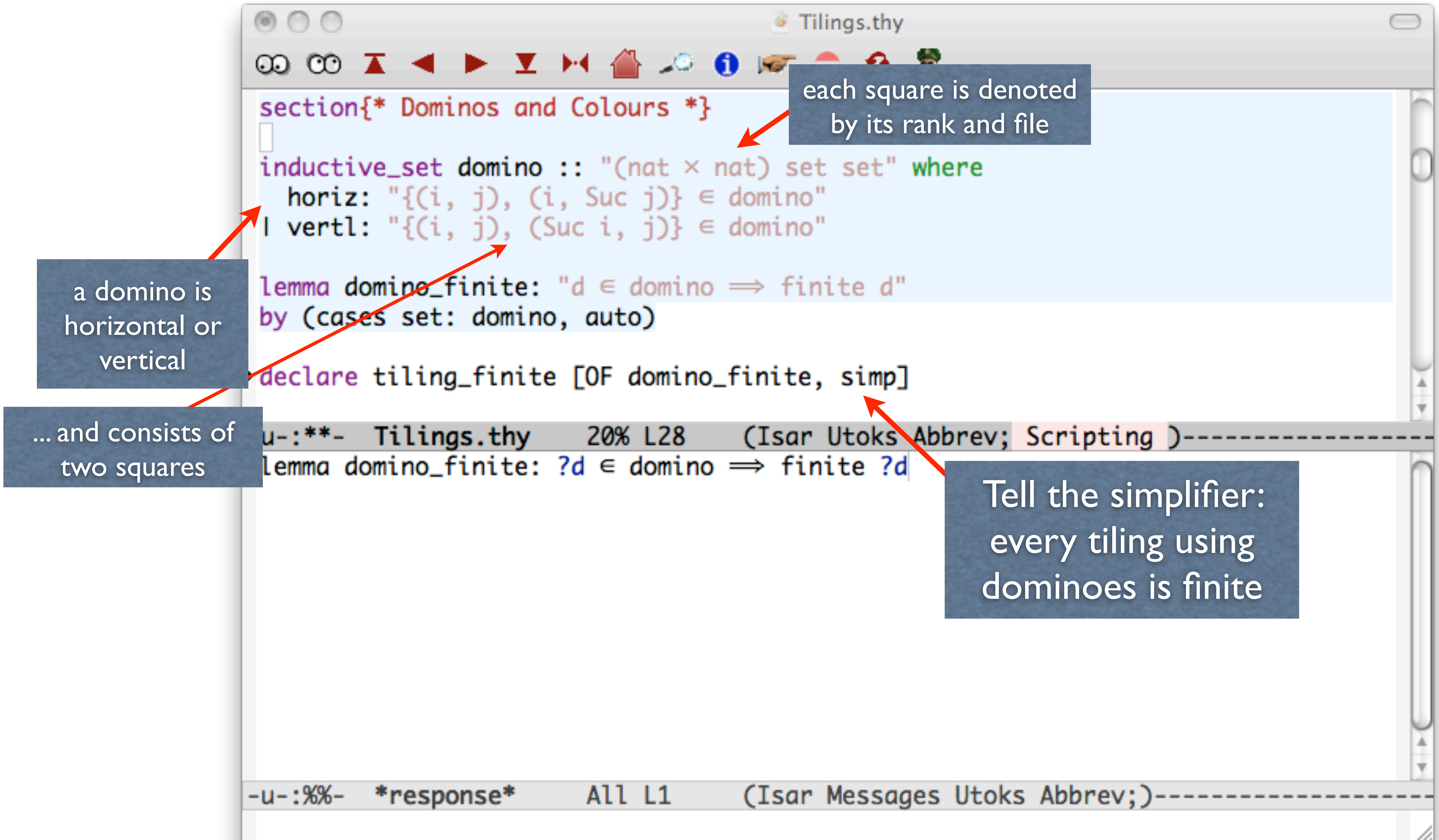
If each of our tiles is a finite set, then all the tilings we can create are also finite. The induction is again trivial. Even if we have infinitely many tiles, a tiling can only use finitely many of them.

We see something new here: the identifier `assms`. It provides a uniform way of referring to the assumptions of the theorem we are trying to prove, if we have neglected to equip those assumptions with names.

Another novelty is the method `induct set: tiling`, which specifies induction over the named set without requiring us to name the actual induction rule.

Yet another novelty: we can join a series of methods using commas, creating a compound method that executes its constituent methods from left to right. Lengthy chains of methods would be difficult to maintain, but joining two or three as shown is convenient. Now the proof can be expressed using “by”, because it is accomplished by a single (albeit compound) method.

