

# 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 However, the *Tutorial* is very long and a little outdated; although its details remain correct, it presents a style of proof that has become increasingly obsolete with the advent of structured proofs and ever greater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.

Tobias Nipkow has just written a new tutorial entitled *Programming and Proving in Isabelle/HOL*. It is much shorter than the original *Tutorial*, and much more up-to-date. If you would like to read a tutorial from cover to cover, this is the one to read.

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

# Interactive Formal Verification

## *I: Introduction*

Lawrence C Paulson  
Computer Laboratory  
University of Cambridge

# Motivation

- Complex systems almost inevitably contain bugs.
- Debugging suffers from diminishing returns. Many critical bugs are *never fixed!*
- Critical systems (avionics, ...) are required to meet a standard of  $10^{-9}$  failures per hour. Testing to such a standard is infeasible.

“Program testing can be used to show the presence of bugs, but never to show their absence!”

— Edsger W. Dijkstra

# 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/>

# The LCF Architecture

- A small kernel implements the logic and can generate theorems.
- All specification methods and automatic proof procedures expand to full proofs.
- Unsoundness is less likely with this architecture
- ... but the implementation is more complicated, and performance can suffer.
- Used in Isabelle, HOL, Coq but not PVS or ACL2.

# Theorem Provers: Key Features

- Logical formalism (higher-order, type theory etc.)
- Control issues:
  - User interface / Proof language
  - Automation
- Libraries of formalised mathematics
- Tools: typesetting, library search,...

# Isabelle

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



# Higher-Order Logic

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

“HOL = functional programming + logic”

# Basic Syntax of Formulas

formulas  $A, B, \dots$  can be written as

$(A)$

$t = u$

$\sim A$

$A \& B$

$A \mid B$

$A \dashrightarrow B$

$A \leftrightarrow B$

$\text{ALL } x. A$

$\text{EX } x. A$

(Among many others)

Isabelle also supports symbols such as

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

# Some Syntactic Conventions

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

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

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  $((\neg A) \wedge (B = C)) \vee D$

# Basic Syntax of Terms

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

# Types

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

# Product Types for Pairs

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

# Function Types

- Infix operators are curried functions
  - $+ :: \text{nat} \Rightarrow \text{nat} \Rightarrow \text{nat}$
  - $\& :: \text{bool} \Rightarrow \text{bool} \Rightarrow \text{bool}$
  - Curried function notation:  $\lambda x y. t$
- Function arguments can be paired
  - Example:  $\text{nat} * \text{nat} \Rightarrow \text{nat}$
  - 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



# 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)

# 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

# Formal Theories

- Collections of specifications: types, constants, functions, sets and relations...
- even *axioms* occasionally, but it is safer to define explicit models satisfying desired properties.

*Axiom systems are frequently inconsistent!*

- Theories can specify mathematics, formal models or abstract implementations.

name of the  
new theory

# A Tiny 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

# 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"
```

```
type_synonym val = nat -- "values"
```

```
type_synonym state = "loc => val"
```

```
type_synonym aexp = "state => val"
```

```
type_synonym bexp = "state => bool" -- "functions on states"
```

new basic types

```
datatype
```

```
com = SKIP
```

```
| Assign loc aexp  
| Semi com com  
| Cond bexp com com  
| While bexp com
```

concrete syntax for commands

```
("_ ::= _" 60)
```

```
("_ ; _" [60, 60] 10)
```

```
("IF _ THEN _ ELSE _" 60)
```

```
("WHILE _ DO _" 60)
```

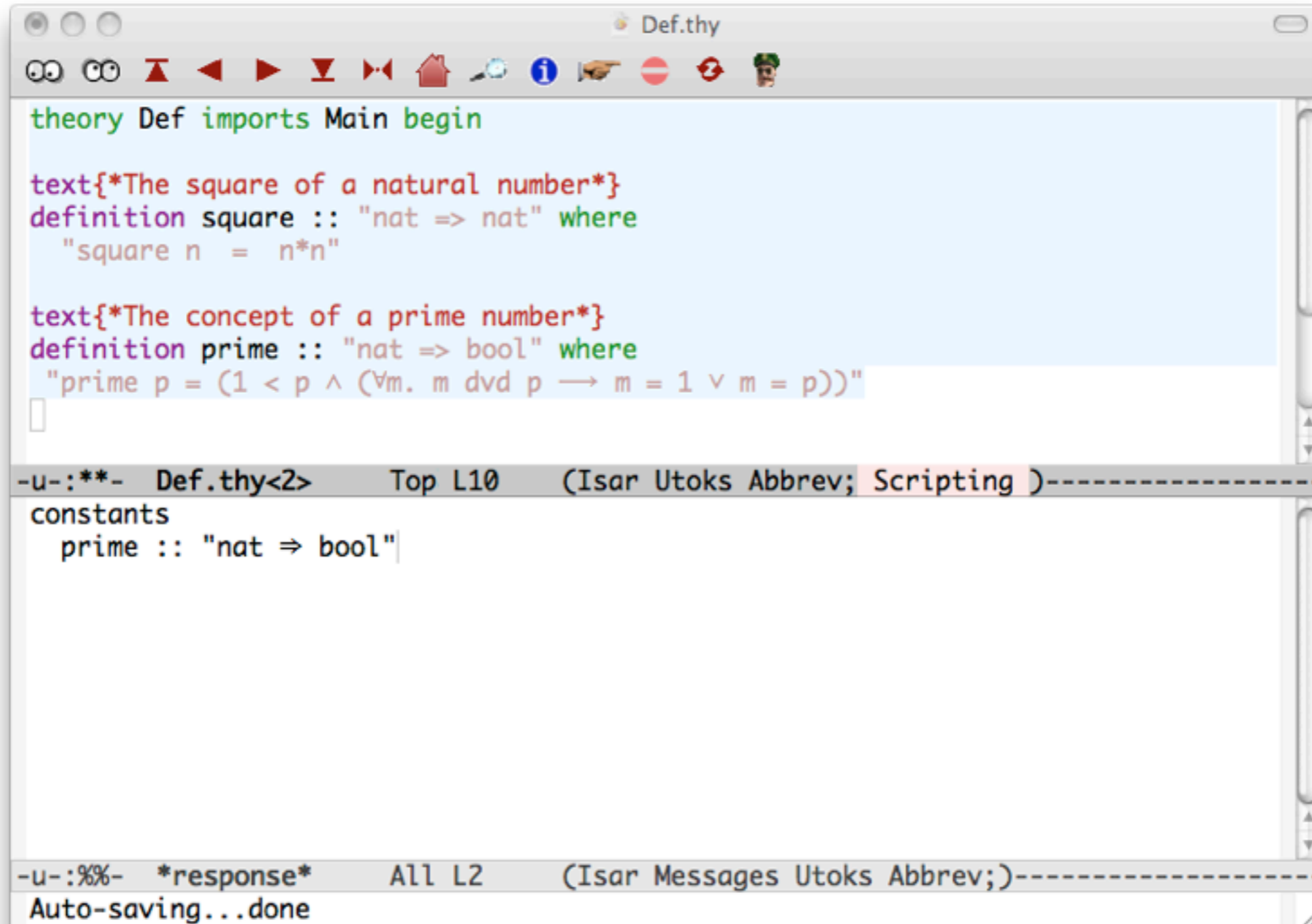
recursive type of commands



# 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*.

# Basic Constant Definitions



```
theory Def imports Main begin

text{*The square of a natural number*}
definition square :: "nat => nat" where
  "square n = n*n"

text{*The concept of a prime number*}
definition prime :: "nat => bool" where
  "prime p = (1 < p ^ (∀m. m dvd p → m = 1 ∨ m = p))"

□

-u-:**- Def.thy<2>      Top L10      (Isar Utoks Abbrev; Scripting )-----
constants
  prime :: "nat ⇒ bool"|

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

# 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 []  $\Rightarrow$  [] | x#xs  $\Rightarrow$  ...)
  - 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  $\text{Nil}$  or  $\text{Cons}(x, xs)$  (for some  $x$  and  $xs$ ).

# Proof General

The screenshot shows the Isabelle Proof General interface with two windows. The top window, titled 'DemoList.thy', contains the following code:

```
theory DemoList imports Plain (*not Main, because lists are built-in*)
begin

datatype 'a list = Nil | Cons 'a "'a list"

fun app :: "'a list => 'a list => 'a list" where
  "app Nil ys = ys"
| "app (Cons x xs) ys = Cons x (app xs ys)"

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

The first two lines of the code are highlighted in blue. A red arrow points from a text box to this highlighted area.

The bottom window, titled '\*goals\*', shows the proof attempt:

```
-u-:--- DemoList.thy Top L10 (Isar Utoks Abbrev: Scripting )-----
proof (prove): step 0
goal (1 subgoal):
1. app xs Nil = xs
```

Two red arrows point from text boxes to the 'proof (prove): step 0' line and the goal '1. app xs Nil = xs'.

processed material highlighted in blue

Isabelle's output shown in a separate window

the very start of a proof attempt

# Proof by Induction

```
theory DemoList imports Plain (*not Main, because lists are built-in*)
begin

datatype 'a list = Nil | Cons 'a "'a list"

fun app :: "'a list => 'a list => 'a list" where
  "app Nil ys = ys"
| "app (Cons x xs) ys = Cons x (app xs ys)"

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

proof (prove): step 1
goal (2 subgoals):
  1. app Nil Nil = Nil
  2.  $\wedge a xs. \text{app } xs \text{ Nil} = xs \implies \text{app } (\text{Cons } a \text{ xs}) \text{ Nil} = \text{Cons } a \text{ xs}$ 

```

structural induction  
on the list xs

base case and  
inductive step

induction hypothesis

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

```
datatype 'a list = Nil | Cons 'a "'a list"

fun app :: "'a list => 'a list => 'a list" where
  "app Nil ys = ys"
| "app (Cons x xs) ys = Cons x (app xs ys)"

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

auto proves both subgoals

```
proof (prove): step 2

goal:
No subgoals!
```

We must still issue "done" to register the theorem

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

The screenshot shows a theorem prover interface with a window titled "DemoList.thy". The interface includes a toolbar with navigation icons and a main text area containing code. A status bar at the bottom shows the current file and proof state.

```
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
```

list reversal function

```
-u-:--- DemoList.thy 22% L20 (Isar Utoks Abbrev; Scripting )-----

proof (prove): step 1
goal (2 subgoals):
1. rev (rev Nil) = Nil
2.  $\wedge a xs. \text{rev (rev xs) = xs} \Rightarrow \text{rev (rev (Cons a xs)) = Cons a xs}$ 
```

Can we prove both subgoals?

```
-u-:%%- *goals* Top L1 (Isar Proofstate Utoks Abbrev;)-----
Wrote /Users/lp15/Dropbox/ACS/1 - Introduction/DemoList.thy
```

# Stuck!

The screenshot shows a theorem prover interface with a window titled "DemoList.thy". The main editor contains the following code:

```
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
```

Below the code, the status bar shows "-u-:--- DemoList.thy 22% L22 (Isar Utoks Abbrev; Script". The main goal area shows:

```
proof (prove): step 2

goal (1 subgoal):
  1.  $\forall a\ xs. \text{rev (rev xs) = xs} \implies \text{rev (app (rev xs) (Cons a Nil)) = Cons a xs}$ 
```

The status bar at the bottom shows "-u-:%%- \*goals\* Top L1 (Isar" and "tool-bar next".

auto made progress but didn't finish

looks like we need a lemma relating rev and app!

# Stuck Again!

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

lemma [simp]: "rev (app xs ys) = app (rev ys) (rev xs)"
  apply (induct xs)
  apply auto
done

lemma rev_rev: "rev (rev xs) = xs"
  apply (induct xs)

proof (prove): step 2

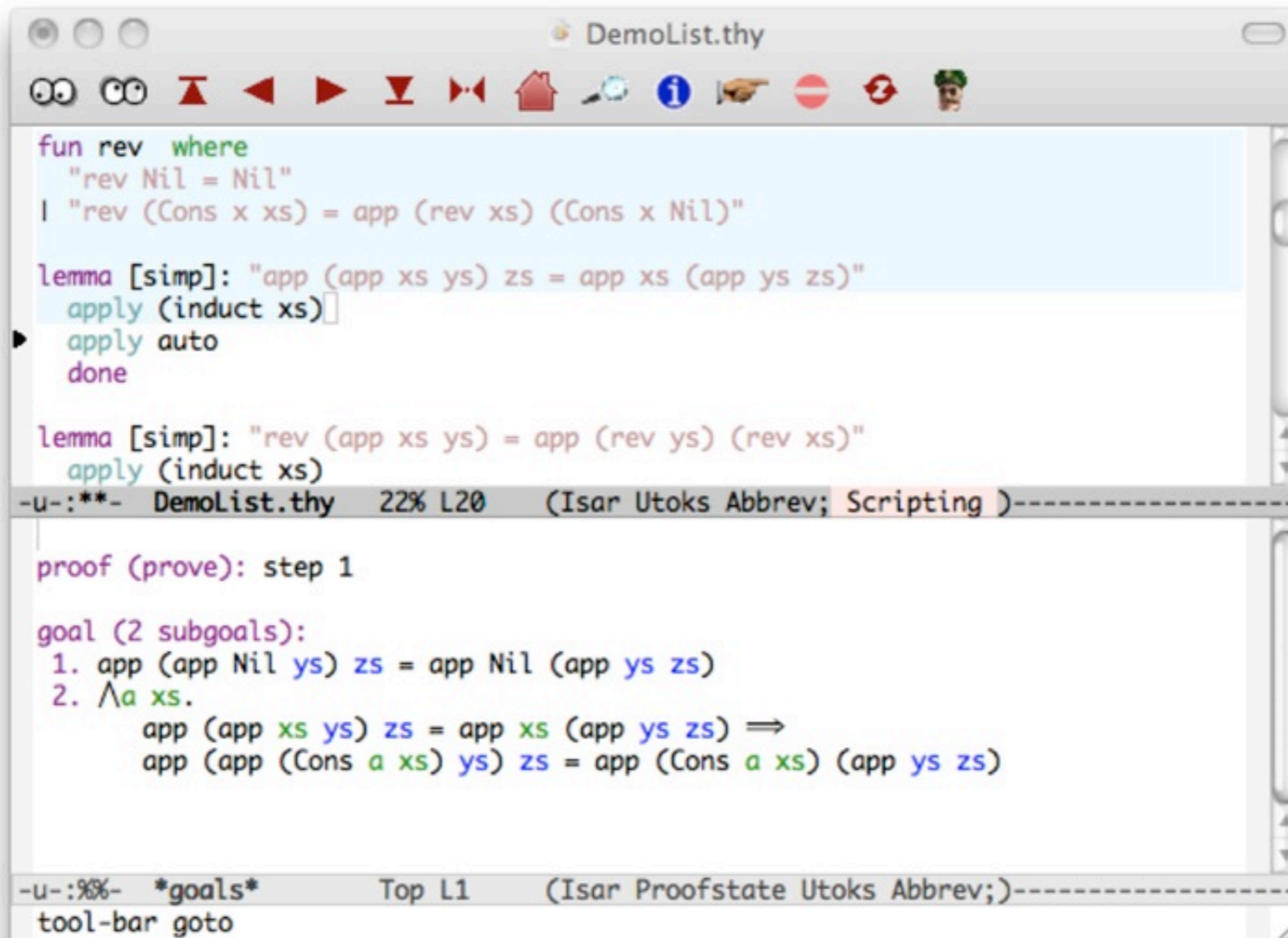
goal (1 subgoal):
1.  $\Lambda a xs.$ 
   rev (app xs ys) = app (rev ys) (rev xs)  $\Rightarrow$ 
   app (app (rev ys) (rev xs)) (Cons a Nil) =
   app (rev ys) (app (rev xs) (Cons a Nil))
```

we dreamt up a lemma...

But it needs another lemma!

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



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

lemma [simp]: "app (app xs ys) zs = app xs (app ys zs)"
  apply (induct xs)
  apply auto
  done

lemma [simp]: "rev (app xs ys) = app (rev ys) (rev xs)"
  apply (induct xs)

-u-:***- DemoList.thy 22% L20 (Isar Utoks Abbrev; Scripting )-----

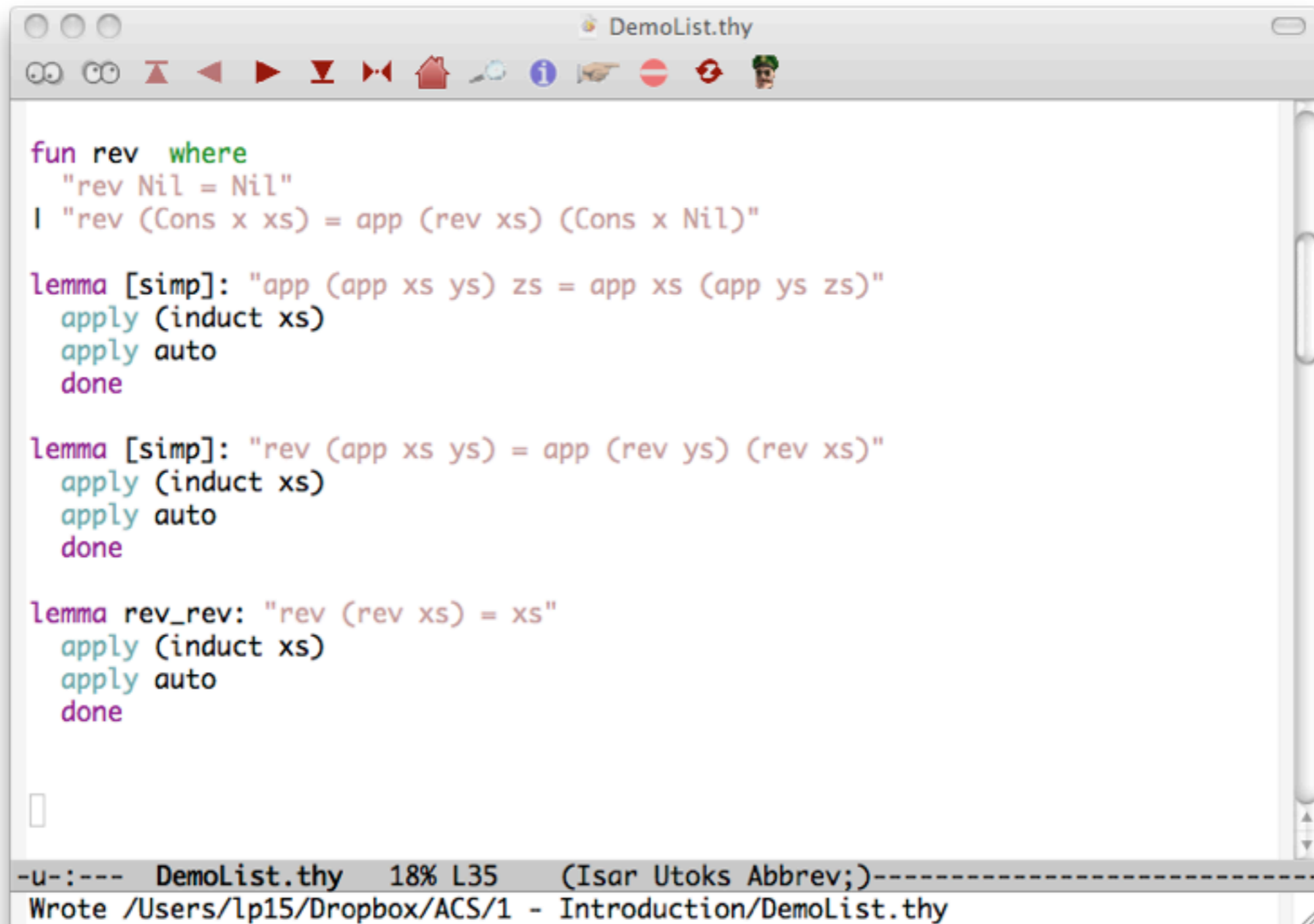
proof (prove): step 1

goal (2 subgoals):
1. app (app Nil ys) zs = app Nil (app ys zs)
2.  $\wedge a xs.$ 
   app (app xs ys) zs = app xs (app ys zs)  $\Rightarrow$ 
   app (app (Cons a xs) ys) zs = app (Cons a xs) (app ys zs)

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

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

# The Finished Proof



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

lemma [simp]: "app (app xs ys) zs = app xs (app ys zs)"
  apply (induct xs)
  apply auto
  done

lemma [simp]: "rev (app xs ys) = app (rev ys) (rev xs)"
  apply (induct xs)
  apply auto
  done

lemma rev_rev: "rev (rev xs) = xs"
  apply (induct xs)
  apply auto
  done
```

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

The lemmas must be proved in the correct order. Each is needed to prove the next.

It is actually more usable to give each lemma a name and to supply the relevant names to auto. The two lemmas proved above, especially the associativity of append, do not look like they would always be useful in simplification, so normally they would be proved without the [simp] attribute.

# Interactive Formal Verification

## 3: Elementary Proof

Lawrence C Paulson  
Computer Laboratory  
University of Cambridge

# Elements of Interactive Proof

- Quite a few theorems can be proved by a combination of *induction* and *simplification*.
- Induction can be a straightforward *structural induction* rule derived from a type declaration, but other induction rules are quite specialised.
- Simplification typically refers to *rewriting* according to the definition of a recursive function...
- but it has many refinements, including automatic *case splitting*, simple logical reasoning and sophisticated *arithmetic* reasoning.

# Goals and Subgoals

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



# Structure of a Subgoal

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

```
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
```

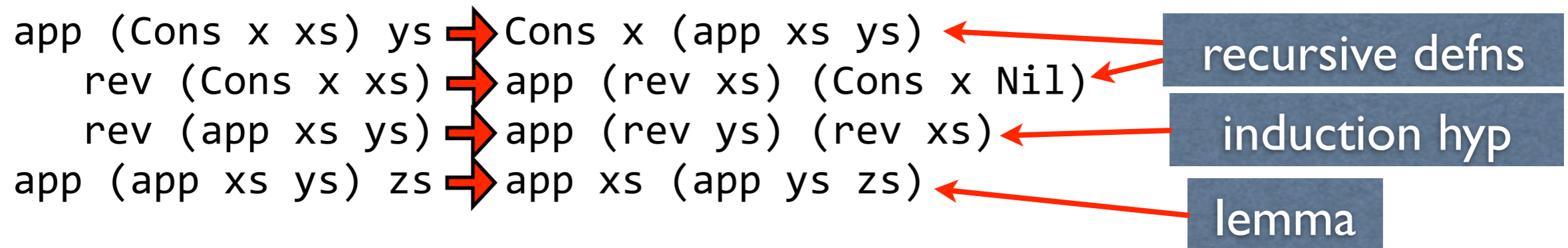
The proof state window shows the following subgoal:

```
2.  $\wedge a\ t1\ t2.$   
   $[[\text{reflect (reflect } t1) = t1; \text{reflect (reflect } t2) = t2]]$   
   $\Rightarrow \text{reflect (reflect (Br } a\ t1\ t2)) = \text{Br } a\ t1\ t2$ 
```

Annotations with arrows point to parts of the subgoal:

- A box labeled "assumptions (two induction hypotheses)" points to the two `reflect` equations in the subgoal's assumptions.
- A box labeled "parameters (arbitrary local variables)" points to the `a`, `t1`, and `t2` variables in the subgoal's parameters.
- A box labeled "conclusion" points to the final equality in the subgoal's conclusion.