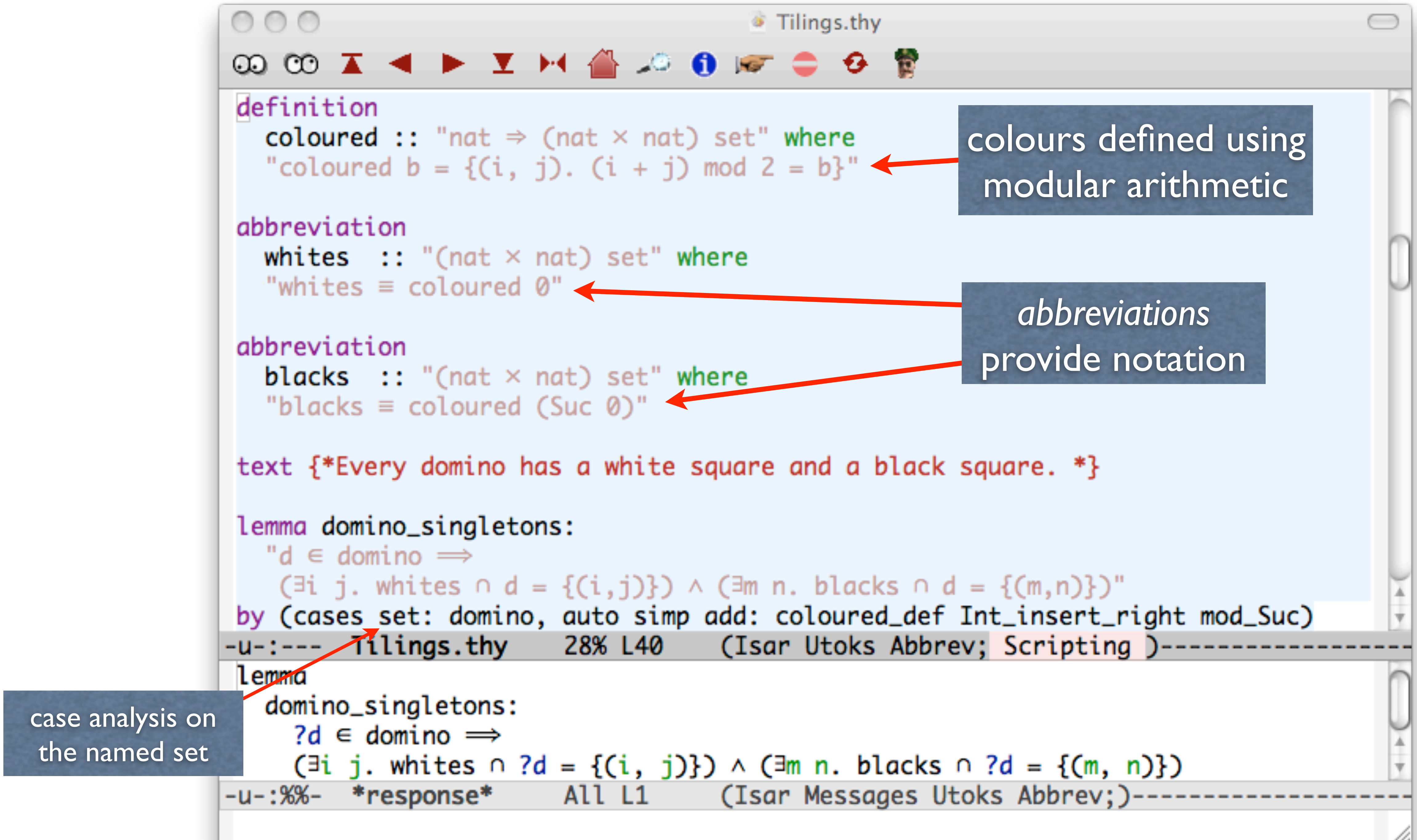
Dominoes for Chess Boards



The formalisation of dominoes is extremely simple: each domino is a two element set of the form $\{(i,j), (i,j+1)\}$ or $\{(i,j), (i+1,j)\}$, expressing a horizontal or vertical orientation. The set of dominoes is not actually inductive and we could have defined it by a formula, but the inductive set mechanism is still convenient.

Because each domino contains two elements, dominoes are trivially finite. The declaration shown above combines two finiteness properties, asserting that tilings that consist of dominoes are finite, and it gives this fact to the simplifier. Concluding a series of attributes by `simp` or `intro` is common.