# Proof by Rewriting



$$\text{rev (app (Cons a xs) ys) = app (rev ys) (rev (Cons a xs))}$$

$$\begin{aligned} &\text{rev (app (Cons a xs) ys) =} \\ &\text{rev (Cons a (app xs ys)) =} \\ &\text{app (rev (app xs ys)) (Cons a Nil) =} \\ &\text{app (app (rev ys) (rev xs)) (Cons a Nil) =} \\ &\text{app (rev ys) (app (rev xs) (Cons a Nil))} \end{aligned}$$

$$\begin{aligned} &\text{app (rev ys) (rev (Cons a xs)) =} \\ &\text{app (rev ys) (app (rev xs) (Cons a Nil))} \end{aligned}$$

# Rewriting with Equivalences

$$(x \text{ dvd } -y) = (x \text{ dvd } y)$$

$$(a * b = 0) = (a = 0 \vee b = 0)$$

$$(A - B \subseteq C) = (A \subseteq B \cup C)$$

$$(a * c \leq b * c) = ((0 < c \rightarrow a \leq b) \wedge (c < 0 \rightarrow b \leq a))$$

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

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

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.

# Conditional Rewrite Rules

$$xs \neq [] \Rightarrow \text{hd } (xs @ ys) = \text{hd } xs$$

$$n \leq m \Rightarrow (\text{Suc } m) - n = \text{Suc } (m - n)$$

$$[| a \neq 0; b \neq 0 |] \Rightarrow b / (a * b) = 1 / a$$

- *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*:  $P(x) \Rightarrow x=0$ 
  - `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

# Variations on `simp` and `auto`

using another rewrite rule

`simp add: add_assoc`

omitting a certain rule

`simp del: rev_rev (no_asm_simp)`

`simp (no_asm)`

not simplifying the assumptions

`simp_all (no_asm_simp) add: ... del: ...`

ignoring all assumptions

`auto simp add: ... del: ...`

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`

# 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.

# Proof General Tools

forward and back

find theorems

query theorem

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

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

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

proof (prove): step 1

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

stop!!

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

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

Stop is necessary to terminate simplifications or other steps that appear to be running for ever.

# Interactive Formal Verification

## 4: Advanced Recursion, Induction and Simplification

Lawrence C Paulson  
Computer Laboratory  
University of Cambridge

# Why does Induction Fail?

In a formal proof—like in a program—even trivial errors can be fatal. Everything must be set up *exactly* right...

- The statement being proved is too weak, so the induction hypothesis is too weak.
- You have chosen an inappropriate induction rule for this proof.
- Or maybe you just don't know how to make use of the induction hypotheses.

# A Failing Proof by Induction

```
Demolist.thy
length of a list
(tail-recursive)
fun itlen :: "'a list => nat => nat" where
  "itlen Nil n = n"
| "itlen (Cons x xs) n = itlen xs (Suc n)"

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

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

proof (prove): step 2
goal (1 subgoal):
1.  $\wedge xs. \text{itlen } xs \ n = \text{size } xs + n \implies \text{itlen } xs \ (\text{Suc } n) = \text{Suc } (\text{size } xs + n)$ 

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

length of a list  
(tail-recursive)

equivalent to the built-  
in length function?

May as well  
give up!

Mismatch between induction  
hypothesis and conclusion!



# Generalising the Induction

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

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

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

Insert a universal quantifier

Induction hypothesis holds for all n

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

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

(No changes need to be saved)

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

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

lemma "itlen xs n = size xs + n"
  apply (induct xs arbitrary: n)
  apply auto
  done
```

Designate a variable as “arbitrary”

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

Induction hypothesis still holds for all n!

```
-u-:--- DemoList.thy 38% L41 (Isar Utoks Abbrev; Scripting )-----
-u-:%%- *goals* Top L1 (Isar Proofstate Utoks Abbrev;)-----
Wrote /Users/lp15/Dropbox/ACS/1 - Introduction/DemoList.thy
```

The approach described above is logically similar to the one on the previous slide, but it avoids the use of a universal quantifier ( $\forall$ ) 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

The image shows a screenshot of the Isabelle/HOL proof assistant interface with the file `Primrec.thy` open. The code defines the Ackermann function `ack` and a lemma `less_ack2`. The proof of `less_ack2` is shown, with three subgoals. Red arrows point from text boxes to specific parts of the code:

- Two variables in the induction!** points to the `induct i j` in the `apply` command.
- Two variables in the recursion!** points to the `ack (Suc m) n` in the recursive call of the `ack` function definition.
- A special induction rule!** points to the `ack.induct` rule used in the `apply` command.
- The subgoals follow the recursion!** points to the first subgoal, `1.  $\wedge n. n < \text{ack } 0 \ n$` .

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

lemma less_ack2 [iff]: "j < ack i j"
  apply (induct i j rule: ack.induct)
  apply auto

proof (prove): step 1

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

# Recursion: Key Points

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

# Special Induction Rules

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

# Another Unusual Recursion

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

lemma set_merge[simp]: "set (merge xs ys) = set xs ∪ set ys"
apply(induct xs ys rule: merge.induct)
apply auto
done
```

recursive calls are guarded by conditions

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

2 induction hypotheses, guarded by conditions!

-u-:--- MergeSort.thy 19% L2  
-u-:%%- \*goals\* Top L1 (Isar Proofstate Utoks Abbrev;)-  
Wrote /Users/lp15/Dropbox/ACS/4 - Advanced Recursion/MergeSort.thy

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

# Proof Outline

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

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

=

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

=

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

=

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

The 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 datatype.
- The simplifier can (upon request!) perform case-splits analogous to those for “if”.
- Case splits in *assumptions* (not the conclusion) never happen unless requested.



# Case-Splits for Lists

```
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 <= y & ordered (y#xs))"
```

# Case-Splitting in Action

```
lemma ordered_merge [simp]: "ordered (merge xs ys) =
  apply (induct xs ys rule: merge.induct)
  apply simp_all
  apply (auto split: list.split)
-u:--- MergeSort.thy 52% L27 (Isar Utoks Abbrev; Scripting )-----

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

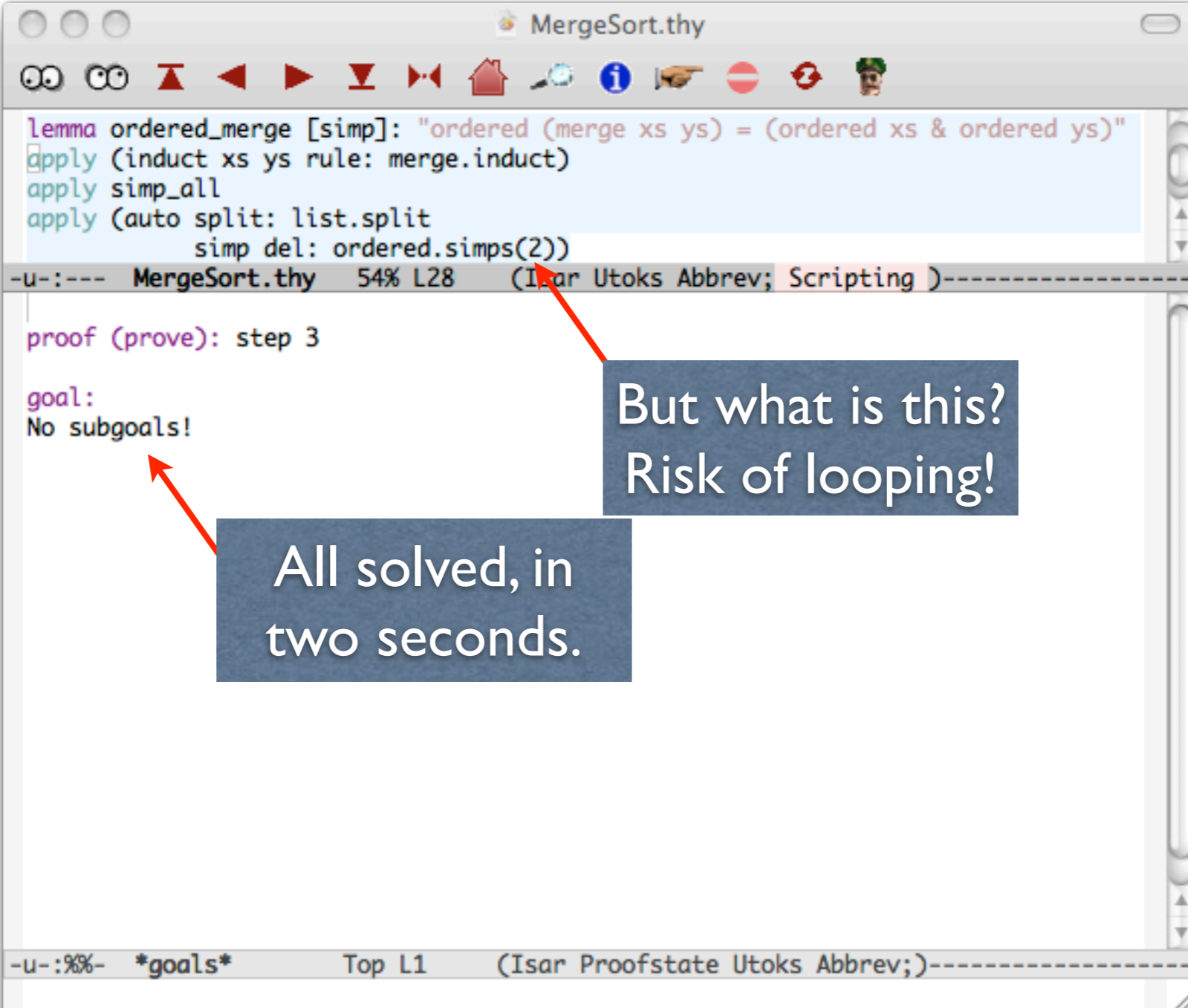
Automatic case splitting to the rescue!

Help! Look at all the case-splits!

There isn't room to show the full subgoal, but the second part of the conjunction (beginning with  $\neg x \leq y$ ) 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 screenshot shows a theorem prover window titled "MergeSort.thy". The main text area contains the following code:

```
lemma ordered_merge [simp]: "ordered (merge xs ys) = (ordered xs & ordered ys)"
apply (induct xs ys rule: merge.induct)
apply simp_all
apply (auto split: list.split
      simp del: ordered.simps(2))
```

Below the code, a status bar indicates "MergeSort.thy 54% L28 (Isar Utoks Abbrev; Scripting)".

The proof state shows:

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

Two blue callout boxes with white text and red arrows point to the proof state:

- One box points to the "goal:" line and contains the text "All solved, in two seconds."
- Another box points to the "No subgoals!" line and contains the text "But what is this? Risk of looping!"

At the bottom of the window, another status bar shows "\*goals\*" and "Top L1 (Isar Proofstate Utoks Abbrev;)"

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

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

by

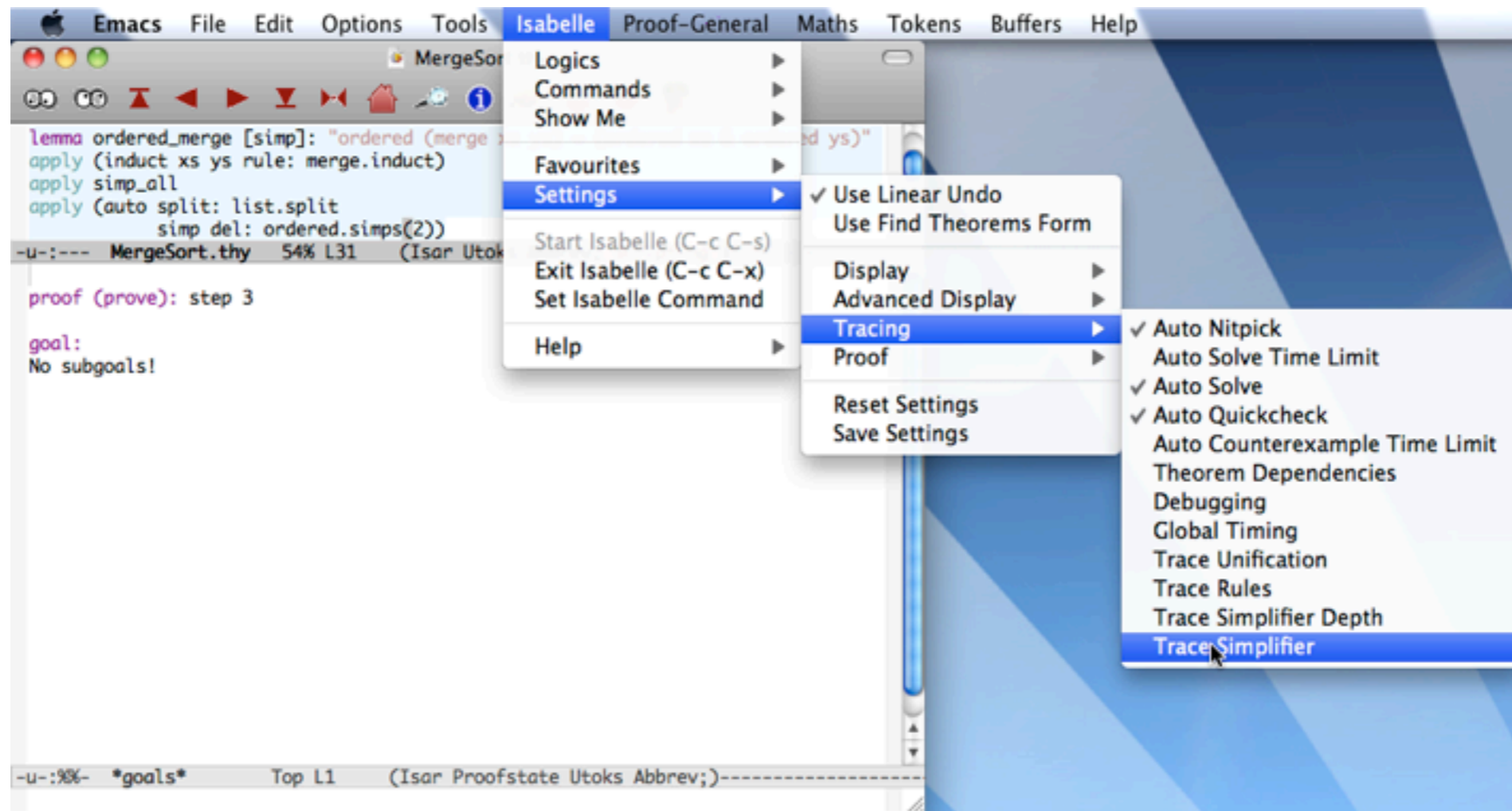
$$(xs = [] \rightarrow P(a)) \wedge (\forall a l. xs = a \# l \rightarrow P(b a l))$$

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

# 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

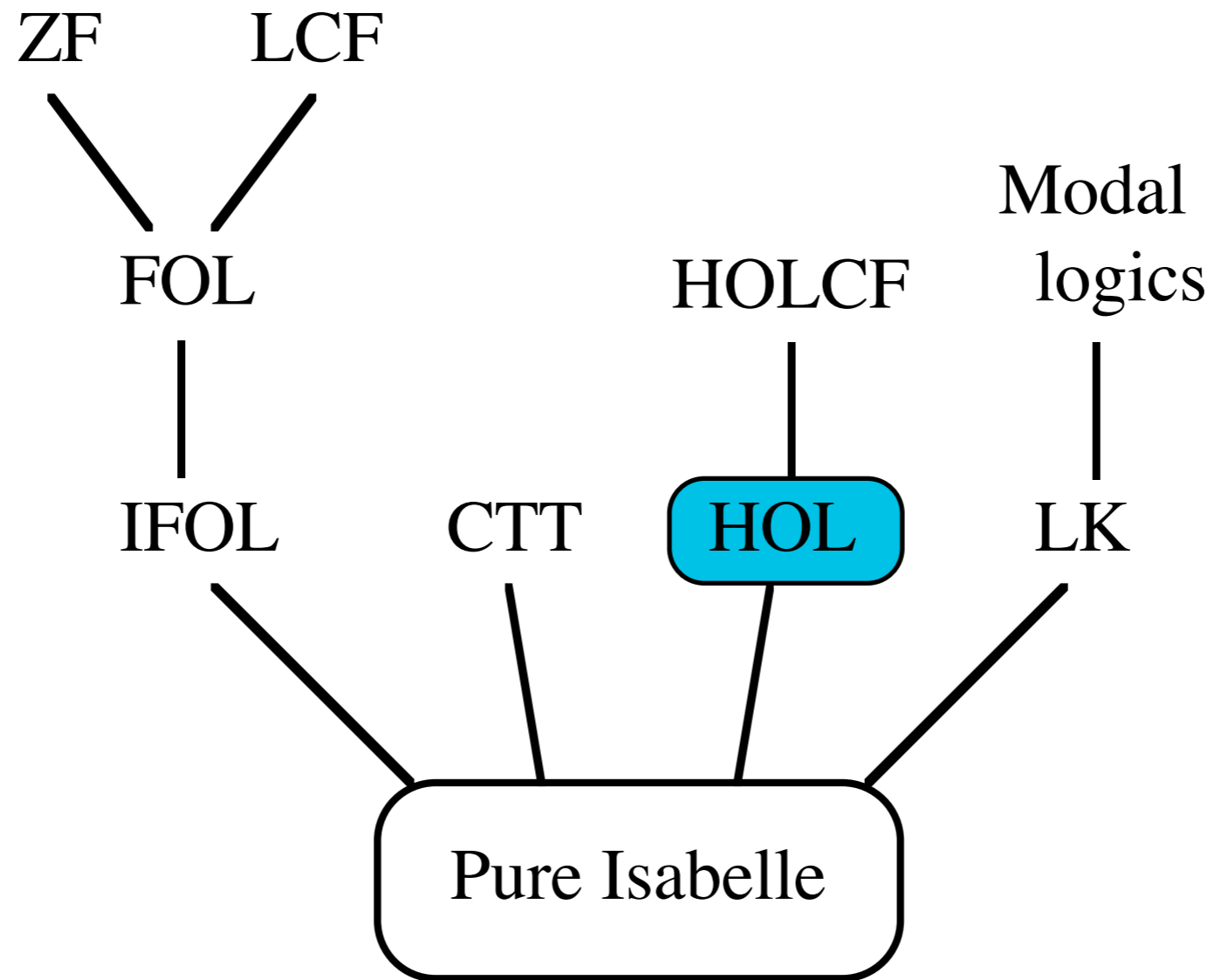
Lawrence C Paulson  
Computer Laboratory  
University of Cambridge

# 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)
  - $\wedge$  (universal quantification)



# Isabelle's Family of Logics



# Natural Deduction Basics

- Proof is done using mainly *inference rules* rather than axioms.
- For each logical symbol, there are rules to *introduce* and *eliminate* it.
- Assumptions can be *introduced* and *discharged*.
- Contrast with *Hilbert-style* proof systems, where typically the main inference rule is *modus ponens*...
- and there are many cryptic axioms, each combining a number of logical symbols.

# Natural Deduction in Isabelle

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

$$P \Rightarrow (Q \Rightarrow P \wedge Q)$$

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

$$P \wedge Q \Rightarrow P$$

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

$$P \wedge Q \Rightarrow Q$$

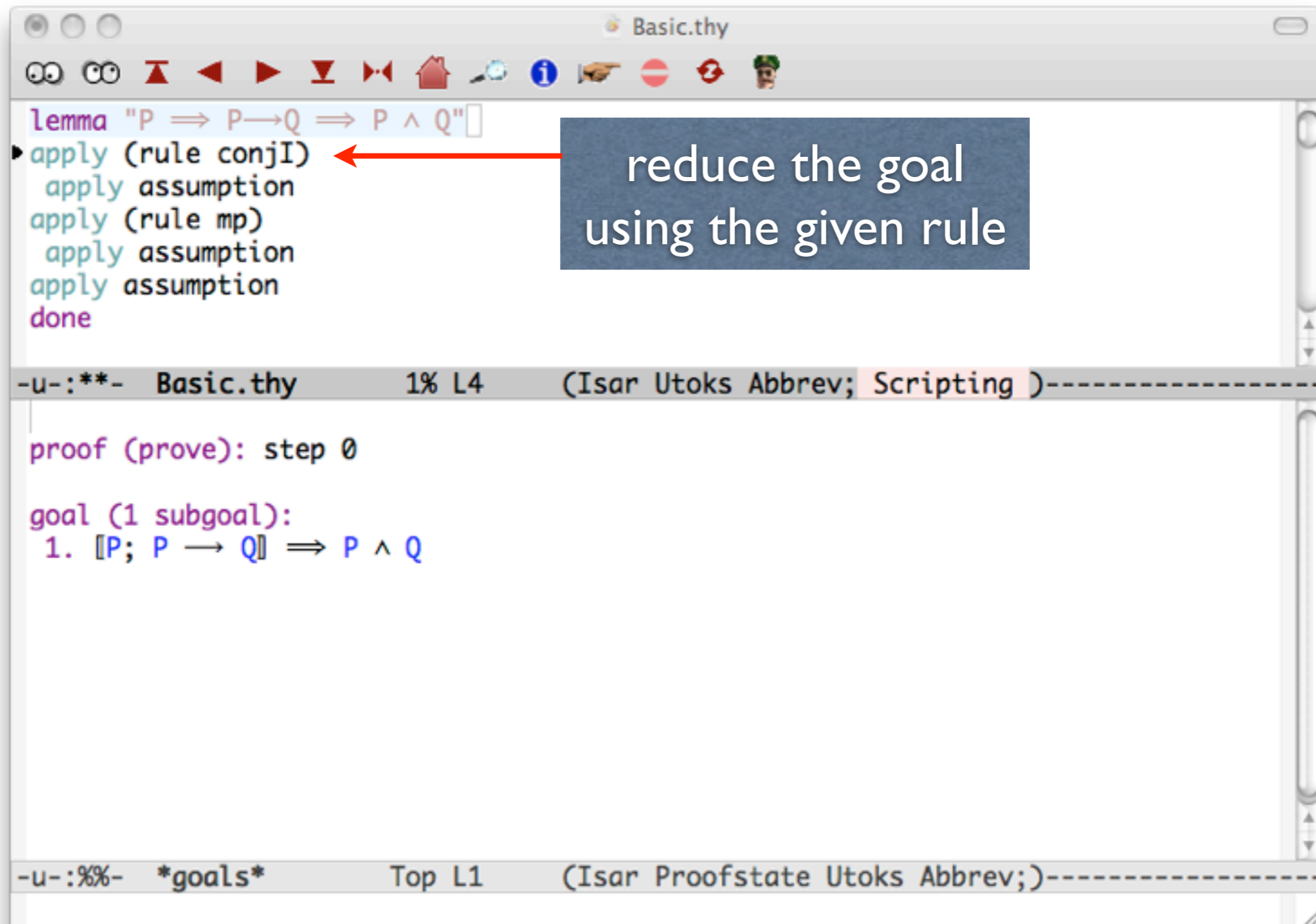
$$\frac{P \rightarrow Q \quad P}{Q}$$

$$P \rightarrow Q \Rightarrow (P \Rightarrow Q)$$

# 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  $[[P; Q]] \Rightarrow R$

# A Trivial Proof



```
lemma "P  $\implies$  P  $\longrightarrow$  Q  $\implies$  P  $\wedge$  Q"
  apply (rule conjI)
    apply assumption
    apply (rule mp)
    apply assumption
  done
```

reduce the goal using the given rule

```
-u-:***- Basic.thy 1% L4 (Isar Utoks Abbrev; Scripting )-----
```

```
proof (prove): step 0
```

```
goal (1 subgoal):
```

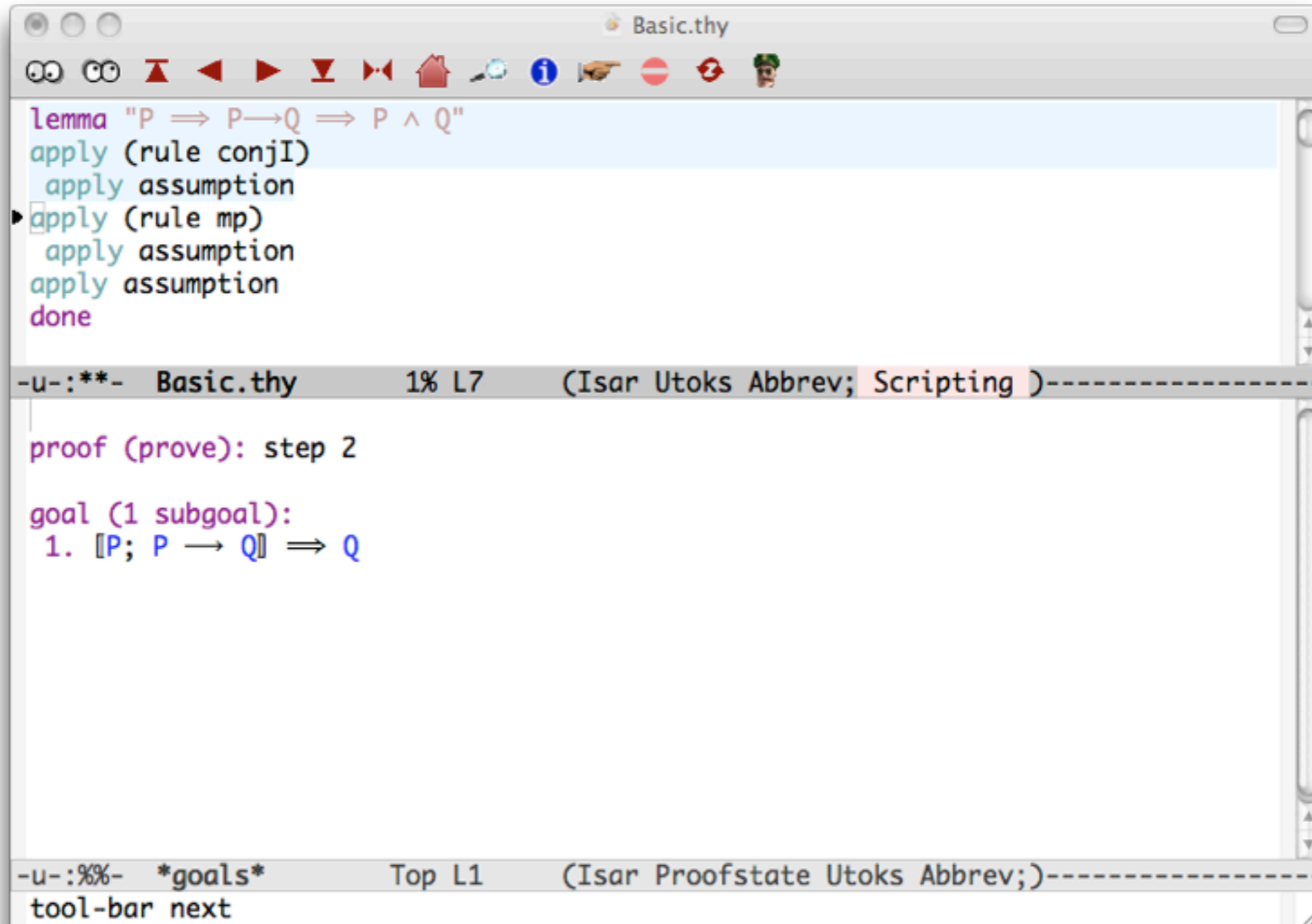
```
1. [[P; P  $\longrightarrow$  Q]]  $\implies$  P  $\wedge$  Q
```

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

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



```
Basic.thy
lemmas "P ==> P -> Q ==> P ^ Q"
apply (rule conjI)
  apply assumption
  apply (rule mp)
    apply assumption
    apply assumption
  done

-u-:***- Basic.thy 1% L7 (Isar Utoks Abbrev; Scripting )-----
proof (prove): step 2
goal (1 subgoal):
  1. [[P; P -> Q]] ==> Q

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

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

```
lemma "P ==> P -> Q ==> P ^ Q"
  apply (rule conjI)
  apply assumption
  apply (rule mp)
  apply assumption
  apply assumption
  done
```

-u-:\*\*\*- Basic.thy 1% L8 (Isar Utoks Abbrev; Scripting )-----

```
proof (prove): step 3
```

goal (2 subgoals):

1.  $\llbracket P; P \rightarrow Q \rrbracket \Rightarrow ?P3 \rightarrow Q$
2.  $\llbracket P; P \rightarrow Q \rrbracket \Rightarrow ?P3$

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

tool-bar next

We need some instance of mp!

formula placeholder

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  $?P \Rightarrow (?Q \Rightarrow ?P \ \& \ ?Q)$ .

# Unknowns and Unification

The screenshot shows a theorem prover interface with a toolbar at the top. The main window displays a lemma proof:

```
lemma "P  $\implies$  P  $\rightarrow$  Q  $\implies$  P  $\wedge$  Q"  
  apply (rule conjI)  
  apply assumption  
  apply (rule mp)  
  apply assumption  
  apply assumption  
done
```

Below the lemma, a status bar indicates the current position: `-u-:**- Basic.thy 1% L9 (Isar Utoks Abbrev; Scripting )`. The main window also shows a proof step:

```
proof (prove): step 4  
goal (1 subgoal):  
  1. [[P; P  $\rightarrow$  Q]]  $\implies$  P
```

A red arrow points from a blue box containing the text `?P3 has been replaced by P` to the `P` in the goal statement. The bottom status bar shows: `-u-:%%- *goals* Top L1 (Isar Proofstate Utoks Abbrev;)` and `tool-bar next`.

Proving  $?P3 \rightarrow Q$  from the assumption  $P \rightarrow Q$  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

$$\frac{\begin{array}{c} [P] \\ \vdots \\ Q \end{array}}{P \rightarrow Q}$$

$$(P \Rightarrow Q) \Rightarrow P \rightarrow Q$$

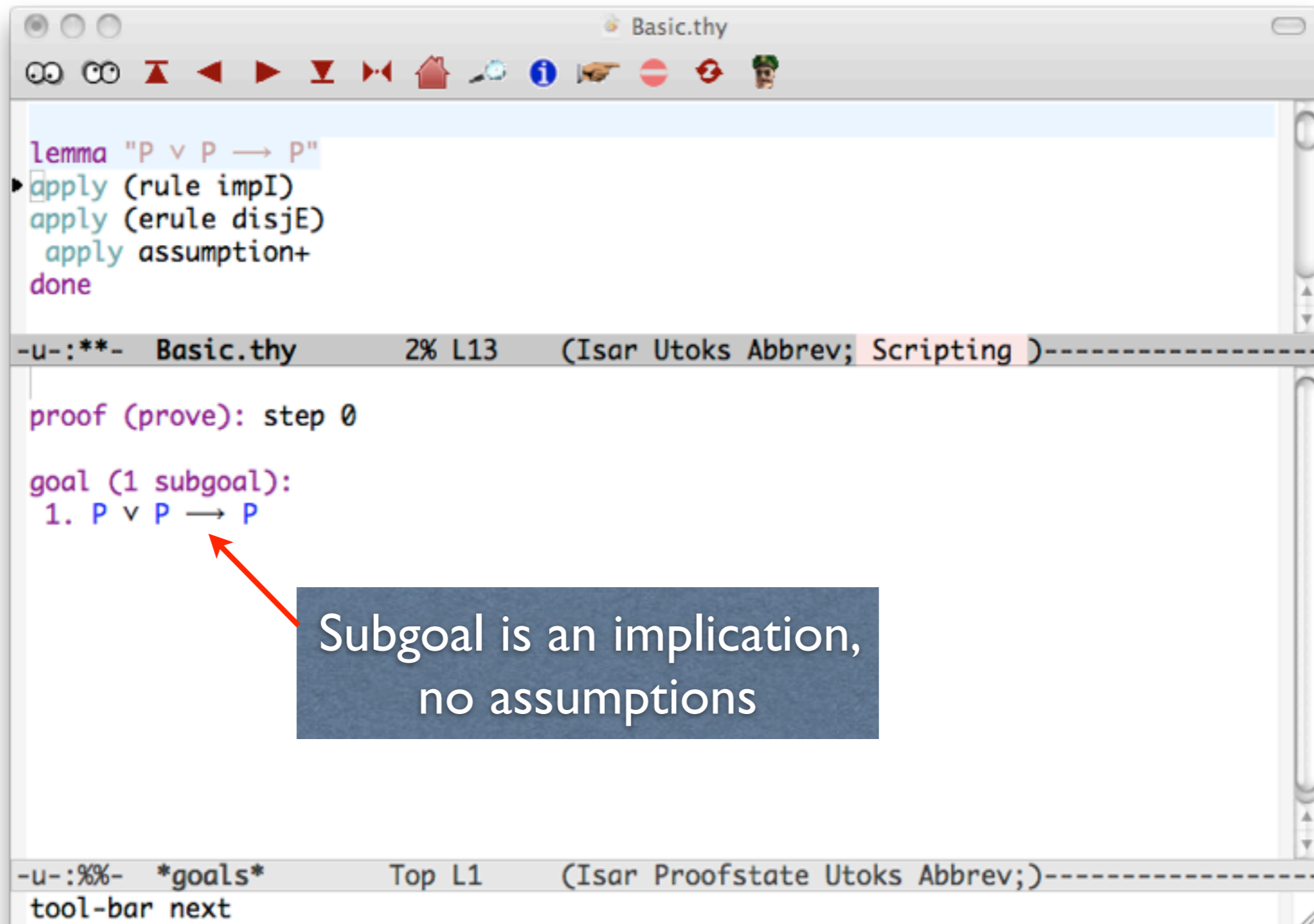
$$\frac{P \vee Q \quad \begin{array}{c} [P] \\ \vdots \\ R \end{array} \quad \begin{array}{c} [Q] \\ \vdots \\ R \end{array}}{R}$$

$$[[P \vee Q; P \Rightarrow R; Q \Rightarrow R]] \Rightarrow R$$

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  $P \rightarrow Q$ , it suffices to assume  $P$  and prove  $Q$ .

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

# A Proof using Assumptions



```
lemma "P ∨ P → P"
  apply (rule impI)
  apply (erule disjE)
  apply assumption+
  done
```

-u-:\*\*\*- Basic.thy 2% L13 (Isar Utoks Abbrev; Scripting )-----

```
proof (prove): step 0
  goal (1 subgoal):
  1. P ∨ P → P
```

Subgoal is an implication,  
no assumptions

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

# After Implies-Introduction

```
lemma "P ∨ P → P"
  apply (rule impI)
  apply (erule disjE)
  apply assumption+
  done
```

**Prove P using  $P \vee P$**

```
proof (prove): step 1
  goal (1 subgoal):
  1.  $P \vee P \Rightarrow P$ 
```

**Assumption will be used, then **deleted****

**\*goals\*** Top L1 (Isar Proofstate Utoks Abbrev;)-----  
tool-bar next

# Disjunction Elimination

The screenshot shows a theorem prover interface with two panels. The top panel displays a lemma proof:

```
lemma "P ∨ P → P"  
  apply (rule impI)  
  apply (erule disjE)  
  apply assumption+  
done
```

A red arrow points from the text "erule is good with elimination rules" to the `erule disjE` line in the proof.

The bottom panel shows the goal state:

```
proof (prove): step 2  
goal (2 subgoals):  
1. P ⇒ P  
2. P ⇒ P
```

A red arrow points from the text "An instance of ?P ∨ ?Q has been found" to the first goal, `1. P ⇒ P`.

The interface includes a toolbar with navigation icons and status bars at the bottom of each panel. The top status bar shows "Basic.thy 2% L15 (Isar Utoks Abbrev; Scripting)". The bottom status bar shows "\*goals\* Top L1 (Isar Proofstate Utoks Abbrev;)" and "tool-bar next".

# The Final Step

```
lemma "P ∨ P → P"  
  apply (rule impI)  
  apply (erule disjE)  
  apply assumption+  
done
```

+ applies a method one or more times

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

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

# Quantifiers

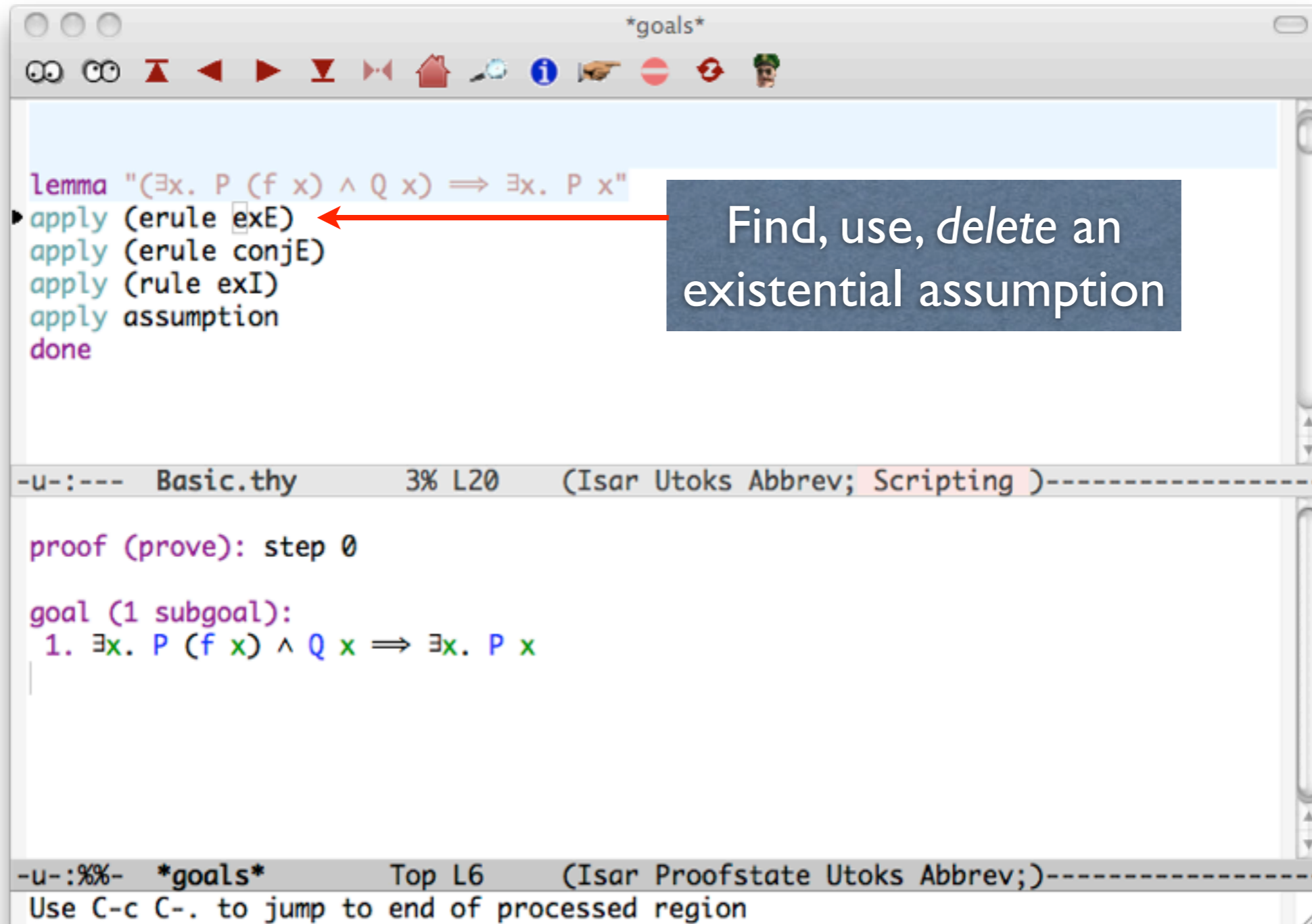
$$\frac{P(t)}{\exists x. P(x)}$$

$$P(x) \Rightarrow \exists x. P(x)$$

$$\frac{\exists x. P(x) \quad \begin{array}{c} [P(x)] \\ \vdots \\ Q \end{array}}{Q} \quad \llbracket \exists x. P(x); \forall x. P(x) \Rightarrow Q \rrbracket \Rightarrow Q$$

meta-universal quantifier  
states the variable condition

# A Tiny Quantifier Proof



The screenshot shows a proof assistant window titled '\*goals\*'. The main area contains a lemma and its proof steps. A red arrow points from a text box to the 'exE' rule in the proof steps.

```
lemma "( $\exists x. P (f x) \wedge Q x$ )  $\implies \exists x. P x$ "  
• apply (erule exE) ←  
  apply (erule conjE)  
  apply (rule exI)  
  apply assumption  
done
```

Find, use, *delete* an existential assumption

-u-:--- Basic.thy 3% L20 (Isar Utoks Abbrev; Scripting )-----

```
proof (prove): step 0  
goal (1 subgoal):  
1.  $\exists x. P (f x) \wedge Q x \implies \exists x. P x$   
|
```

-u-:%%- \*goals\* Top L6 (Isar Proofstate Utoks Abbrev;)-----  
Use C-c C-. to jump to end of processed region

# Conjunction Elimination

```
lemma "( $\exists x. P (f x) \wedge Q x$ )  $\implies$   $\exists x. P x$ "
apply (erule exE)
apply (erule conjE)
apply (rule exI)
apply assumption
done
```

Find, use, delete a conjunctive assumption

```
proof (prove): step 1
goal (1 subgoal):
1.  $\wedge x. P (f x) \wedge Q x \implies \exists x. P x$ 
```

The  $x$  that is claimed to exist

-u-:--- Basic.thy 3% L20 (Isar Utoks Abbrev; Scripting )-----

-u-:%%- \*goals\* Top L6 (Isar Proofstate Utoks Abbrev;)-----

Use C-c C-. to jump to end of processed region

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 $\exists$ -Introduction

```
*goals*  
lemmas "( $\exists x. P (f x) \wedge Q x$ )  $\implies \exists x. P x$ "  
apply (erule exE)  
apply (erule conjE)  
• apply (rule exI) ← Apply the rule exI  
apply assumption  
done  
-----  
-u-:--- Basic.thy 3% L20 (Isar Utoks Abbrev; Scripting )-----  
proof (prove): step 2  
goal (1 subgoal):  
1.  $\wedge x. [P (f x); Q x] \implies \exists x. P x$  ← Two assumptions instead of one  
-----  
-u-:%%- *goals* Top L6 (Isar Proofstate Utoks Abbrev;)-----  
Use C-c C-. to jump to end of processed region
```

# An Unknown for the Witness

```
lemma "( $\exists x. P (f x) \wedge Q x$ )  $\implies \exists x. P x$ "
apply (erule exE)
apply (erule conjE)
apply (rule exI)
apply assumption
done

-u-:--- Basic.thy 3% L20 (Isar Utoks Abbrev; Scripting )-----

proof (prove): step 3
goal (1 subgoal):
1.  $\wedge x. [P (f x); Q x] \implies P (?x4 x)$ 

-u-:%%- Use C-c C-. to jump to end of processed region
```

Proof by assumption will unify these two terms

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,  $x$  may appear in the witness.

# Done!

The screenshot shows a proof assistant window titled '\*goals\*'. The main area contains a lemma and its proof steps. The lemma is  $(\exists x. P (f x) \wedge Q x) \implies \exists x. P x$ . The proof steps are: `apply (erule exE)`, `apply (erule conjE)`, `apply (rule exI)`, and `apply assumption`. The proof is completed with `done`. Below the main area, there are two status bars. The first status bar shows the file `Basic.thy` at line 3, column 20, with the text `(Isar Utoks Abbrev; Scripting )`. The second status bar shows the current goal state `*goals*` at line 6, column 6, with the text `(Isar Proofstate Utoks Abbrev;)` and a note: `Use C-c C-. to jump to end of processed region`.

```
lemma "( $\exists x. P (f x) \wedge Q x$ )  $\implies \exists x. P x$ "
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: Structured Proofs

Lawrence C Paulson  
Computer Laboratory  
University of Cambridge

# A Proof about “Divides”

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

```
Struct.thy
lemma "(k dvd (n + k)) = (k dvd (n::nat))"
apply (auto simp add: dvd_def)
□

proof (prove): step 1
goal (2 subgoals)
1.  $\wedge ka. n + k = k * ka \implies \exists ka. n = k * ka$ 
2.  $\wedge ka. \exists kb. k * ka + k = k * kb$ 
```

We unfold the definition and get...?

an assumption

locally bound variables

A messy proof with two subgoals...

# Complex Subgoals

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

# A Structured Proof

```
Struct.thy
lemma "(k dvd (n + k)) = (k dvd (n::nat))"
proof (auto simp add: dvd_def)
  fix m
  assume "n + k = k * m"
  hence "n = k * (m - 1)"
    by (metis diff_add_inverse diff_mult_distrib2 nat_add_commute nat_mult_1_right)
  thus "∃m'. n = k * m'"
  by blast
next
  fix m
  show "∃m'. k * m + k = k * m'"
  by (metis mult_Suc_right nat_add_commute)
qed
-u-:--- Struct.thy      2% L11      (Isar Utoks Abbrev; Scripting )-----
proof (prove): step 6
using this:
  n = k * (m - 1)
goal (1 subgoal):
  1. ∃m'. n = k * m'
-u-:%%- *goals*      Top L1      (Isar Proofstate Utoks Abbrev;)-----
tool-bar goto
```

But how do you write them?