White and Black Squares

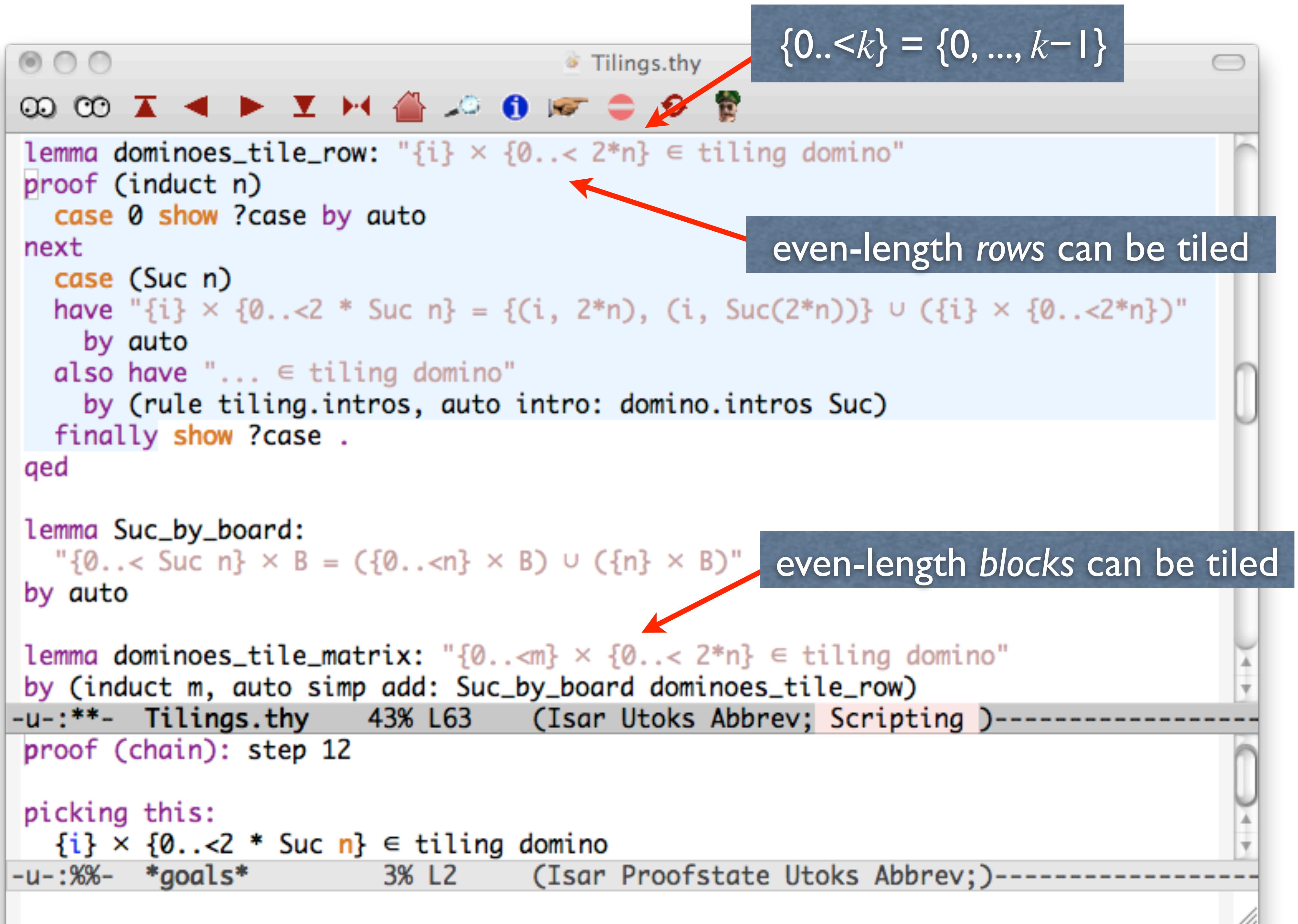


The distinction between white and black is made using modulo-2 arithmetic. The constants “whites” and “blacks” do not have definitions in the normal sense; they are declared as abbreviations, which means that these constants never occur in terms. They provide a shorthand for expressing the terms “coloured 0” and “coloured (Suc 0)”. Recall that to define a constant in Isabelle introduces an equation that can be used to replace the constant by the defining term. And this equation is not even available to the simplifier by default. With abbreviations, no such equations exist.

See the *Tutorial*, section **4.1.4 Abbreviations**, for more information. More generally, section 4.1 describes concrete syntax and infix annotations for Isabelle constants.

It is now trivial to prove that every domino has a white square and a black square, by case analysis on the two kinds of domino. The proof requires giving the simplifier some facts about intersection and the modulus function.

Rows and Columns



The first theorem states that any row of even length can be tiled by dominoes. In the inductive step, observe how the expression `{0..<2 * Suc n}` is rewritten to involve an explicit domino, `{(i, 2*n), (i, Suc(2*n))}`. Structured proofs make this sort of transformation easy, provided we are willing to write the desired term explicitly.

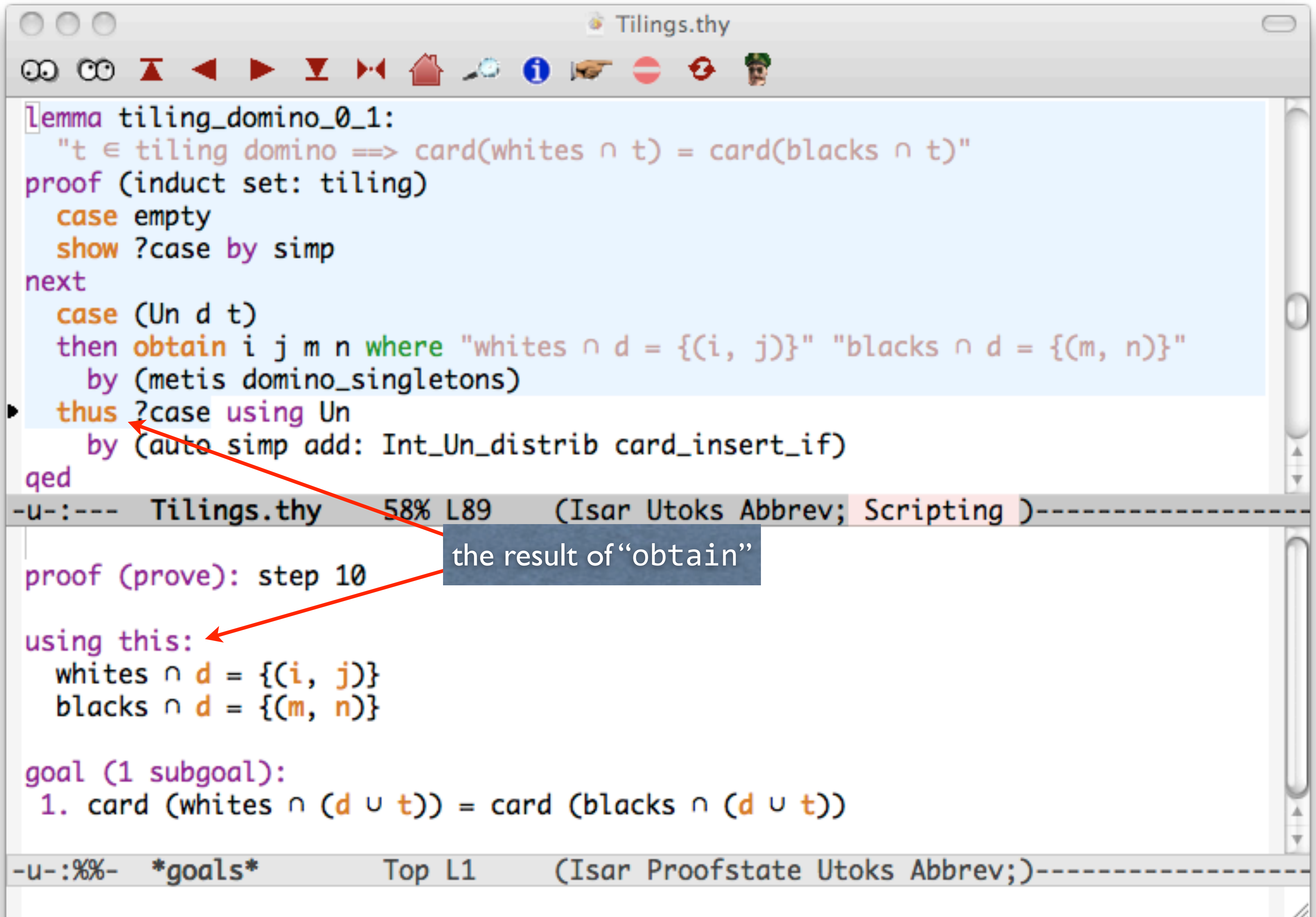
The alternative approach, of choosing rewrite rules that transform a term precisely as we wish, eliminates the need to write the intermediate stages of the transformation, but it can be more time-consuming overall. You know this other approach has been adopted if you see this sort of command:

```
apply (simp add: mult_assoc [symmetric] del: fact_Suc)
```

The theorem `mult_assoc` is given a reverse orientation using the attribute `[symmetric]`, while the theorem `fact_Suc` is removed from this simplifier call.