# The Elements of Isar

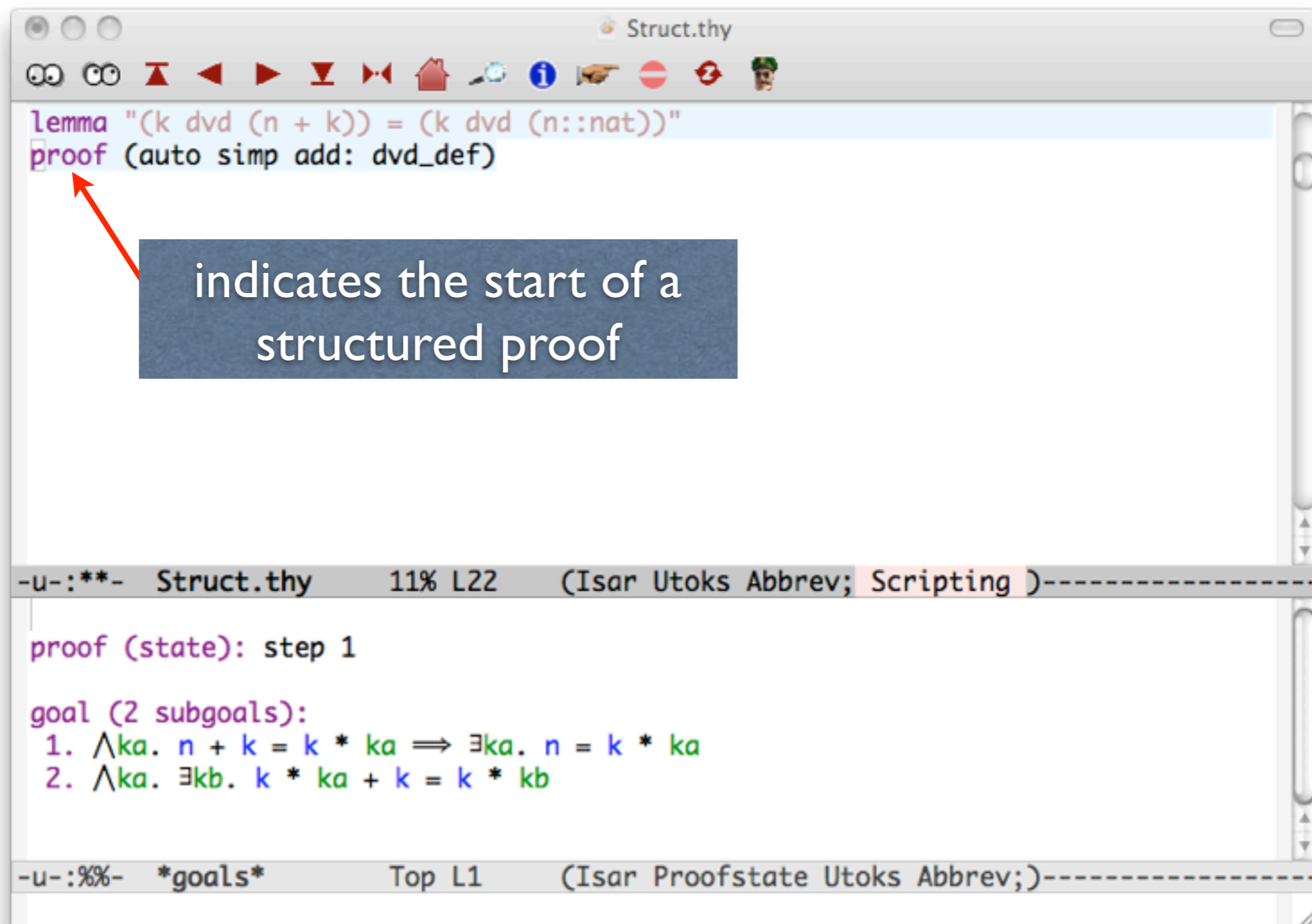
- A *proof context* holds a local variables and assumptions of a subgoal.
  - In a context, the variables are free and the assumptions are simply theorems.
  - Closing a context yields a theorem having the structure of a subgoal.
- The Isar language lets us state and prove intermediate results, express 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 screenshot shows a window titled "Struct.thy" with a toolbar at the top. The main text area contains the following code:

```
lemma "(k dvd (n + k)) = (k dvd (n::nat))"
proof (auto simp add: dvd_def)
```

A red arrow points from a dark blue box containing the text "indicates the start of a structured proof" to the `proof` keyword in the code above.

Below the main text area, there are two status bars. The first one shows "-u-:\*\*\*- Struct.thy 11% L22 (Isar Utoks Abbrev; Scripting )-----". The second one shows "-u-:%%- \*goals\* Top L1 (Isar Proofstate Utoks Abbrev;)-----".

The bottom part of the window shows the output of the proof:

```
proof (state): step 1
goal (2 subgoals):
  1.  $\wedge ka. n + k = k * ka \implies \exists ka. n = k * ka$ 
  2.  $\wedge ka. \exists kb. k * ka + k = k * kb$ 
```

indicates the start of a structured proof

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

The screenshot shows a proof skeleton in Isabelle/HOL. The code is as follows:

```
[lemma "(k dvd (n + k)) = (k dvd (n::nat))"
proof (auto simp add: dvd_def)
  fix m
  assume "n + k = k * m"
  show "∃m'. n = k * m'"
  sorry
next
  fix m
  show "∃m'. k * m + k = k * m'"
  sorry
qed
```

Annotations on the left side:

- assumption (points to `assume`)
- conclusion (points to `show`)
- dummy proofs (points to `sorry`)

Annotations on the right side:

- a name for the bound variable (points to `fix m`)
- separates proofs of goals (points to `next`)
- terminates the proof (points to `qed`)

The bottom part of the screenshot shows the Isabelle output:

```
-u-:***- Struct.thy 11% L21 (Isar Utoks Abbrev; Scripting )-----
Successful attempt to solve goal by exported rule:
  (n + k = k * ?m2) ==> ∃m'. n = k * m'

Successful attempt to solve goal by exported rule:
  ∃m'. k * ?m2 + k = k * m'

lemma (?k dvd ?n + ?k) = (?k dvd ?n)
-u-:%%- *response* All L7 (Isar Messages Utoks Abbrev;)
```

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, "Local statement will fail to refine any pending goal".

# Fleshing Out that Skeleton

```
lemma "(k dvd (n + k)) = (k dvd (n::nat))"
proof (auto simp add: dvd_def)
  fix m
  assume 1: "n + k = k * m"
  have 2: "n = k * (m - 1)" using 1
    sorry
  show "∃m'. n = k * m'" using 2
    by blast
next
  fix m
  show "∃m'. k * m + k = k * m'"
    sorry
qed
```

-u-:\*\*\*- Struct.thy 15% L37 (Isar Utoks Abbrev; Scripting )-----

Successful attempt to solve goal by exported rule:  
 $(n + k = k * ?m2) \implies \exists m'. n = k * m'$

Successful attempt to solve goal by exported rule:  
 $\exists m'. k * ?m2 + k = k * m'$

lemma (?k dvd ?n + ?k) = (?k dvd ?n)

-u-:%%- \*response\* All L7 (Isar Messages Utoks Abbrev;)

labels for facts

more labels

inserting a helpful fact

a real proof!

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

```
lemma "(k dvd (n + k)) = (k dvd (n::nat))"
proof (auto simp add: dvd_def)
  fix m
  assume 1: "n + k = k * m"
  have 2: "n = k * (m - 1)" using 1
    by (metis diff_add_inverse diff_mult_distrib2 nat_add_commute nat_mult_1_right)
  show "∃m'. n = k * m'" using 2
    by blast
next
  fix m
  show "∃m'. k * m + k = k * m'"
  sorry
qed
```

found using sledgehammer

sledgehammer does it again!

```
-u-:***- Struct.thy 20% L65 (Isar Utoks Abbrev, Scripting)
Sledgehammer: external prover "spass" for subgoal 1:
∃m'. k * m + k = k * m'
Try this command: apply (metis mult_Suc_right nat_add_commute)
For minimizing the number of lemmas try this command:
atp_minimize [atp=spass] mult_Suc_right nat_add_commute

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

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

<code>assume 1: "n + k = k * m"</code>	$\longrightarrow$	<code>assume "n + k = k * m"</code>
<code>have 2: "n = k * (m - 1)" using 1</code>	$\longrightarrow$	<code>hence "n = k * (m - 1)"</code>
<code>by (metis diff_add_inverse diff)</code>		<code>by (metis diff_add_inverse diff)</code>
<code>show "∃m'. n = k * m'" using 2</code>	$\longrightarrow$	<code>thus "∃m'. n = k * m'"</code>

- 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

```
lemma abs_m_1:
  fixes m :: int
  assumes mn: "abs (m * n) = 1"
  shows      "abs m = 1"
proof -
  have 0: "m ≠ 0" using mn
    by auto
  have "~ (2 ≤ abs m)"
    sorry
  thus "abs m = 1" using 0
    by auto
qed
```

specify m's type

declare a premise separately

proof -

restricting the range of abs m

makes the conclusion trivial

Successful attempt to solve goal by exported rule:  
|m| = 1

```
lemma abs_m_1:
  !?m * ?n = 1 ⇒ !?m = 1
```

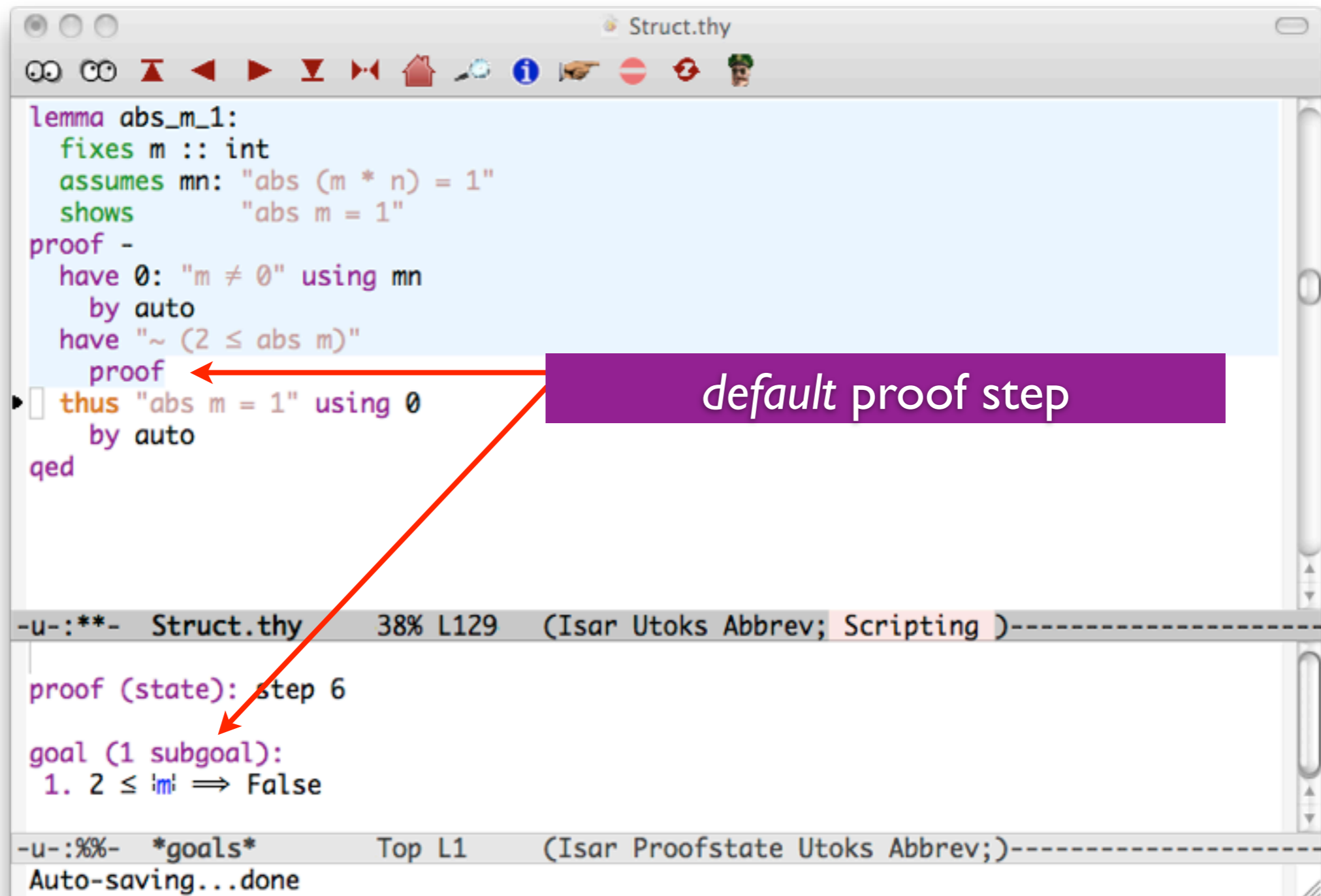
\*response\* All L5 (Isar Messages Utoks Abbrev;)

tool-bar goto

This is an example of an obvious fact is proof is not obvious. Clearly  $m \neq 0$ , since otherwise  $m \cdot n = 0$ . If we can also show that  $|m| \geq 2$  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



The screenshot shows a theorem prover interface with a main editor and a bottom status bar. The main editor contains the following code:

```
lemma abs_m_1:
  fixes m :: int
  assumes mn: "abs (m * n) = 1"
  shows      "abs m = 1"
proof -
  have 0: "m ≠ 0" using mn
    by auto
  have "~ (2 ≤ abs m)"
    proof
  ◻ thus "abs m = 1" using 0
    by auto
qed
```

A red arrow points from a purple box labeled "default proof step" to the "proof" keyword in the nested proof block. The bottom status bar shows the current proof state:

```
-u-:***- Struct.thy 38% L129 (Isar Utoks Abbrev; Scripting )-----
proof (state): step 6
goal (1 subgoal):
1. 2 ≤ |m| ⇒ False
-u-:%%- *goals* Top L1 (Isar Proofstate Utoks Abbrev;)-----
Auto-saving...done
```

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 “ $\neg (2 \leq \text{abs } m)$ ” begins with a negation sign, so the default method applies the corresponding logical inference: it reduces the problem to proving False under the assumption  $2 \leq \text{abs } m$ .

# A Nested Proof Skeleton

```
Struct.thy
lemma abs_m_1:
  fixes m :: int
  assumes mn: "abs (m * n) = 1"
  shows      "abs m = 1"
proof -
  have 0: "m ≠ 0" using mn
    by auto
  have "~ (2 ≤ abs m)"
    proof
      assume "2 ≤ abs m"
      thus "False"
        sorry
    qed
  thus "abs m = 1" using 0
    by auto
qed
```

assumption

conclusion

-u-:\*\*- Struct.thy 37% L133 (Isar Utoks Abbrev; Scripting )-----

Successful attempt to solve goal by exported rule:  
(2 ≤ |m|) ⇒ False

have ¬ 2 ≤ |m|

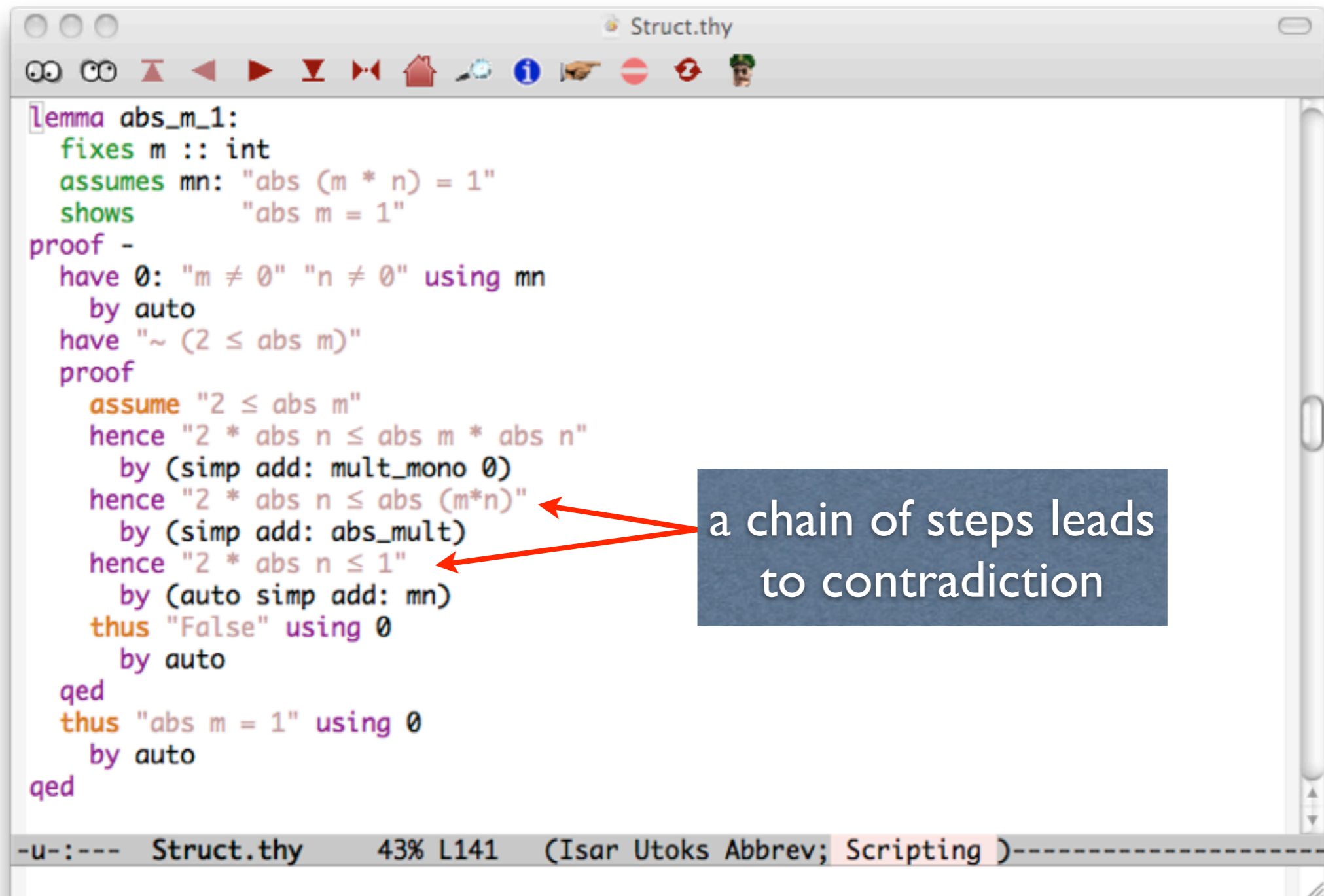
-u-:%%- \*response\* All L4 (Isar Messages Utoks Abbrev;)-----

Auto-saving...done

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



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

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

a chain of steps leads to contradiction

This example is typical of a structured proof. From the assumption,  $2 \leq \text{abs } m$ , 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  $m \neq 0$  to include as well  $n \neq 0$ , which we need to obtain a contradiction from  $2 \times \text{abs } n \leq 1$ . In fact, “0” here denotes a list of facts.

# Calculational Proofs

The screenshot shows a theorem prover interface with a window titled "Struct.thy". The main editor contains the following code:

```
have "~ (2 ≤ abs m)"
proof
  assume "2 ≤ abs m"
  hence "2 * abs n ≤ abs m * abs n"
    by (simp add: mult_mono 0)
  also have "... = abs (m*n)"
    by (simp add: abs_mult)
  also have "... = 1"
    by (simp add: mn)
  finally have "2 * abs n ≤ 1" .
  thus "False" using 0
```

A red arrow points from the text "form a series of equalities and inequalities" to the line "also have \"... = abs (m\*n)\"".

Below the code, the goal state is shown:

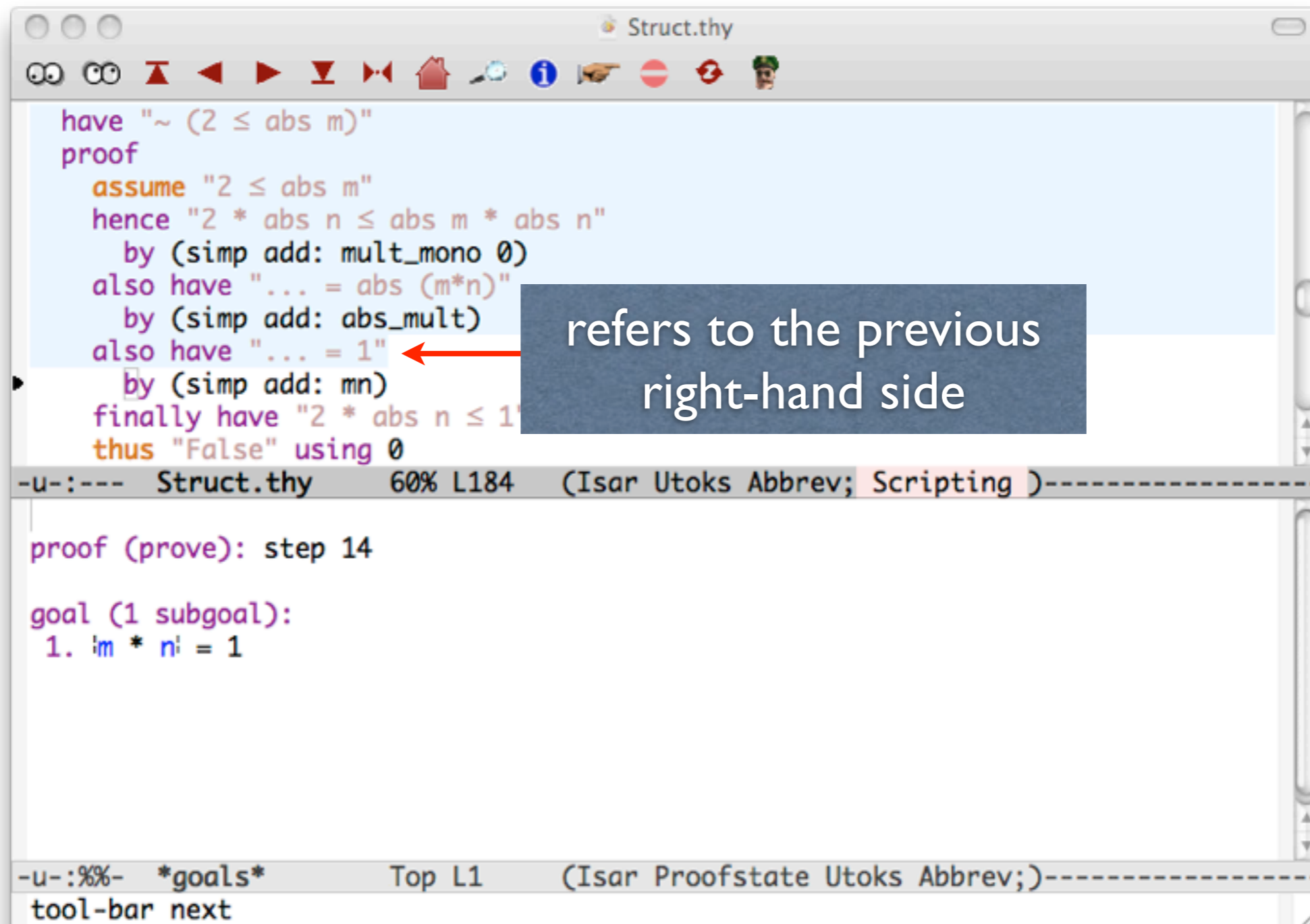
```
proof (prove): step 11
goal (1 subgoal):
  1. |m| * |n| = |m * n|
```

The status bar at the bottom indicates the current goal and provides information about the proof state: "-u-:%%- \*goals\* Top L1 (Isar Proofstate Utoks Abbrev;)----- (No files need saving)".

form a series of equalities and inequalities

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 equalities. Isar supports this notation.

# The Next Step



The screenshot shows a window titled "Struct.thy" with a toolbar at the top. The main area contains a proof script in a light blue background. A red arrow points from a dark blue callout box to the line "also have '... = 1'". Below the script is a status bar with a dashed line, and another status bar at the bottom with a solid line.

```
have "~ (2 ≤ abs m)"
proof
  assume "2 ≤ abs m"
  hence "2 * abs n ≤ abs m * abs n"
    by (simp add: mult_mono 0)
  also have "... = abs (m*n)"
    by (simp add: abs_mult)
  also have "... = 1"
    by (simp add: mn)
  finally have "2 * abs n ≤ 1"
  thus "False" using 0
```

-----  
-u-:--- Struct.thy 60% L184 (Isar Utoks Abbrev; Scripting )-----  
proof (prove): step 14  
goal (1 subgoal):  
1. |m| \* |n| = 1

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

# The Internal Calculation

The screenshot shows the Isabelle proof editor interface. The top window displays a proof script for a theorem named "Struct.thy". The script is as follows:

```
have "~ (2 ≤ abs m)"  
proof  
  assume "2 ≤ abs m"  
  hence "2 * abs n ≤ abs m * abs n"  
    by (simp add: mult_mono 0)  
  also have "... = abs (m*n)"  
    by (simp add: abs_mult)  
  also have "... = 1"  
    by (simp add: mn)  
  finally have "2 * abs n ≤ 1" .  
  thus "False" using 0
```

Red arrows from a text box on the left point to the "hence", "also have", "also have", and "finally" lines of the script, with the text "structure of a calculation".

The bottom window shows the internal calculation generated by Isabelle:

```
calculation: 2 * |n| ≤ 1
```

Red arrows from a text box on the right point to this line, with the text "Isabelle displays the internal calculation when it encounters also and finally".

At the bottom of the editor, there is a status bar with the text: "-u-:%%- \*response\* All L1 (Isar Messages Utoks Abbrev;) tool-bar next".

structure  
of a  
calculation

Isabelle displays the internal  
calculation when it encounters  
also and finally

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  $2 \times \text{abs } n \leq 1$  has been derived.

# Ending the Calculation

```
have "~ (2 ≤ abs m)"
proof
  assume "2 ≤ abs m"
  hence "2 * abs n ≤ abs m * abs n"
    by (simp add: mult_mono 0)
  also have "... = abs (m*n)"
    by (simp add: abs_mult)
  also have "... = 1"
    by (simp add: mn)
  finally have "2 * abs n ≤ 1" .
thus "False" using 0
```

Struct.thy 60% L186 (Isar Utoks Abbrev; Scripting )

```
have 2 * |n| ≤ 1
```

\*response\* All L1 (Isar Messages Utoks Abbrev;)

tool-bar next

indicates a trivial proof

We have deduced  $2 \times \text{abs } n \leq 1$

# Structure of a Calculation

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

# Interactive Formal Verification

## 7: Sets

Lawrence C Paulson  
Computer Laboratory  
University of Cambridge

# Set Notation in Isabelle

- Set notation is crucial to mathematical discourse.
- Operators such as union, intersection, powerset and image naturally express many complex constructions.
- Functions, relations, and concepts such as transitive closure are available.
- A set in higher-order logic is similar to a *boolean-valued map*: in other words, to a logical predicate.
- The elements of a set must all have the *same type*!



# Set Theory Primitives

- The type  $\alpha$  set, which is similar to  $\alpha \Rightarrow \text{bool}$
- The membership relation:  $\in$
- The subset relation:  $\subseteq$ 
  - Reflexive, anti-symmetric, transitive
- The empty set:  $\{\}$
- The universal set:  $\text{UNIV}$

# Basic Set Theory Operations

$$e \in \{x. P(x)\} \iff P(e)$$

$$e \in \{x \in A. P(x)\} \iff e \in A \wedge P(e)$$

$$e \in -A \iff e \notin A$$

$$e \in A \cup B \iff e \in A \vee e \in B$$

$$e \in A \cap B \iff e \in A \wedge e \in B$$

$$e \in \text{Pow}(A) \iff e \subseteq A$$

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

The logical equivalences shown above are effectively the definitions of the primitives shown, and any occurrences of the left-hand side formula will be replaced by the right-hand side by Isabelle's simplifier.

# Big Union and Intersection

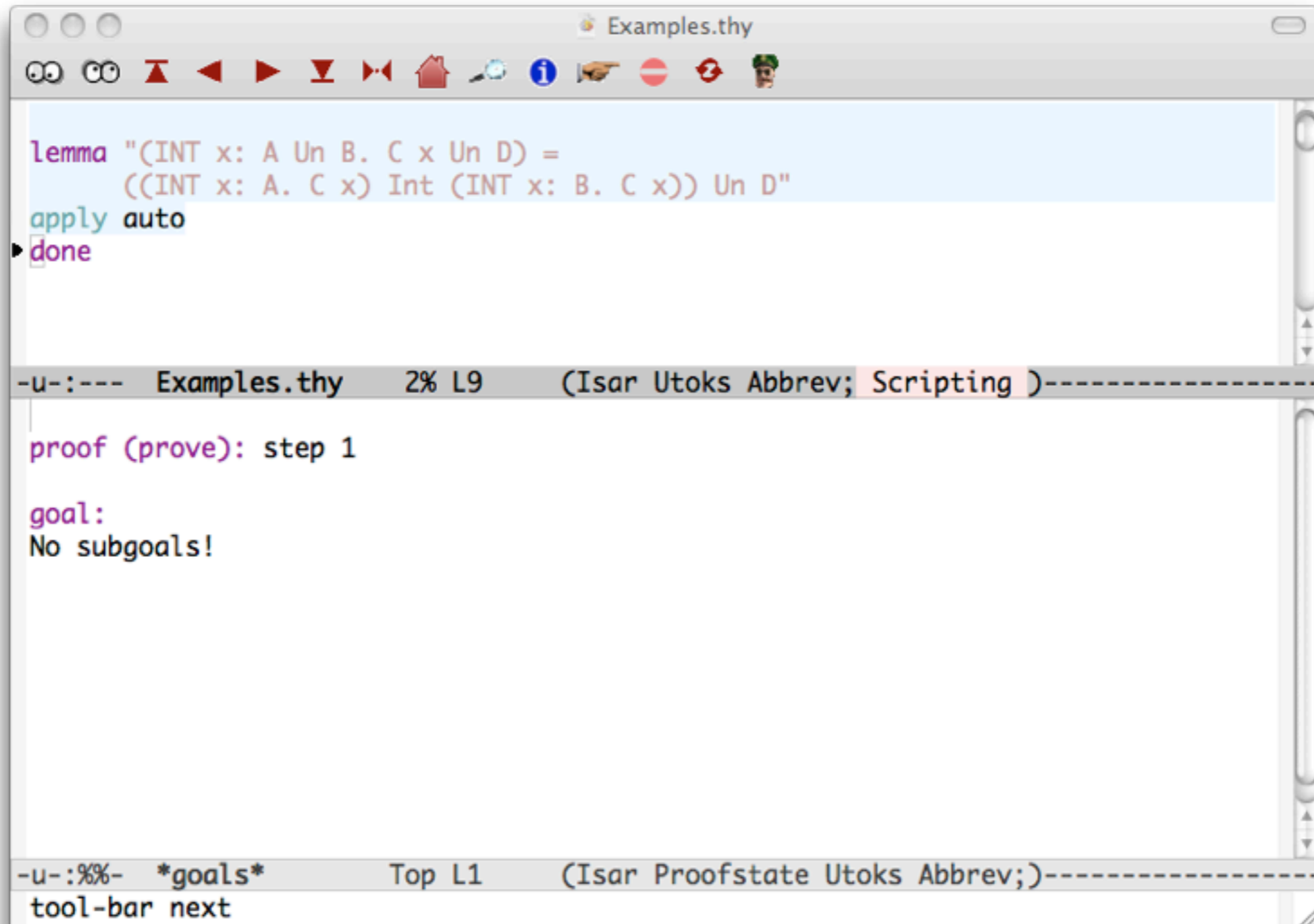
$$e \in \left( \bigcup x. B(x) \right) \iff \exists x. e \in B(x)$$
$$e \in \left( \bigcup x \in A. B(x) \right) \iff \exists x \in A. e \in B(x)$$
$$e \in \bigcup A \iff \exists x \in A. e \in x$$

And the analogous forms of intersections...

Once again, the logical equivalences are essentially definitions.

The third form of union is seldom seen.

# A Simple Set Theory Proof



```
Examples.thy
--u-:--- Examples.thy 2% L9 (Isar Utoks Abbrev; Scripting )-----
lemma "(INT x: A Un B. C x Un D) =
      ((INT x: A. C x) Int (INT x: B. C x)) Un D"
apply auto
done

proof (prove): step 1
goal:
No subgoals!

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

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.

# Functions

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

$$e \in (f^{-1} A) \iff f(e) \in A$$

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

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

# Finite Set Notation

$$\{a_1, \dots, a_n\} = \text{insert}(a_1, \dots, \text{insert}(a_n, \{\}))$$

$$e \in \text{insert}(a, B) \iff e = a \vee e \in B$$

# Finite Sets

A finite set is defined *inductively*  
in terms of  $\{\}$  and *insert*

$$\text{finite}(A \cup B) = (\text{finite } A \wedge \text{finite } B)$$

$$\text{finite } A \implies \text{card}(\text{Pow } A) = 2^{\text{card } A}$$

# Intervals, Sums and Products

$$\{..<u\} == \{x. x < u\}$$

$$\{..u\} == \{x. x \leq u\}$$

$$\{1<..\} == \{x. 1 < x\}$$

$$\{1..\} == \{x. 1 \leq x\}$$

$$\{1<..<u\} == \{1<..\} \cap \{..<u\}$$

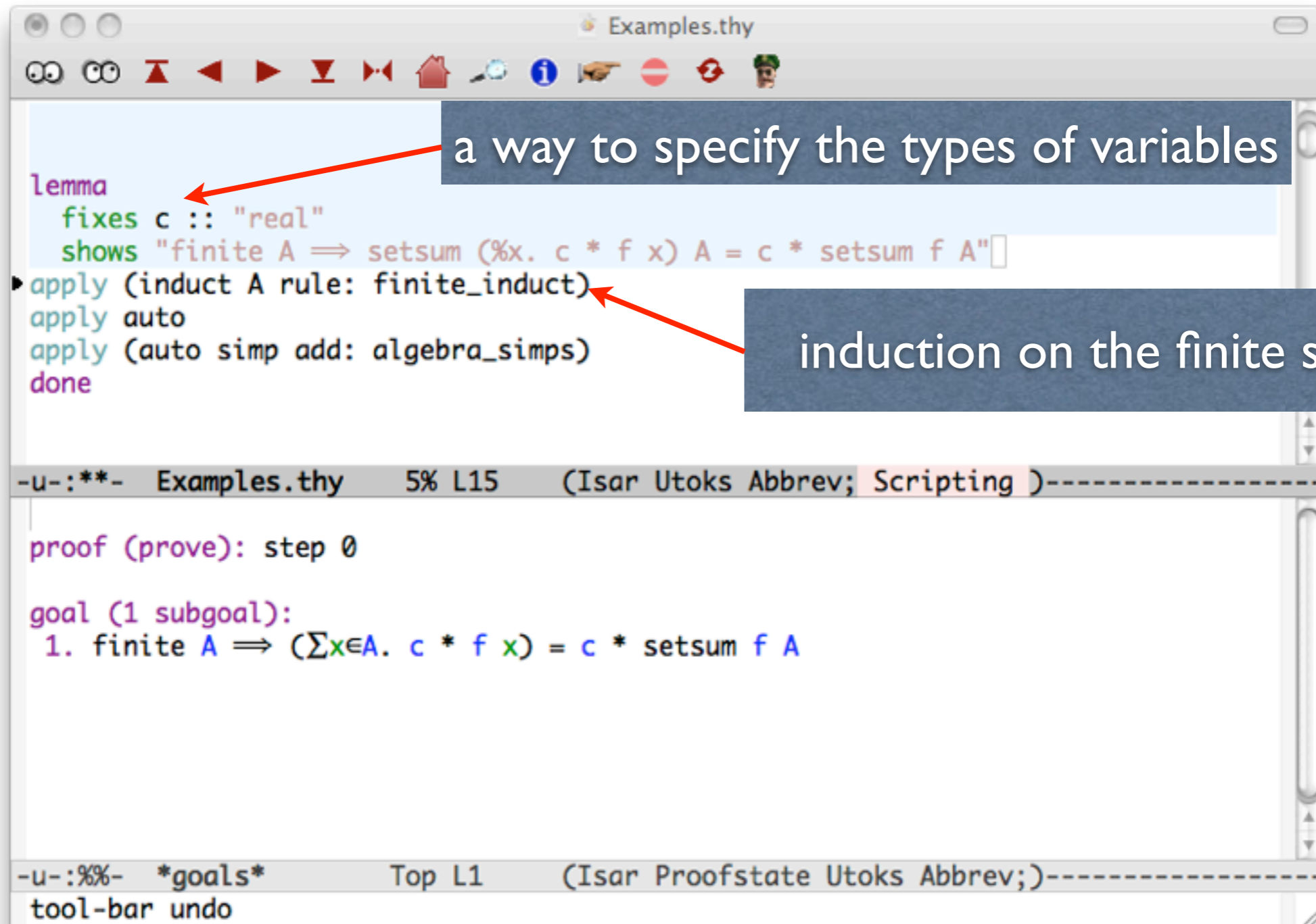
$$\{1..\<u\} == \{1..\} \cap \{..<u\}$$

setsum f A **and** setprod f A

$\sum_{i \in I}. f$  **and**  $\prod_{i \in I}. f$



# A Harder Proof Involving Sets



The screenshot shows the Isabelle/Isar IDE interface. The main editor displays the following code:

```
lemma
  fixes c :: "real"
  shows "finite A  $\Rightarrow$  setsum (%x. c * f x) A = c * setsum f A"
  apply (induct A rule: finite_induct)
  apply auto
  apply (auto simp add: algebra_simps)
  done
```

Annotations with red arrows point to specific parts of the code:

- A blue box with the text "a way to specify the types of variables" points to the line `fixes c :: "real"`.
- A blue box with the text "induction on the finite set, A" points to the line `apply (induct A rule: finite_induct)`.

The status bar at the bottom shows the current goal:

```
-u-:%%- *goals* Top L1 (Isar Proofstate Utoks Abbrev;)-----
proof (prove): step 0
goal (1 subgoal):
1. finite A  $\Rightarrow$  ( $\sum_{x \in A} c * f x$ ) = c * setsum f A
tool-bar undo
```

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

```
Examples.thy
--u-:***- Examples.thy 5% L17 (Isar Utoks Abbrev; Scripting )-----
lemma
  fixes c :: "real"
  shows "finite A  $\implies$  setsum (%x. c * f x) A = c * setsum f A"
  apply (induct A rule: finite_induct)
  apply auto
  apply (auto simp add: algebra_simps)
  done

--u-:***- Examples.thy 5% L17 (Isar Utoks Abbrev; Scripting )-----
proof (prove): step 1
goal (2 subgoals):
1.  $(\sum_{x \in \{ \}}. c * f x) = c * \text{setsum } f \{ \}$ 
2.  $\wedge x F. [\text{finite } F; x \notin F; (\sum_{x \in F}. c * f x) = c * \text{setsum } f F]$ 
 $\implies (\sum_{x \in \text{insert } x F}. c * f x) = c * \text{setsum } f (\text{insert } x F)$ 
--u-:%%- *goals* Top L1 (Isar Proofstate Utoks Abbrev;)-----
tool-bar next
```

base case: A is empty

inductive step: A = insert x F

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  $\text{setsum } f (\text{insert } a F) = f a + \text{setsum } f F$

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

# Almost There!

```
Examples.thy

lemma
  fixes c :: "real"
  shows "finite A  $\implies$  setsum (%x. c * f x) A = c * setsum f A"
apply (induct A rule: finite_induct)
apply auto
- apply (auto simp add: algebra_simps)
done

-u-:**- Examples.thy 5% L18 (Isar Utoks Abbrev; Scripting )-----

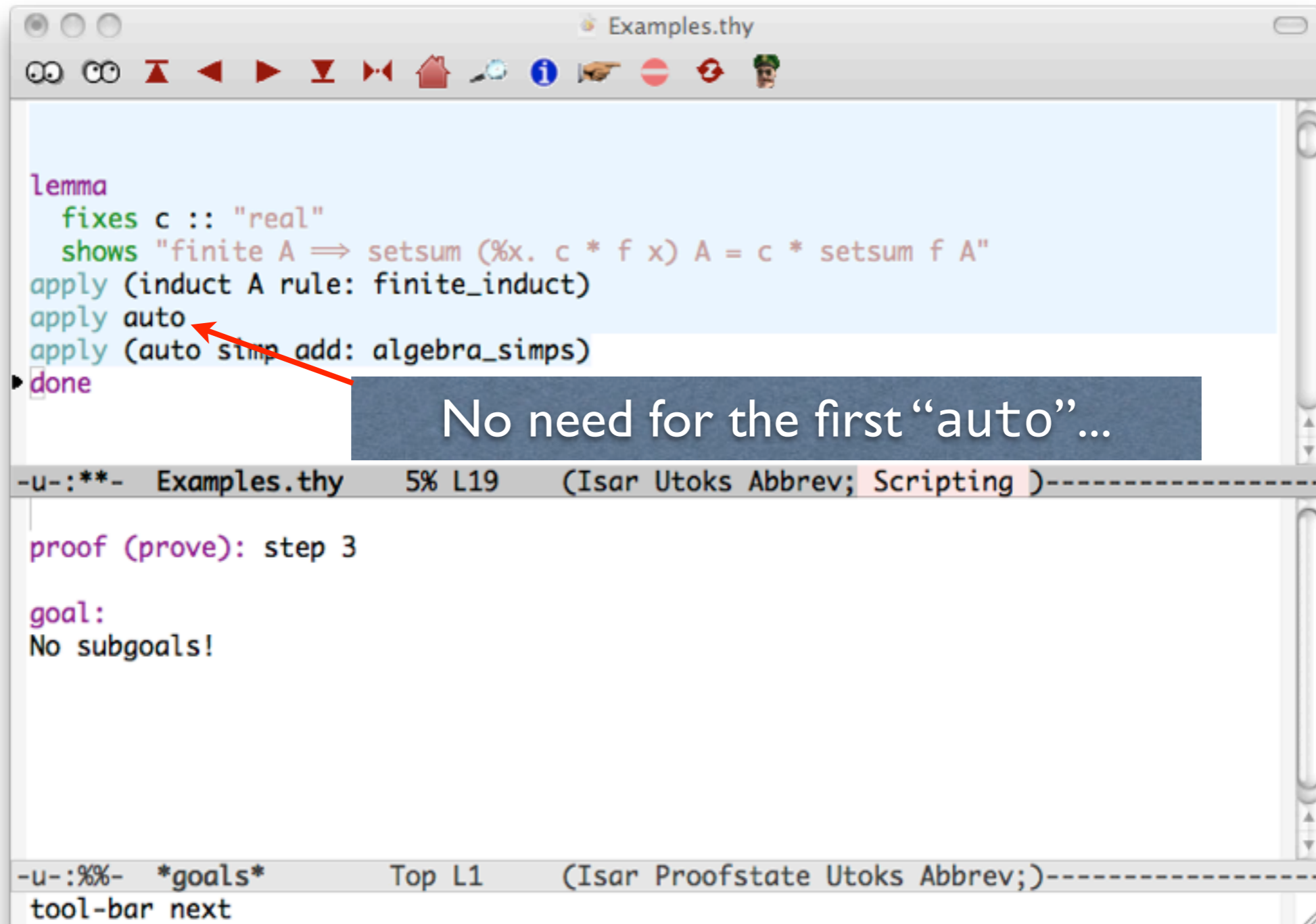
proof (prove): step 2

goal (1 subgoal):
1.  $\forall x \in F. [\text{finite } F; x \notin F; (\sum_{x \in F}. c * f x) = c * \text{setsum } f F]$ 
 $\implies c * f x + c * \text{setsum } f F = c * (f x + \text{setsum } f F)$ 

need to apply a distributive law

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

# Finished!