The induction at the bottom of this slide is an example of the alternative approach done correctly. We first prove a lemma to rewrite the induction step precisely as we wish: in other words, so that it will create an instance of `dominoes_tile_row`. The lemma is easily proved and the inductive proof is also easy.

For Tilings, #Whites = #Blacks



The crux of the argument is that any area tiled by dominoes must contain the same number of white and black squares. This statement is easily expressed using set theoretic primitives such as cardinality and intersection. The proof is by induction on tilings. It is trivial for the empty tiling. For a non-empty one, we note that the last domino consists of a white square and a black square, added to another tiling that (by induction) has the same number of white and black squares.

No Tilings for Mutilated Boards

default proof of a negation

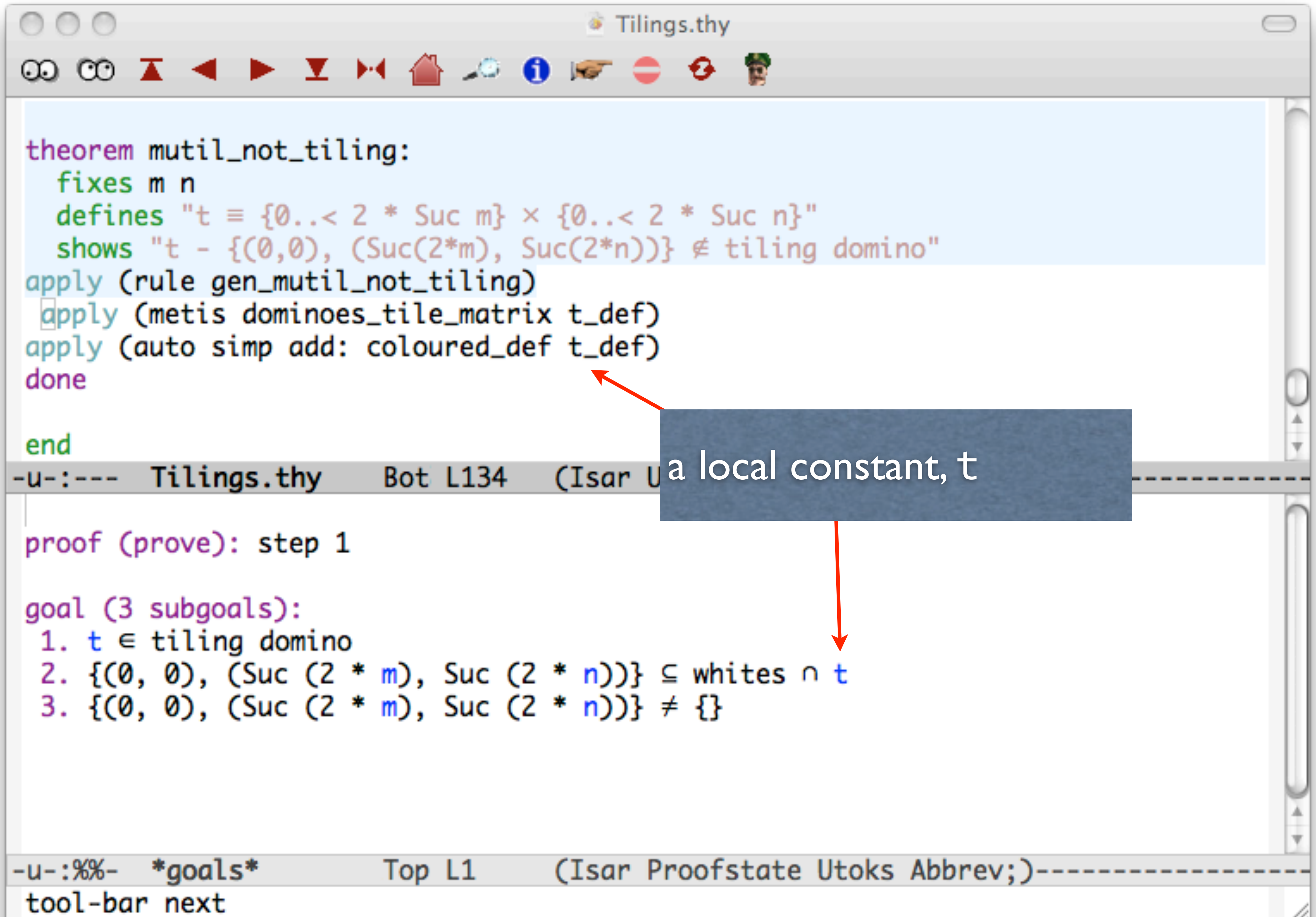
accumulating some facts

```
theorem gen_mutil_not_tiling:
  assumes "t ∈ tiling domino" "sqs ⊆ whites ∩ t" "sqs ≠ {}"
  shows "(t - sqs) ∉ tiling domino"
proof
  assume tm: "t - sqs ∈ tiling domino"
  have fsqs: "finite sqs" using assms
    by (metis Int_subset_iff finite_subset tiling_finite [OF domino_finite])
  hence c: "0 < card sqs" "0 < card (whites ∩ t)" using assms
    by (auto simp add: card_gt_0_iff)
  have "card (whites ∩ (t-sqs)) = card ((whites ∩ t) - sqs)"
    by (metis Int_Diff)
  also have "... < card (whites ∩ t)" using fsqs c assms
    by (auto simp add: card_Diff_subset)
  also have "... = card (blacks ∩ t)"
    by (blast intro: tiling_domino_0_1 assms)
  also have "... = card (blacks ∩ (t - sqs))"
    proof -
      have "blacks ∩ (t - sqs) = blacks ∩ t" using assms
        by (force simp add: coloured_def)
      thus ?thesis by simp
    qed
  finally show False using tiling_domino_0_1 [OF tm] by auto
qed
-u-:---  Tilings.thy     70% L103    (Isar Utoks Abbrev; Scripting )------------------
```