```
Examples.thy  
lemma  
  fixes c :: "real"  
  shows "finite A  $\implies$  setsum (%x. c * f x) A = c * setsum f A"  
  apply (induct A rule: finite_induct)  
  apply auto  
  apply (auto simp add: algebra_simps)  
  done  
proof (prove): step 3  
goal:  
No subgoals!  
*goals*  
Top L1 (Isar Proofstate Utoks Abbrev;)  
tool-bar next
```

No need for the first "auto"...

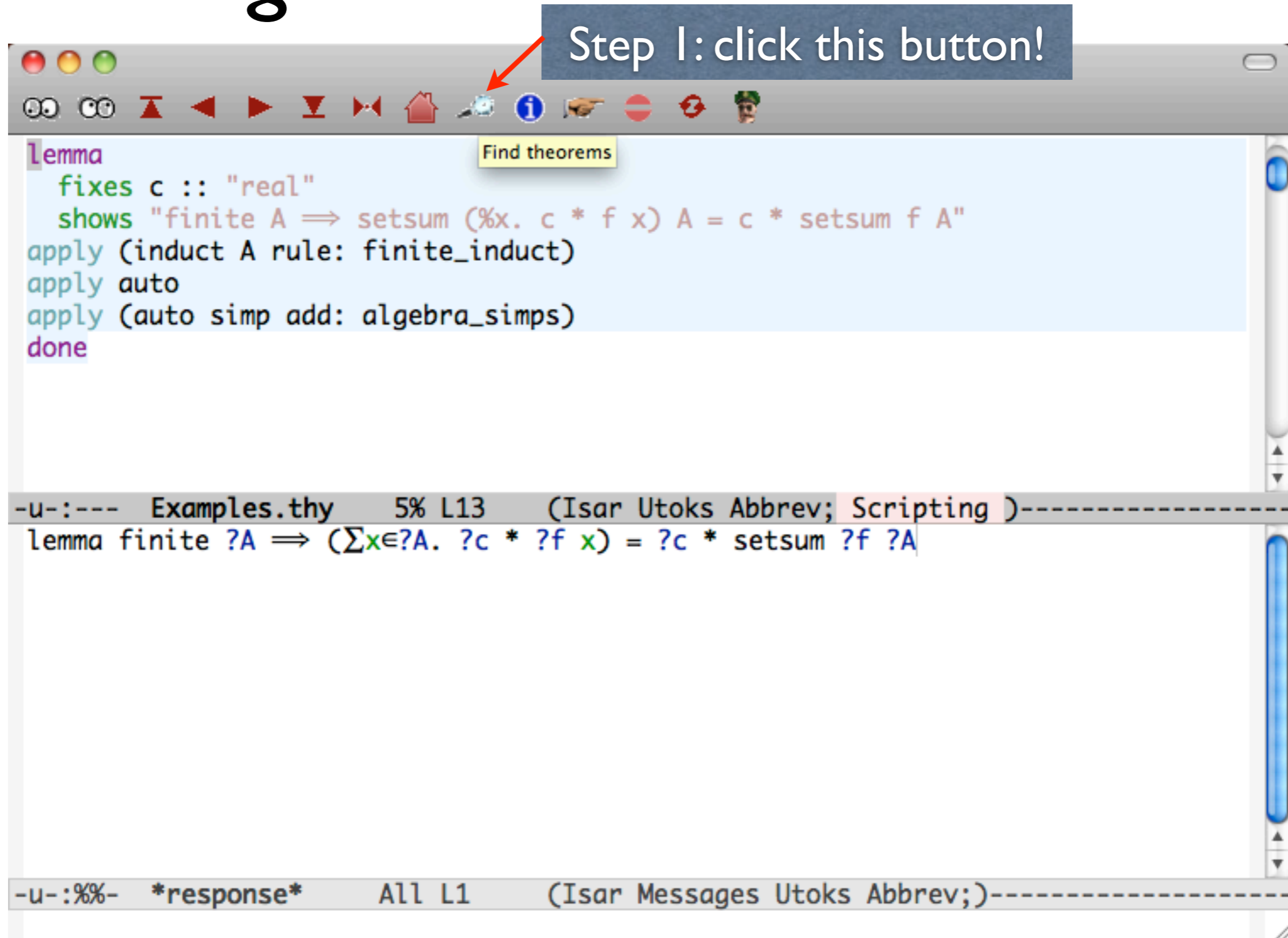
Recall that `algebra_simps` is a list of simplification rules for multiplying out algebraic expressions.

# 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

Step 1: click this button!



The screenshot shows the Isabelle/Isar IDE interface. At the top, a toolbar contains various navigation icons. A red arrow points to the 'Find theorems' button, which is highlighted with a yellow tooltip. Below the toolbar, a code editor displays a lemma definition:

```
lemma
  fixes c :: "real"
  shows "finite A  $\implies$  setsum (%x. c * f x) A = c * setsum f A"
apply (induct A rule: finite_induct)
apply auto
apply (auto simp add: algebra_simps)
done
```

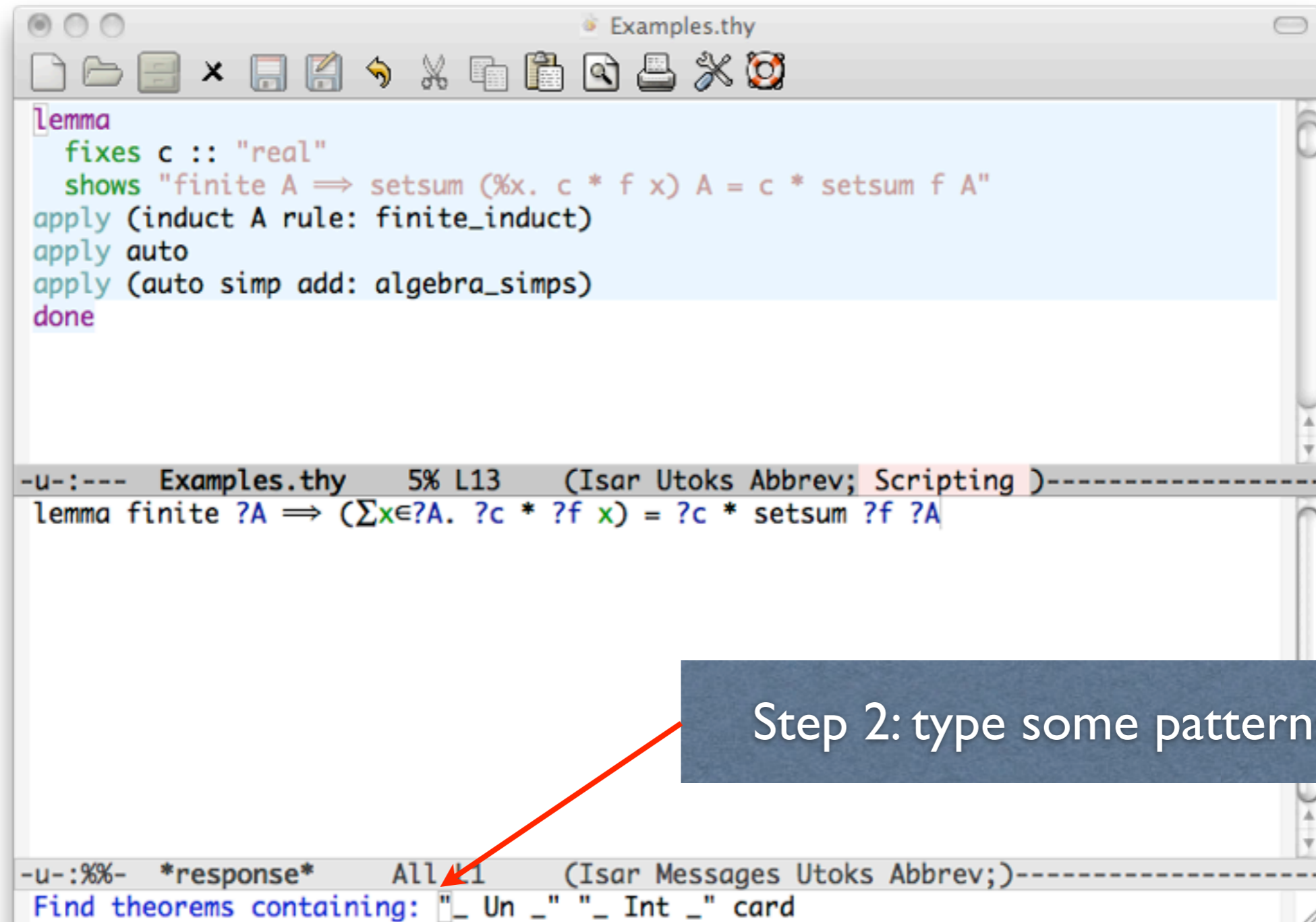
Below the code editor, a status bar shows the current file and line: `-u-:--- Examples.thy 5% L13 (Isar Utoks Abbrev; Scripting )`. The main text area displays the lemma statement in a different format:

```
lemma finite ?A  $\implies$  ( $\sum_{x \in ?A} ?c * ?f x$ ) = ?c * setsum ?f ?A
```

At the bottom, another status bar shows: `-u-:%%- *response* All L1 (Isar Messages Utoks Abbrev;)`.

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

```
lemma
  fixes c :: "real"
  shows "finite A  $\implies$  setsum (%x. c * f x) A = c * setsum f A"
apply (induct A rule: finite_induct)
apply auto
apply (auto simp add: algebra_simps)
done
```

Below the code is a status bar with the text: "-u-:--- Examples.thy 5% L13 (Isar Utoks Abbrev; Scripting )-----".

The search results area shows the lemma: "lemma finite ?A  $\implies$  ( $\sum_{x \in ?A} ?c * ?f x$ ) = ?c \* setsum ?f ?A".

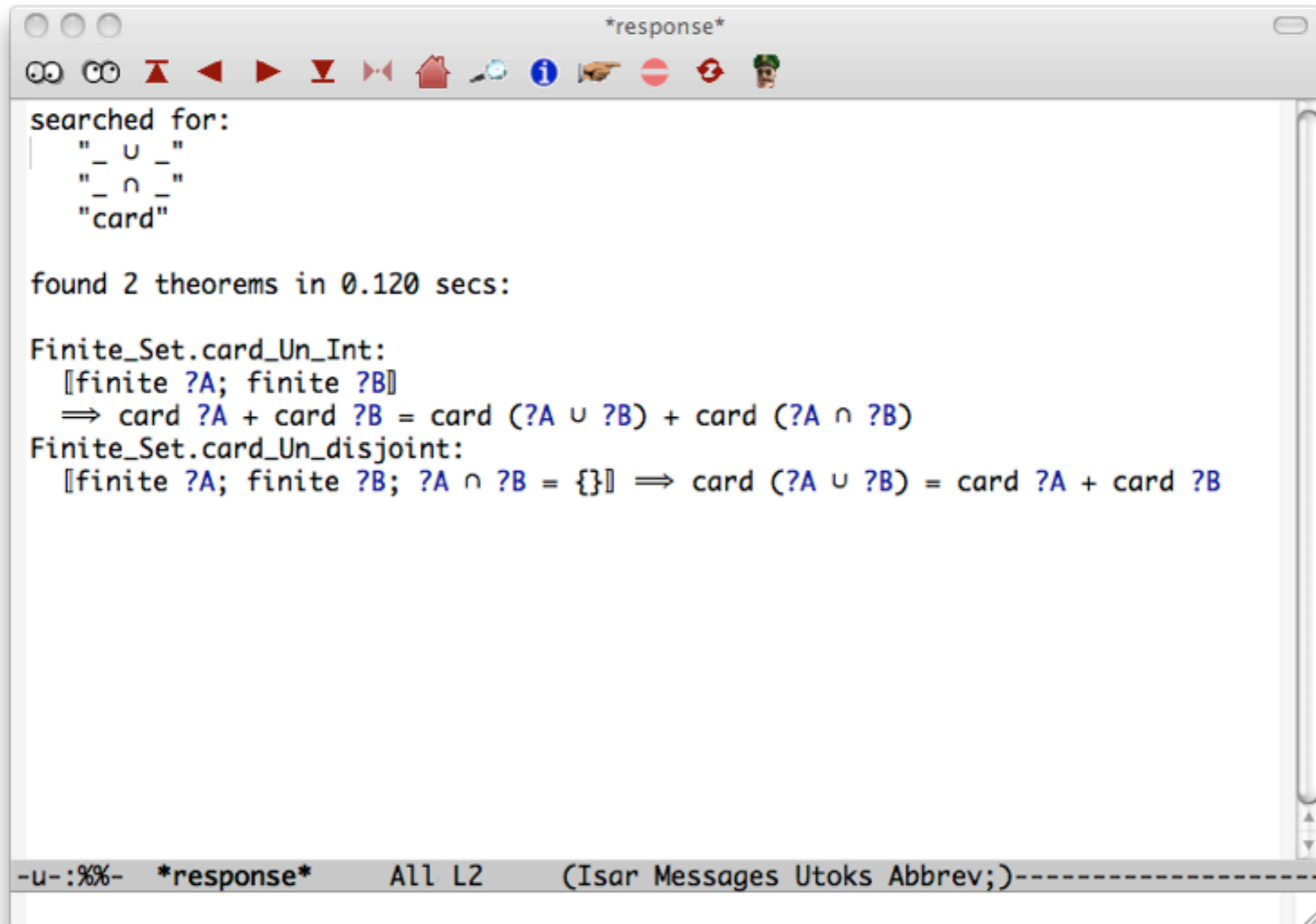
At the bottom, there is a search prompt: "-u-:%%- \*response\* All 11 (Isar Messages Utoks Abbrev;)-----" followed by "Find theorems containing: "\_ Un \_" "\_ Int \_" card".

A red arrow points from a blue box containing the text "Step 2: type some patterns" to the search prompt.

Step 2: type some patterns

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.

# Which Theorems Were Found?



```
*response*
searched for:
  "_ u _"
  "_ n _"
  "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

## 8: Inductive Definitions

Lawrence C Paulson  
Computer Laboratory  
University of Cambridge

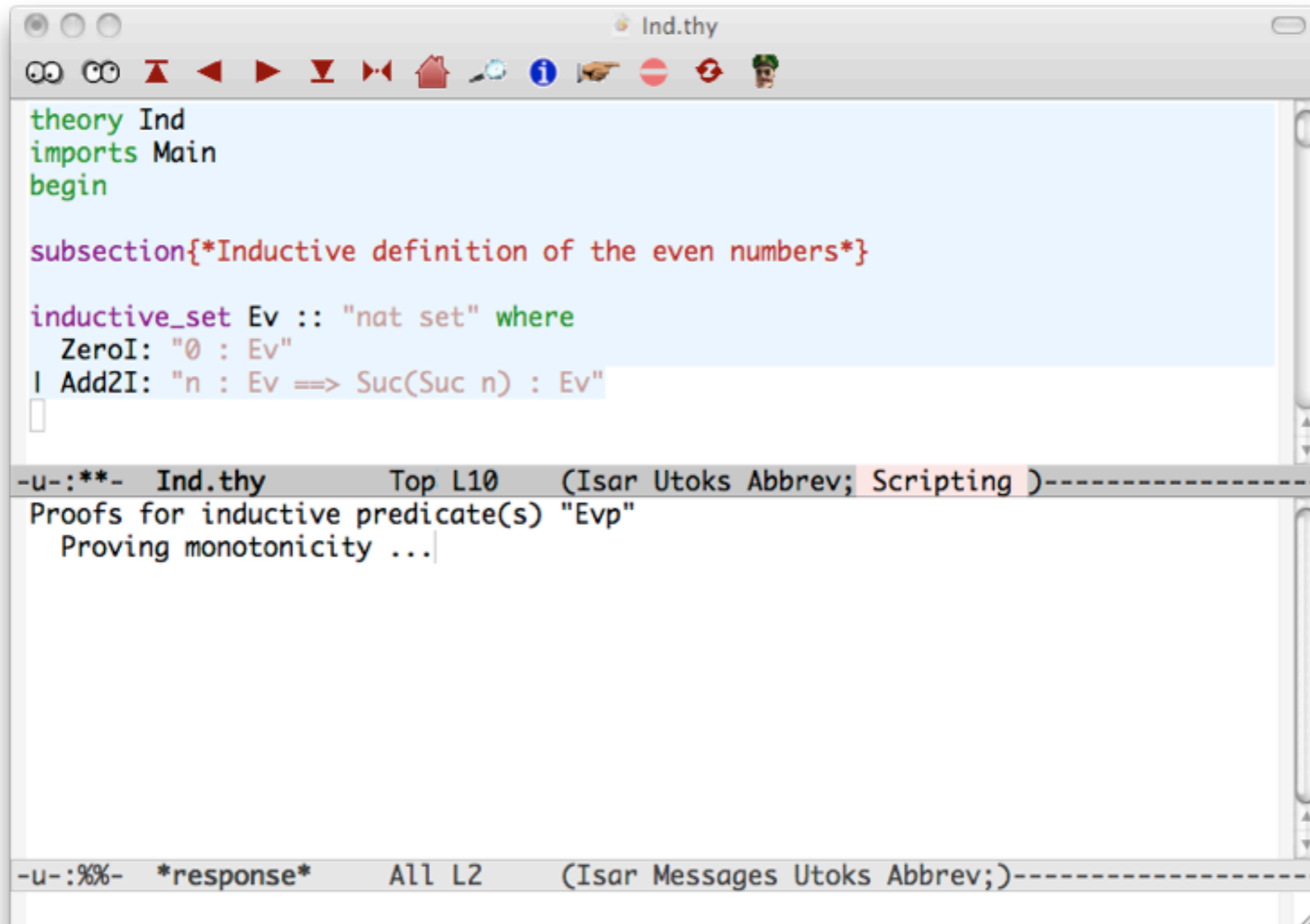
# Overview

- An introduction to inductive definitions
- A demonstration of their use in reasoning about finite sets.
- New forms of proof automation: the `arith` proof method and the *sledgehammer* tool.

# 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.

# Inductive Definitions in Isabelle



```
theory Ind
imports Main
begin

subsection{*Inductive definition of the even numbers*}

inductive_set Ev :: "nat set" where
  ZeroI: "0 : Ev"
| Add2I: "n : Ev ==> Suc(Suc n) : Ev"
[]

-u-:**- Ind.thy      Top L10      (Isar Utoks Abbrev; Scripting )-----
Proofs for inductive predicate(s) "Ev"
  Proving monotonicity ...|

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

# Even Numbers Belong to Ev

The screenshot shows a theorem prover interface with a file named 'Ind.thy'. The main editor contains the following code:

```
text{*All even numbers belong to this set.*}  
lemma "2*k : Ev"  
  apply (induct k)  
  apply auto  
  apply (auto simp add: ZeroI Add2I)  
  done
```

Below the code, the proof state is shown:

```
proof (prove): step 1  
goal (2 subgoals):  
1.  $2 * 0 \in \text{Ev}$   
2.  $\wedge k. 2 * k \in \text{Ev} \Rightarrow 2 * \text{Suc } k \in \text{Ev}$ 
```

Two red arrows point from a text box to the 'apply (induct k)' line in the code and the first subgoal in the proof state. The text box contains the text: 'ordinary induction yields two subgoals'.

# Proving Set Membership

The screenshot shows a theorem prover interface with a proof script in the top pane and the resulting goals in the bottom pane. A blue highlight covers the first line of the script. A dark blue callout box with white text and two red arrows points to the script and the second goal.

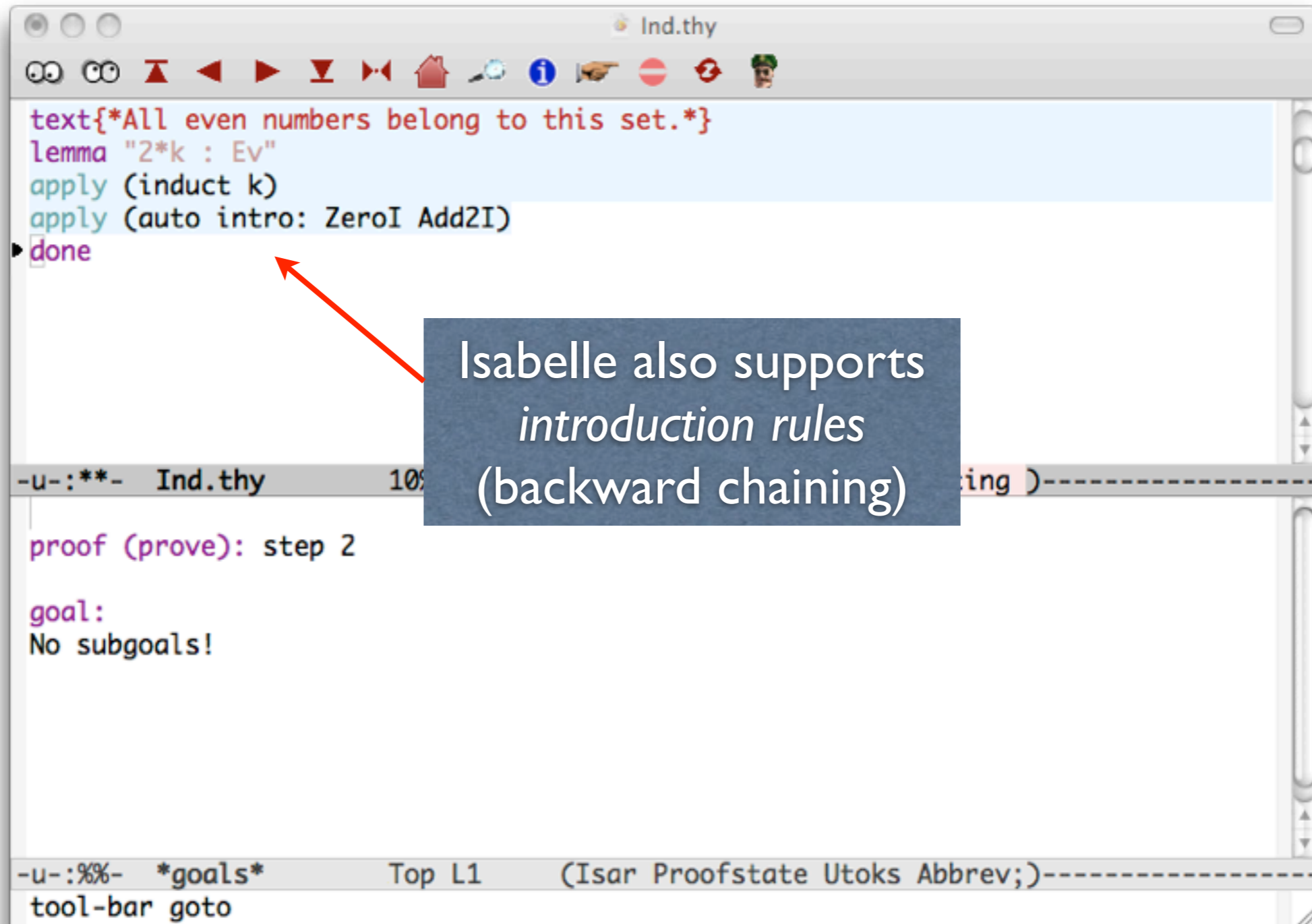
```
text{*All even numbers belong to this set.*}
lemma "2*k : Ev"
apply (induct k)
apply auto
apply (auto simp add: ZeroI Add2I)
done
```

after simplification, the subgoals resemble the introduction rules

```
proof (prove): step 2
goal (2 subgoals):
1. 0 ∈ Ev
2.  $\wedge k. 2 * k \in \text{Ev} \Rightarrow \text{Suc} (\text{Suc} (2 * k)) \in \text{Ev}$ 
```

tool-bar next

# Finishing the Proof



The screenshot shows the Isabelle proof editor interface. The main window displays the following code:

```
text{*All even numbers belong to this set.*}  
lemma "2*k : Ev"  
  apply (induct k)  
  apply (auto intro: ZeroI Add2I)  
  done
```

A red arrow points from a text box to the `done` keyword. The text box contains the text: "Isabelle also supports *introduction rules* (backward chaining)".

Below the main code, the proof state is shown:

```
proof (prove): step 2  
goal:  
No subgoals!
```

The bottom status bar shows: `-u-:%%- *goals* Top L1 (Isar Proofstate Utoks Abbrev;)-----` and `tool-bar goto`.

# 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

```
text{*All elements of this set are even.*}
lemma "n ∈ Ev ⇒ ∃k. n = 2*k"
  apply (induct n rule: Ev.induct)
  apply auto
  apply arith
  done

□
```

rule induction is needed!

naming the induction rule

```
-u-:**- Ind.thy 13% L43 (Isar Utoks Abbrev; Scripting )-----
|
| proof (prove): step 0
| goal (1 subgoal):
| 1. n ∈ Ev ⇒ ∃k. n = 2 * k
|
|-----
```

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

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

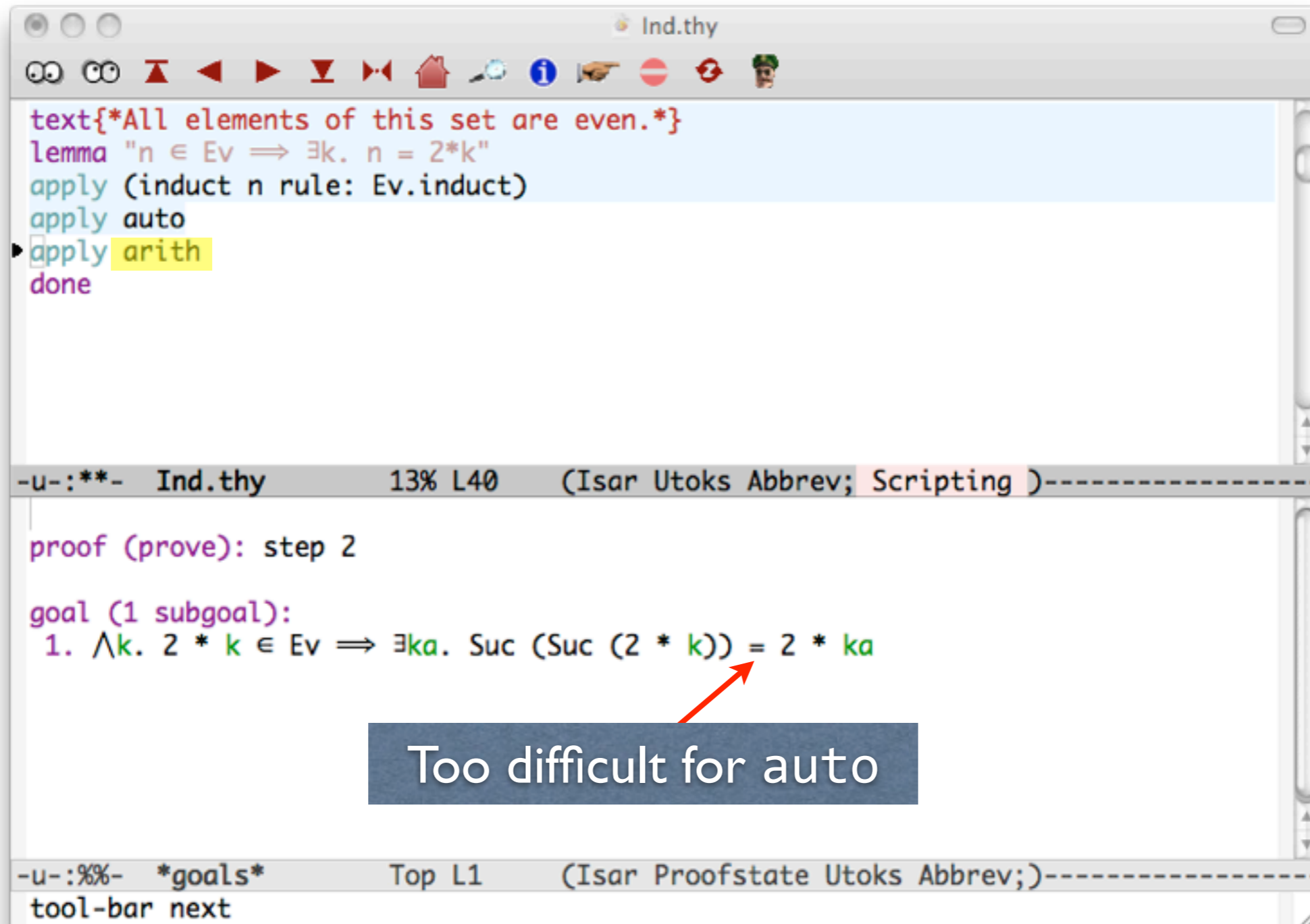
```
Ind.thy
text{*All elements of this set are even.*}
lemma "n ∈ Ev ⇒ ∃k. n = 2*k"
apply (induct n rule: Ev.induct)
apply auto
apply arith
done

-u-:**- Ind.thy 13% L39 (Isar Utoks Abbrev; Scripting )-----
proof (prove): step 1
goal (2 subgoals):
1. ∃k. 0 = 2 * k
2. ∧n. [n ∈ Ev; ∃k. n = 2 * k] ⇒ ∃k. Suc (Suc n) = 2 * k
-u-:%%-
tool-bar next
```

base case: n replaced by 0

induction step: n replaced by Suc (Suc n)

# One Tricky Goal Left!

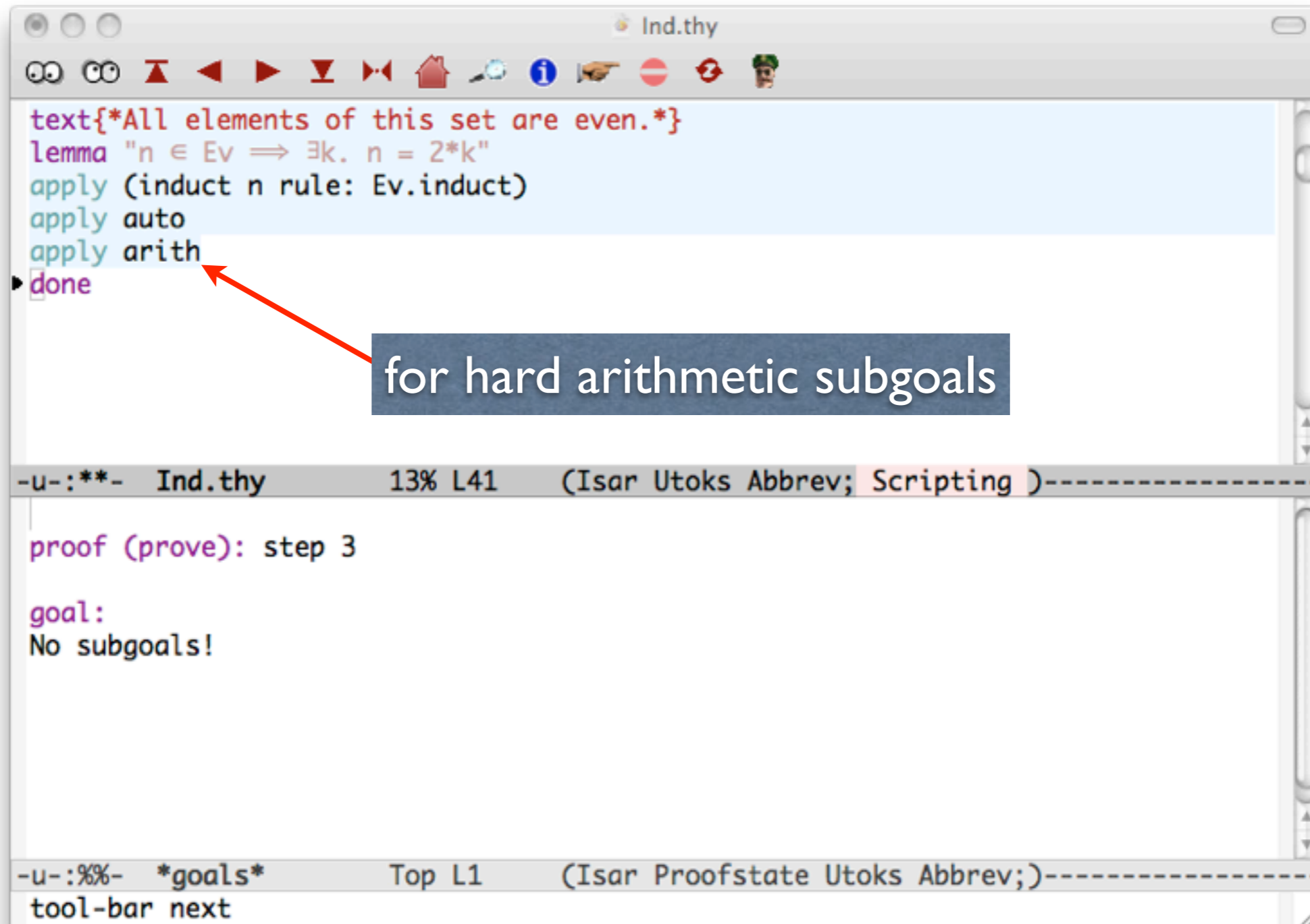


```
Ind.thy
text{*All elements of this set are even.*}
lemma "n ∈ Ev ⇒ ∃k. n = 2*k"
apply (induct n rule: Ev.induct)
apply auto
apply arith
done

-u-:**- Ind.thy 13% L40 (Isar Utoks Abbrev; Scripting )-----
proof (prove): step 2
goal (1 subgoal):
1.  $\wedge k. 2 * k \in \text{Ev} \Rightarrow \exists ka. \text{Suc} (\text{Suc} (2 * k)) = 2 * ka$ 
Too difficult for auto
-u-:%%- *goals* Top L1 (Isar Proofstate Utoks Abbrev;)-----
tool-bar next
```

Too difficult for auto

# The arith Proof Method



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

```
text{*All elements of this set are even.*}  
lemma "n ∈ Ev ⇒ ∃k. n = 2*k"  
apply (induct n rule: Ev.induct)  
apply auto  
apply arith  
done
```

A red arrow points from a dark blue callout box containing the text "for hard arithmetic subgoals" to the `arith` command in the script. Below the code, a status bar shows the current position: "-u-:\*\*- Ind.thy 13% L41 (Isar Utoks Abbrev; Scripting )".

The bottom section of the window shows the execution state:

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

At the very bottom, another status bar shows: "-u-:%%- \*goals\* Top L1 (Isar Proofstate Utoks Abbrev;)" and a "tool-bar next" button.

# 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"

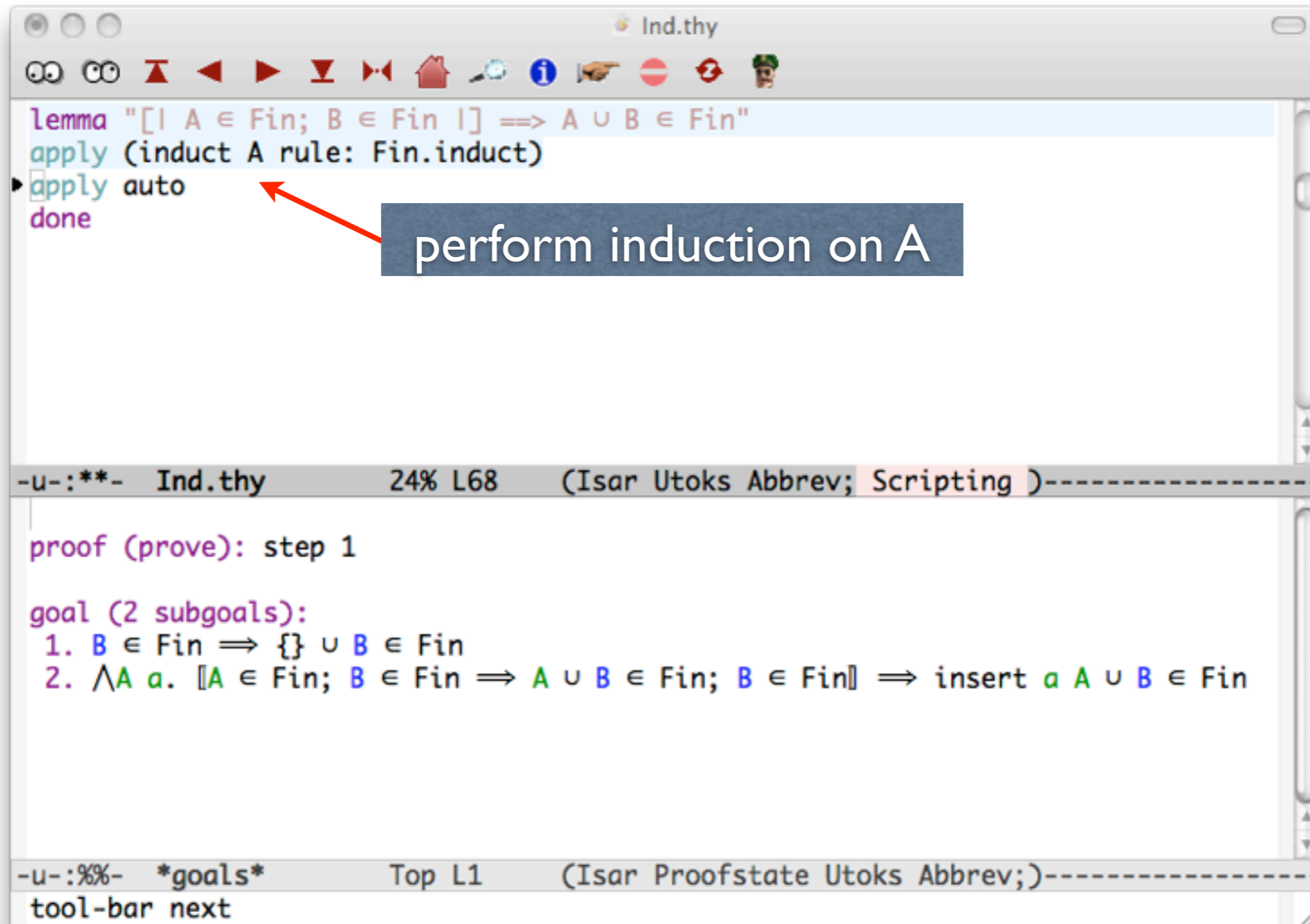
declare Fin.intros [intro]

```

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 screenshot shows a theorem prover interface with a toolbar at the top. The main window displays the following code:

```
lemma "[| A ∈ Fin; B ∈ Fin |] ==> A ∪ B ∈ Fin"  
  apply (induct A rule: Fin.induct)  
  apply auto  
done
```

An annotation box with the text "perform induction on A" has a red arrow pointing to the `apply (induct A rule: Fin.induct)` line.

The bottom panel shows the proof state:

```
proof (prove): step 1  
goal (2 subgoals):  
1. B ∈ Fin ==> {} ∪ B ∈ Fin  
2. ∧A a. [A ∈ Fin; B ∈ Fin ==> A ∪ B ∈ Fin; B ∈ Fin] ==> insert a A ∪ B ∈ Fin
```

The bottom status bar indicates the current goal and tool-bar options:

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

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

# A Subset of a Finite Set

The screenshot shows a theorem prover interface with a file named 'Ind.thy'. The main window contains the following code:

```
lemma "[| A ∈ Fin; B ⊆ A |] ==> B ∈ Fin"
  apply (induct A arbitrary: B rule: Fin.induct)
  apply auto
```

A red arrow points from a blue callout box to the lemma statement. The callout box contains the text: "to prove that every subset of A is finite".

Below the main window, a status bar shows the file name 'Ind.thy', progress '27% L79', and a list of loaded files: '(Isar Utoks Abbrev; Scripting )'. Below this is a second window showing the proof state:

```
proof (prove): step 1
  goal (2 subgoals):
  1.  $\bigwedge B. B \subseteq \{\}$   $\Rightarrow B \in \text{Fin}$ 
  2.  $\bigwedge A a B. [A \in \text{Fin}; \bigwedge B. B \subseteq A \Rightarrow B \in \text{Fin}; B \subseteq \text{insert } a A] \Rightarrow B \in \text{Fin}$ 
```

A red arrow points from a blue callout box to the first subgoal. The callout box contains the text: "as seen in the induction hypothesis".

The bottom status bar shows the file name '\*goals\*', progress 'Top L1', and a list of loaded files: '(Isar Proofstate Utoks Abbrev;)'. Below this is a toolbar with the text 'tool-bar next'.

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

# A Critical Point in the Proof

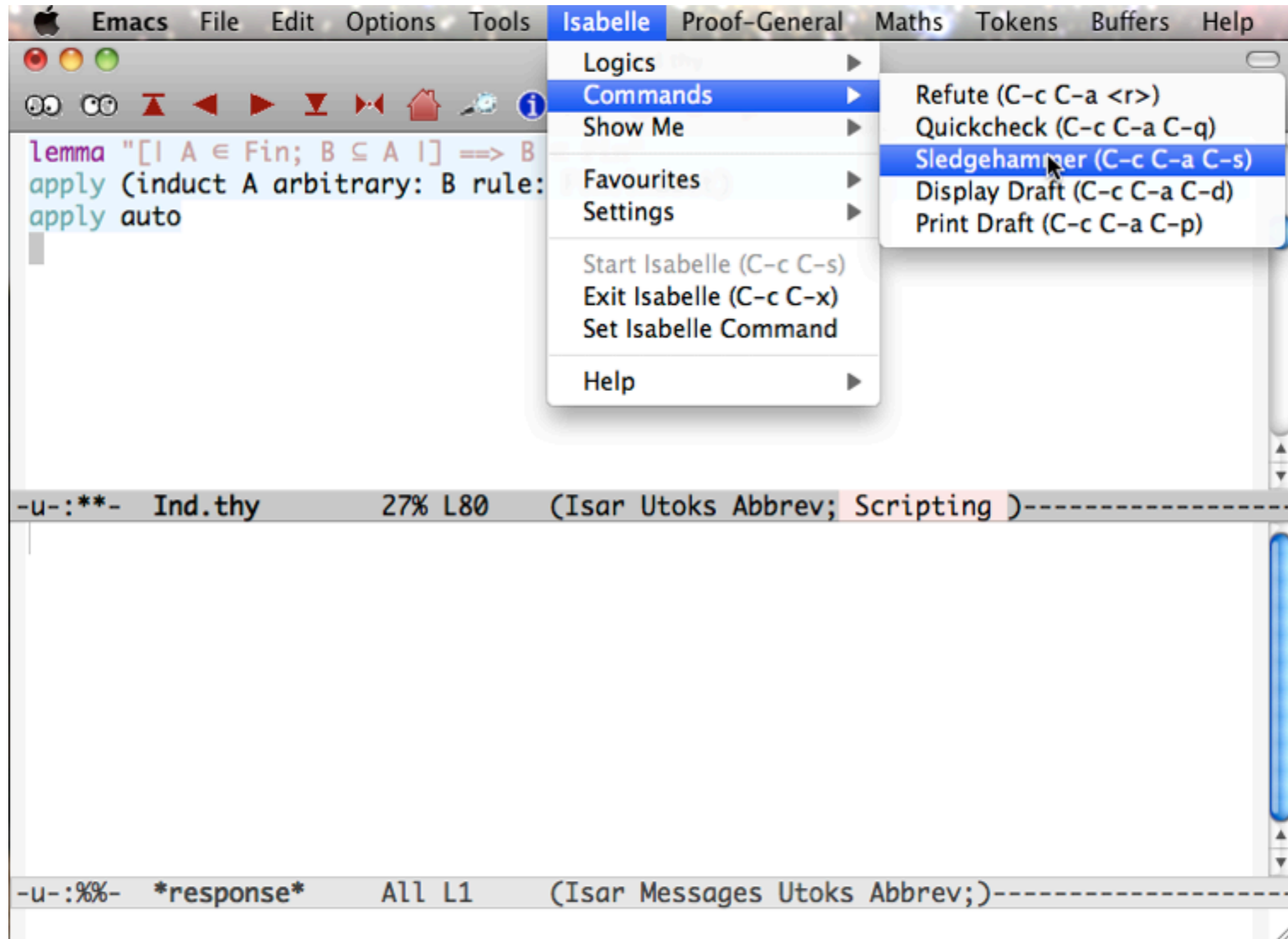
```
Ind.thy
lemma "[| A ∈ Fin; B ⊆ A |] ⇒ B ∈ Fin"
  apply (induct A arbitrary: B rule: Fin.induct)
  apply auto
  .
-u-:**- Ind.thy 27% L80 (Isar Utoks Abbrev; Scripting )-----
proof (prove): step 2
goal (1 subgoal):
  1.  $\wedge A a B. [\wedge B. B \subseteq A \Rightarrow B \in \text{Fin}; B \subseteq \text{insert } a A] \Rightarrow B \in \text{Fin}$ 
now what??
-u-:%%- *goals* Top L1 (Isar Proofstate Utoks Abbrev;)-----
tool-bar next
```

now what??

None of Isabelle's automatic proof methods (auto, blast, force) have any effect on this subgoal. Informally, we might consider case analysis on whether  $a \in B$ . 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!

The screenshot shows the Isabelle/Proof General interface. At the top, a window titled "Ind.thy" contains a proof script:

```
lemma "[| A ∈ Fin; B ⊆ A |] ⇒ B ∈ Fin"  
apply (induct A arbitrary: B rule: Fin.induct)  
apply auto  
□
```

Below the script, the output window shows the execution of the proof. The output is divided into two sections, both highlighted in yellow:

The first section shows the goal and a suggestion for a command:

```
-u-:**- Ind.thy 27% L80 (Isar Utoks Abbrev;  
Sledgehammer: external prover "spass" for subgoal 1  
∧ A a B. [[∧ B. B ⊆ A ⇒ B ∈ Fin; B ⊆ insert a A] ⇒ B ∈ Fin  
Try this command: apply (metis Fin.insertI Int_absorb1 Int_commute Int_insert_right Int_lower1 mem_def subset_insert)
```

The second section shows a suggestion for a more compact command:

```
For minimizing the number of lemmas try this command:  
atp_minimize [atp=spass] Fin.insertI Int_absorb1 Int_commute Int_insert_right Int_lower1 mem_def subset_insert
```

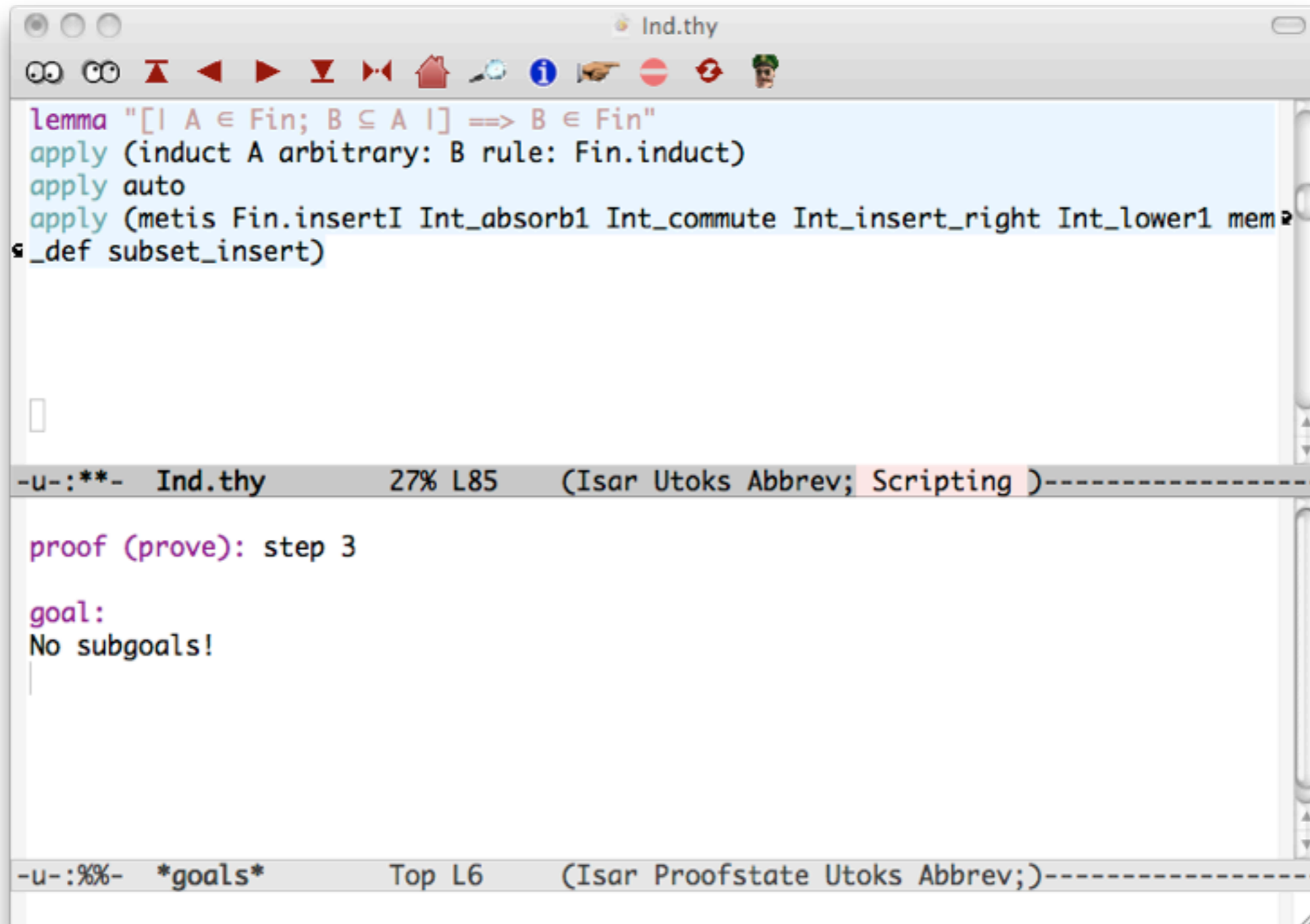
Two blue callout boxes with red arrows point to the highlighted sections:

- The first callout box says: "this command should prove the goal" (pointing to the `apply (metis ...)` command).
- The second callout box says: "this one may return a more compact command" (pointing to the `atp_minimize ...` command).