card (whites ∩ (t - sqs)) < card (blacks ∩ (t - sqs))

The other crucial point is that if some white squares are removed, then there will be fewer white squares than black ones; although obvious to us, this proof requires the series of calculations shown on the slide. Once we have established this inequality, then it is trivial to show that the remaining squares cannot be tiled.

The Final Proof...

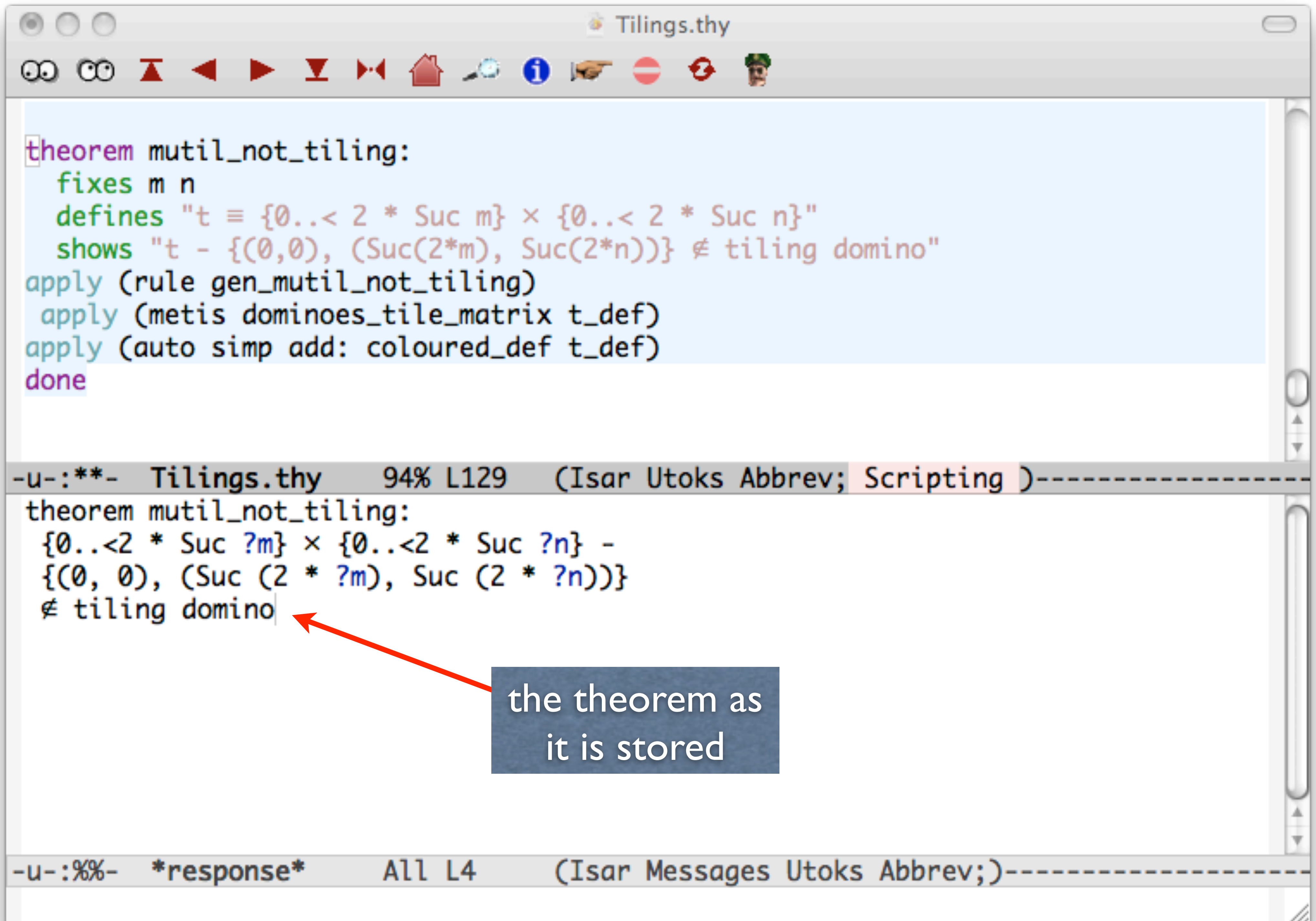


An 8 x 8 chess board can be generalised slightly, but the dimensions must be even (otherwise, the removed squares will not be white) and positive (otherwise, nothing can be removed).

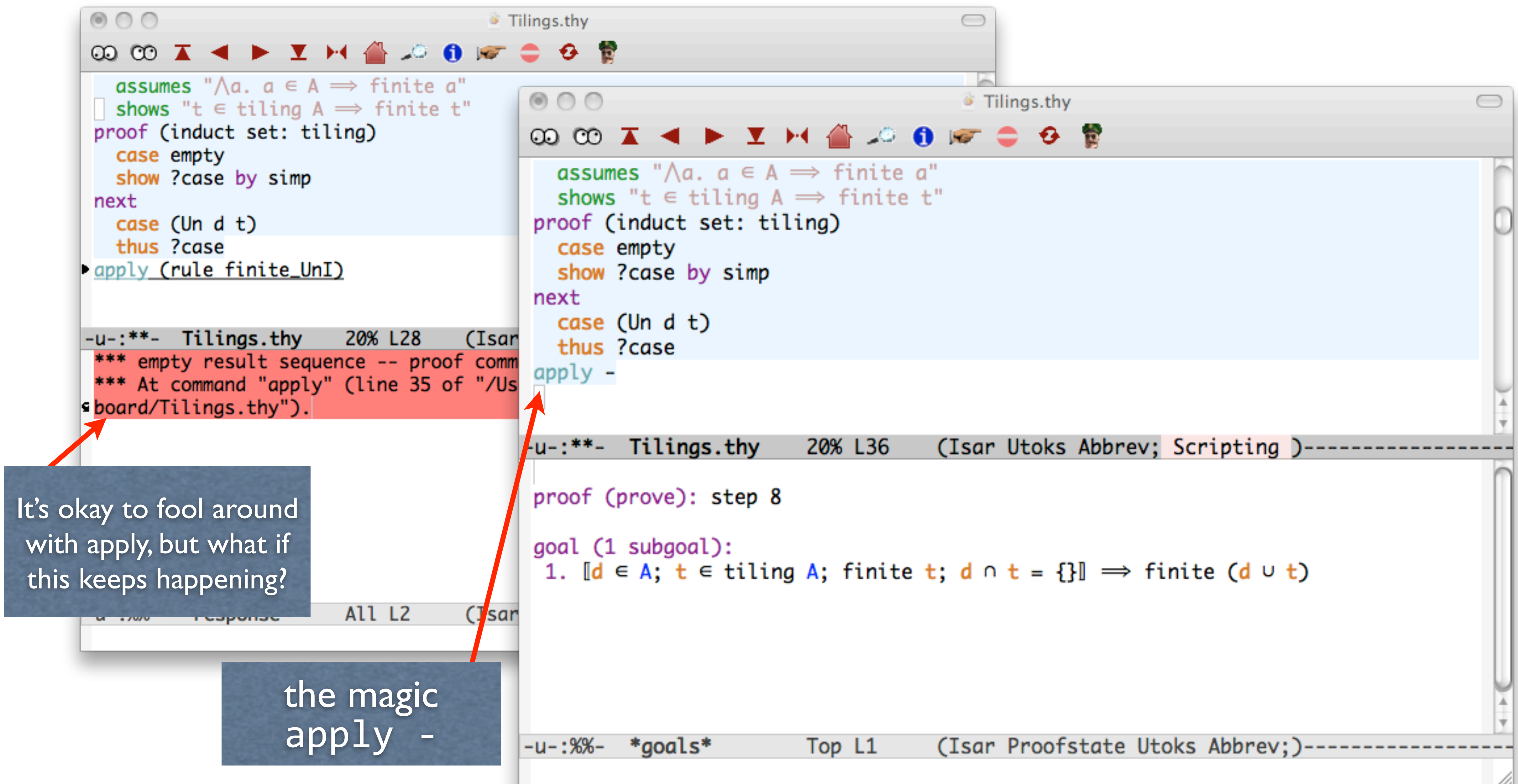
Here we display yet another novelty: a "`defines`" element. Within the proof, `t` is a constant whose definition is available as the theorem `t_def`. But once the proof is finished, Isabelle stores a theorem that does not mention `t` at all.

The "fixes" element is necessary because otherwise the "`defines`" element will be rejected on the grounds that it has "hanging" variables (`m` and `n`) on the right-hand side.

The Result for Chess Boards



Finding Structured Proofs



A common way to arrive at structured proofs is to look for a short sequence of `apply`-steps that solve the goal at hand. If successful, you can even leave this sequence (terminated by "`done`") as part of the proof, though it is better style to shorten it to a use of "`by`". Sometimes however almost everything you try produces an error message. The problem may be that you are piping facts into your proof using `then`/`hence`/`thus`/`using`. Some proof methods (in particular, "`rule`" and its variants) expect these facts to match a premise of the theorem you give to "`rule`". The simplest way to deal with this situation is to type `apply -`, which simply inserts those facts as new assumptions. It would be very ugly to leave `-` as a step in your final proof, but it is useful when exploring.

Other Facets of Isabelle

- *Document preparation*: you can generate $\LaTeX$ documents from your theories.
- *Axiomatic type classes*: a general approach to polymorphism and overloading when there are shared laws.
- *Code generation*: you can generate executable code from the formal functional programs you have verified.
- *Locales*: encapsulated contexts, ideal for formalising abstract mathematics.

See the *Tutorial*, section **4.2**, for an introduction to document preparation.

Locales are documented in the "Tutorial to Locales and Locale Interpretation" by Clemens Ballarin, which can be downloaded from Isabelle's documentation page.

Axiomatic Type Classes

- Controlled overloading of operators, including + − × / ^ ≤ and even gcd
- Can define concept hierarchies abstractly:
    - Prove theorems about an operator from its axioms
    - Prove that a type belongs to a class, making those theorems available
- Crucial to Isabelle's formalisation of arithmetic

Axiomatic type classes are inspired by the type class concept in the programming language Haskell, which is based on the following seminal paper:

Philip Wadler and Stephen Blott. How to make ad-hoc polymorphism less ad hoc. In *16th Annual Symposium on Principles of Programming Languages*, pages 60–76. ACM Press, 1989.

A very early version was available in Isabelle by 1993:

Tobias Nipkow. Order-sorted polymorphism in Isabelle. In Gérard Huet and Gordon Plotkin, editors, *Logical Environments*, pages 164–188. Cambridge University Press, 1993.

More recent papers include the following:

Markus Wenzel. Type Classes and Overloading in Higher-Order Logic. *In*: Elsa L. Gunter and Amy P. Felty, *Theorem Proving in Higher Order Logics*. Springer Lecture Notes In Computer Science 1275 (1997), 307 - 322.

Lawrence C. Paulson. Organizing Numerical Theories Using Axiomatic Type Classes. *J. Automated Reasoning* **33** 1 (2004), 29–49.

Full documentation is available: see "Haskell-style type classes with Isabelle/Isar", which can be downloaded from Isabelle's documentation page, `http://www.cl.cam.ac.uk/research/hvg/Isabelle/documentation.html`

Code Generation

- Isabelle definitions can be translated to equivalent ML and Haskell code.
- Inefficient and non-executable parts of definitions can be replaced by equivalent, efficient terms.
- Algorithms can be verified and then executed.
- The method `eval` provides *reflection*: it proves equations by execution.

See “Code generation from Isabelle/HOL theories”, by Florian Haftmann; it can be downloaded from Isabelle’s documentation page.