At the bottom of the interface, the status bar shows: "-u-:%%- \*response\* All L1 (Isar Messages Utoks Abbrev;)- menu-bar Isabelle Commands Sledgehammer".

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



The screenshot shows a proof assistant window titled "Ind.thy". The main editor contains the following code:

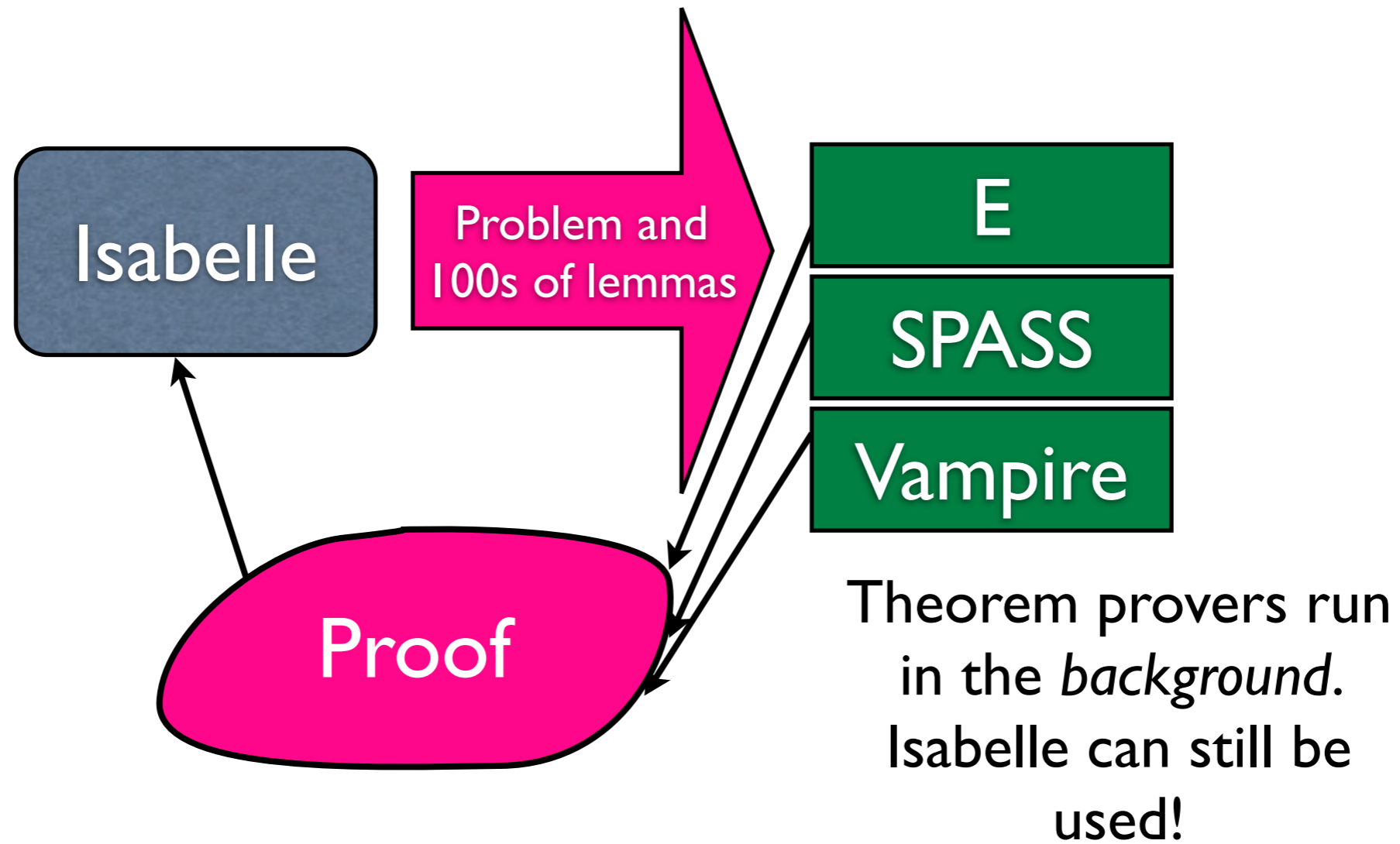
```
lemma "[| A ∈ Fin; B ⊆ A |] ==> B ∈ Fin"  
  apply (induct A arbitrary: B rule: Fin.induct)  
  apply auto  
  apply (metis Fin.insertI Int_absorb1 Int_commute Int_insert_right Int_lower1 mem  
_def subset_insert)
```

The status bar at the bottom of the editor shows: `-u-:**- Ind.thy 27% L85 (Isar Utoks Abbrev; Scripting )`. Below the editor, the goal pane shows:

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

The goal pane status bar shows: `-u-:%%- *goals* Top L6 (Isar Proofstate Utoks Abbrev;)`.

# How Sledgehammer Works



# Notes on Sledgehammer

- It is always available, but it cannot work miracles.
- It does not prove the goal, but returns a call to `metis`. This command *usually* works, but sometimes it runs too slowly to be of any use.
- The `minimise` option removes redundant theorems, increasing the likelihood of success.
- Calling `metis` directly is difficult unless you know exactly which lemmas are needed.

# Interactive Formal Verification

## 9: 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

```
typedec1 loc -- "an unspecified type of locations"
```

```
type_synonym val = nat -- "values"
```

```
type_synonym state = "loc => val"
```

```
type_synonym aexp = "state => val"
```

```
type_synonym bexp = "state => bool" -- "functions on states"
```

```
datatype
```

```
  com = SKIP
```

```
    | Assign loc aexp
```

```
    | Semi    com com
```

```
    | Cond    bexp com com
```

```
    | While   bexp com
```

Arithmetic & boolean expressions  
are *functions* over the state

```
( "_ ::= _ " 60 )
```

```
( "_; _" [60, 60] 10 )
```

```
( "IF _ THEN _ ELSE _" 60 )
```

```
( "WHILE _ DO _" 60 )
```

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.



# A “Big-Step” Semantics

$$\langle \mathbf{skip}, s \rangle \rightarrow s$$

$$\langle x := a, s \rangle \rightarrow s[x := a \ s]$$

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

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

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

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

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

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

```
text {* The big-step execution relation @{text evalc} is defined inductively *}
inductive
  evalc :: "[com,state,state]  $\Rightarrow$  bool" ("<_,_>/  $\rightsquigarrow$  _" [0,0,60] 60)
where
  Skip:    "<SKIP,s>  $\rightsquigarrow$  s"
| Assign: "<x := a,s>  $\rightsquigarrow$  s(x := a s)"
| Semi:   "<c0,s>  $\rightsquigarrow$  s'  $\Rightarrow$  <c1,s'>  $\rightsquigarrow$  s'  $\Rightarrow$  <c0; c1, s>  $\rightsquigarrow$  s'"
| IfTrue: "b s  $\Rightarrow$  <c0,s>  $\rightsquigarrow$  s'  $\Rightarrow$  <IF b THEN c0 ELSE c1, s>  $\rightsquigarrow$  s'"
| IfFalse: "~b s  $\Rightarrow$  <c1,s>  $\rightsquigarrow$  s'  $\Rightarrow$  <IF b THEN c0 ELSE c1, s>  $\rightsquigarrow$  s'"
| WhileFalse: "~b s  $\Rightarrow$  <WHILE b DO c,s>  $\rightsquigarrow$  s"
| WhileTrue: "b s  $\Rightarrow$  <c,s>  $\rightsquigarrow$  s'  $\Rightarrow$  <WHILE b DO c, s'>  $\rightsquigarrow$  s'
            $\Rightarrow$  <WHILE b DO c, s>  $\rightsquigarrow$  s'"

lemmas evalc.intros [intro] -- "use those rules in automatic proofs"
```

an inductive predicate with special syntax

declare as introduction rules for auto and blast

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 screenshot shows the Isabelle IDE interface with several annotations:

- name of the new lemma**: Points to the variable name `skipE` in the first line of the code.
- declared as an *elimination rule* to auto and blast**: Points to the `[elim]` tag in the first line.
- $\langle \text{skip}, s \rangle \rightarrow s'$  implies  $s = s'$** : Points to the pattern `"<SKIP, s> ~> s'"` in the first line.
- the typical format of an elimination rule**: Points to the theorem statement `[[<SKIP, ?s> ~> ?s'; ?s' = ?s ==> ?P]] ==> ?P` in the command line.

```
inductive_cases skipE [elim]: "<SKIP, s> ~> s'"
inductive_cases semiE [elim]: "<c0; c1, s> ~> s'"
inductive_cases assignE [elim]: "<x := a, s> ~> s'"
inductive_cases ifE [elim]: "<IF b THEN c0 ELSE c1, s> ~> s'"
inductive_cases whileE [elim]: "<WHILE b DO c, s> ~> s'"

-u-:--- Com.thy 53% 48 (Isar Utoks Abbrev; Scripting )
[[<SKIP, ?s> ~> ?s'; ?s' = ?s ==> ?P]] ==> ?P

-u-:%%- *response* All L1 (Isar Messages Utoks Abbrev;)
```

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

```
Com.thy
inductive_cases skipE [elim]: "<SKIP,s> ~> s'"
inductive_cases semiE [elim]: "<c0; c1, s> ~> s'"
inductive_cases assignE [elim]: "<x := a,s> ~> s'"
inductive_cases ifE [elim]: "<IF b THEN c0 ELSE c1, s> ~> s'"
inductive_cases whileE [elim]: "<WHILE b DO c,s> ~> s'"

-u-:--- Com.thy 53% L49 (Isar Utoks Abbrev; Scripting )-----
[[<c0.0; ?c1.0,?s> ~> ?s'; ^s'']. [[<c0.0,?s> ~> s'; <?c1.0,s'> ~> ?s']] ==> ?P]
==> ?P

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

expresses the existence of the *intermediate* state,  $s'$

# A Non-Termination Proof

$$\langle \text{while true do } c, s \rangle \not\rightarrow s'$$

This formula is not provable by induction!

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

The inductive version considers  
*all* possible commands

# Non-Termination in Isabelle

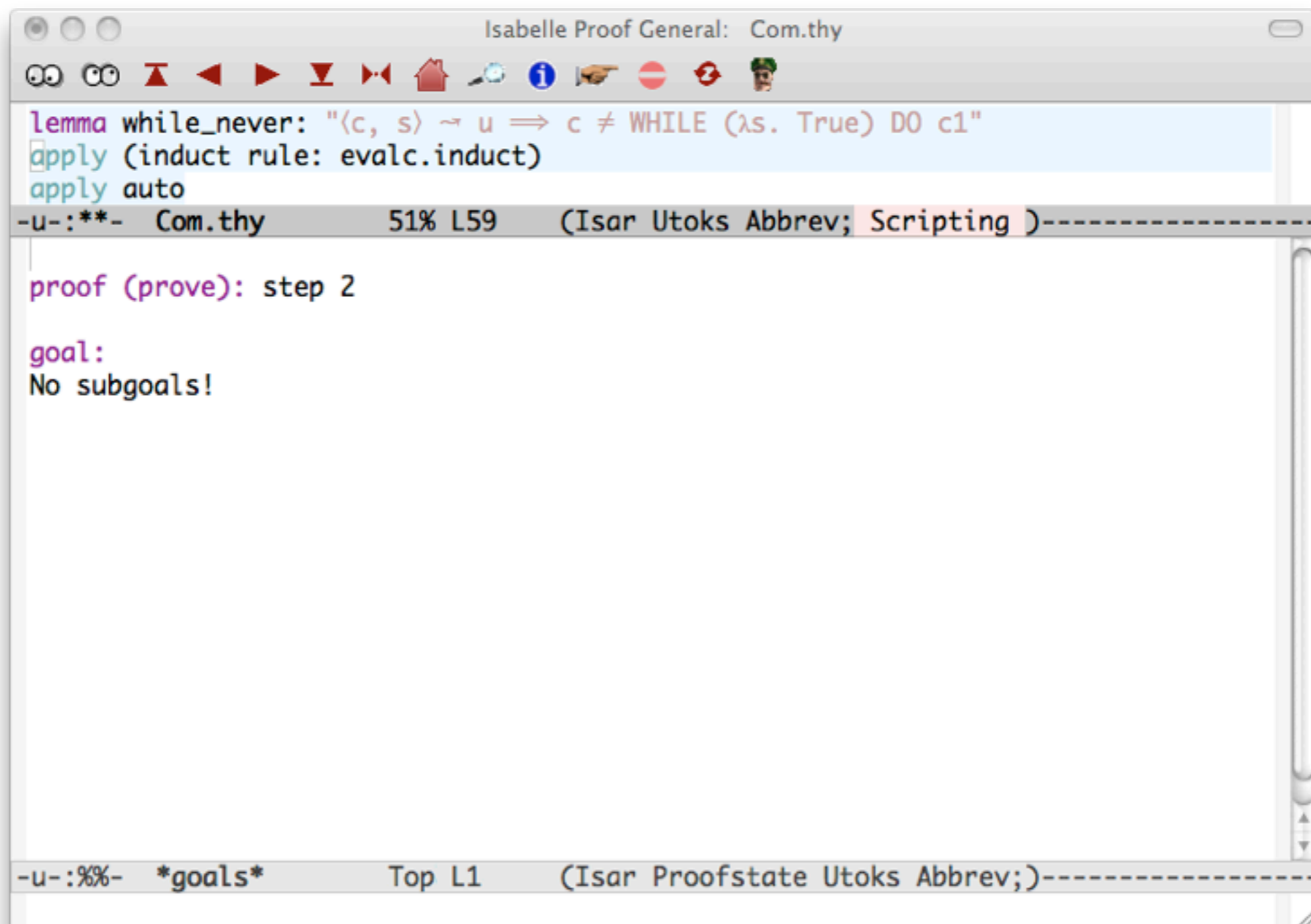
```
Isabelle Proof General: Com.thy
lemmas while_never: "<c, s> ~ u => c ≠ WHILE (λs. True) DO c1"
apply (induct rule: evalc.induct)
apply auto
-u-:***- Com.thy 51% L60 (Isar Utoks Abbrev; Scripting
goal (7 subgoals)
1. ∧s. SKIP ≠ WHILE λs. True DO c1
2. ∧x a s. x ::= a ≠ WHILE λs. True DO c1
3. ∧c0 s s' c1a s'.
   [[<c0,s> ~ s'; c0 ≠ WHILE λs. True DO c1; <c1a,s'> ~ s';
    c1a ≠ WHILE λs. True DO c1]]
   => (c0; c1a) ≠ WHILE λs. True DO c1
4. ∧b s c0 s' c1a.
   [[b s; <c0,s> ~ s'; c0 ≠ WHILE λs. True DO c1]]
   => IF b THEN c0 ELSE c1a ≠ WHILE λs. True DO c1
5. ∧b s c1a s' c0.
   [[¬ b s; <c1a,s> ~ s'; c1a ≠ WHILE λs. True DO c1]]
   => IF b THEN c0 ELSE c1a ≠ WHILE λs. True DO c1
6. ∧b s c. ¬ b s => WHILE b DO c ≠ WHILE λs. True DO c1
7. ∧b s c s' s'.
   [[b s; <c,s> ~ s'; c ≠ WHILE λs. True DO c1; <WHILE b DO c,s'> ~ s';
    WHILE b DO c ≠ WHILE λs. True DO c1]]
   => WHILE b DO c ≠ WHILE λs. True DO c1
-u-:%%- *goals* 2% L4 (Isar Proofstate Utoks Abbrev;)
```

7 subgoals, one for each rule of the definition

Most are trivial, by distinctness

trivial for another reason

# Done!



The screenshot shows the Isabelle Proof General interface. The window title is "Isabelle Proof General: Com.thy". The top toolbar contains various navigation icons. The main text area shows the following code:

```
lemma while_never: "<c, s> ~> u => c ≠ WHILE (λs. True) DO c1"  
apply (induct rule: evalc.induct)  
apply auto
```

The status bar at the top indicates the current file is "Com.thy" at line 59, column 159, with the text "(Isar Utoks Abbrev; Scripting )".

The main proof area shows:

```
proof (prove): step 2  
goal:  
No subgoals!
```

The status bar at the bottom indicates the current goal is "\*goals\*" at line 1, column 1, with the text "(Isar Proofstate Utoks Abbrev;)".

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



# Determinacy

$$\frac{\langle c, s \rangle \rightarrow t \quad \langle c, s \rangle \rightarrow u}{t = u}$$

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

# Determinacy in Isabelle...

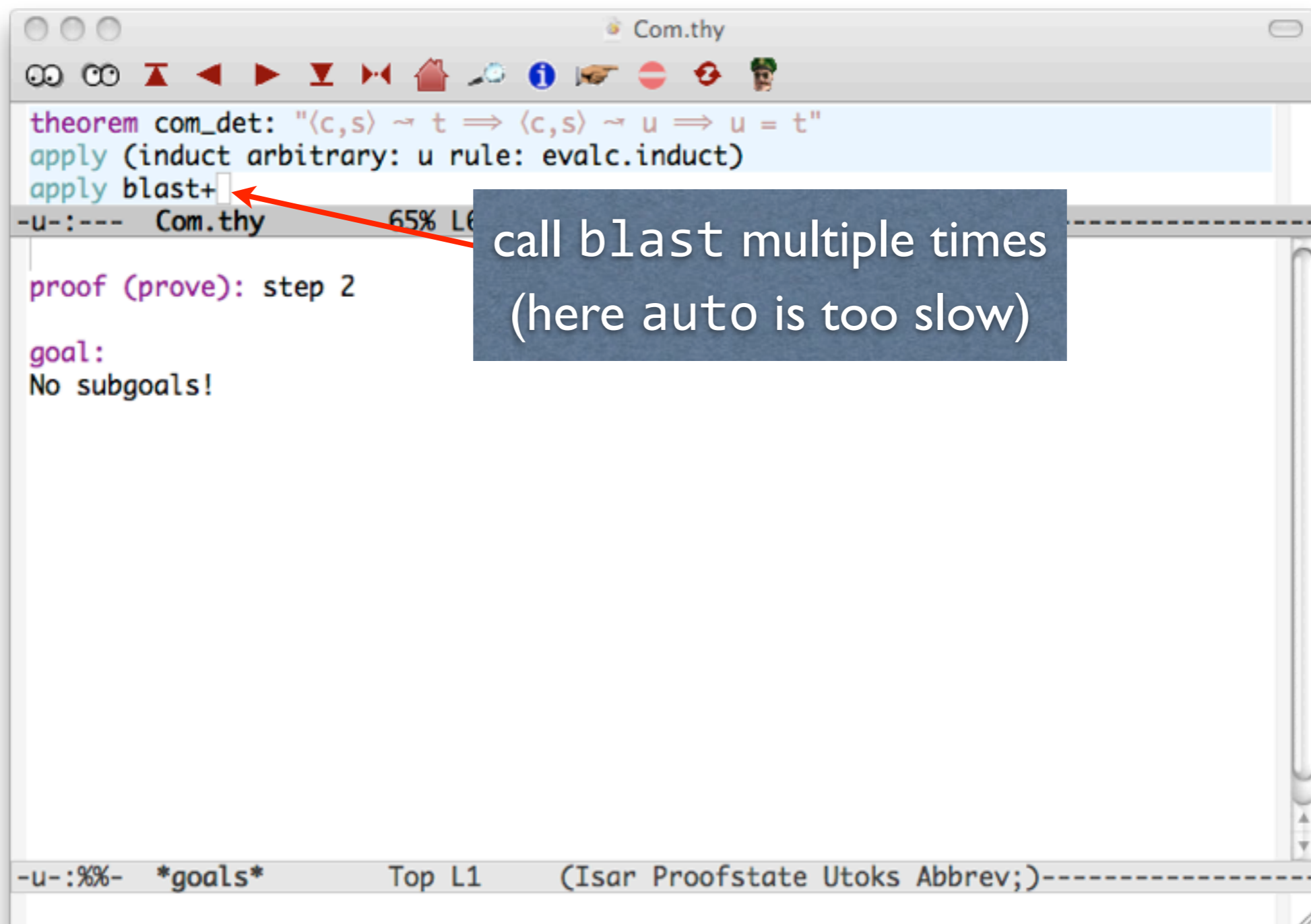
```
Com.thy
theorem com_det: "<c,s> ~ t => <c,s> ~ u => u = t"
apply (induct arbitrary: u rule: evalc.induct)
apply blast+
-u-:**- Com.thy 60% L62 (Isar Utoks Abbrev; Scripting )-----
1.  $\wedge s u. \langle \text{SKIP}, s \rangle \sim u \Rightarrow u = s$ 
2.  $\wedge x a s u. \langle x ::= a, s \rangle \sim u \Rightarrow u = s(x ::= a s)$ 
3.  $\wedge c_0 s s' c_1 s' u.$ 
    $\llbracket \langle c_0, s \rangle \sim s'; \wedge u. \langle c_0, s \rangle \sim u \Rightarrow u = s'; \langle c_1, s' \rangle \sim s';$ 
    $\wedge u. \langle c_1, s' \rangle \sim u \Rightarrow u = s'; \langle c_0; c_1, s \rangle \sim u \rrbracket$ 
    $\Rightarrow u = s'$ 
4.  $\wedge b s c_0 s' c_1 u.$ 
    $\llbracket b s; \langle c_0, s \rangle \sim s'; \wedge u. \langle c_0, s \rangle \sim u \Rightarrow u = s';$ 
    $\langle \text{IF } b \text{ THEN } c_0 \text{ ELSE } c_1, s \rangle \sim u \rrbracket$ 
    $\Rightarrow u = s'$ 
5.  $\wedge b s c_1 s' c_0 u.$ 
    $\llbracket \neg b s; \langle c_1, s \rangle \sim s'; \wedge u. \langle c_1, s \rangle \sim u \Rightarrow u = s';$ 
    $\langle \text{IF } b \text{ THEN } c_0 \text{ ELSE } c_1, s \rangle \sim u \rrbracket$ 
    $\Rightarrow u = s'$ 
6.  $\wedge b s c u. \llbracket \neg b s; \langle \text{WHILE } b \text{ DO } c, s \rangle \sim u \rrbracket \Rightarrow u = s$ 
7.  $\wedge b s c s' s' u.$ 
    $\llbracket b s; \langle c, s \rangle \sim s'; \wedge u. \langle c, s \rangle \sim u \Rightarrow u = s'; \langle \text{WHILE } b \text{ DO } c, s' \rangle \sim s';$ 
    $\wedge u. \langle \text{WHILE } b \text{ DO } c, s' \rangle \sim u \Rightarrow u = s'; \langle \text{WHILE } b \text{ DO } c, s \rangle \sim u \rrbracket$ 
    $\Rightarrow u = s'$ 
-u-:%%- *goals* 3% L5 (Isar Proofstate Utoks Abbrev;)-----
```

allow the other state to vary

trivial by rule inversion

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

```
theorem com_det: "<c,s> ~ t => <c,s> ~ u => u = t"  
apply (induct arbitrary: u rule: evalc.induct)  
apply blast+
```

A red arrow points from the text box to the "blast+" line. Below the code, the interface shows a proof step:

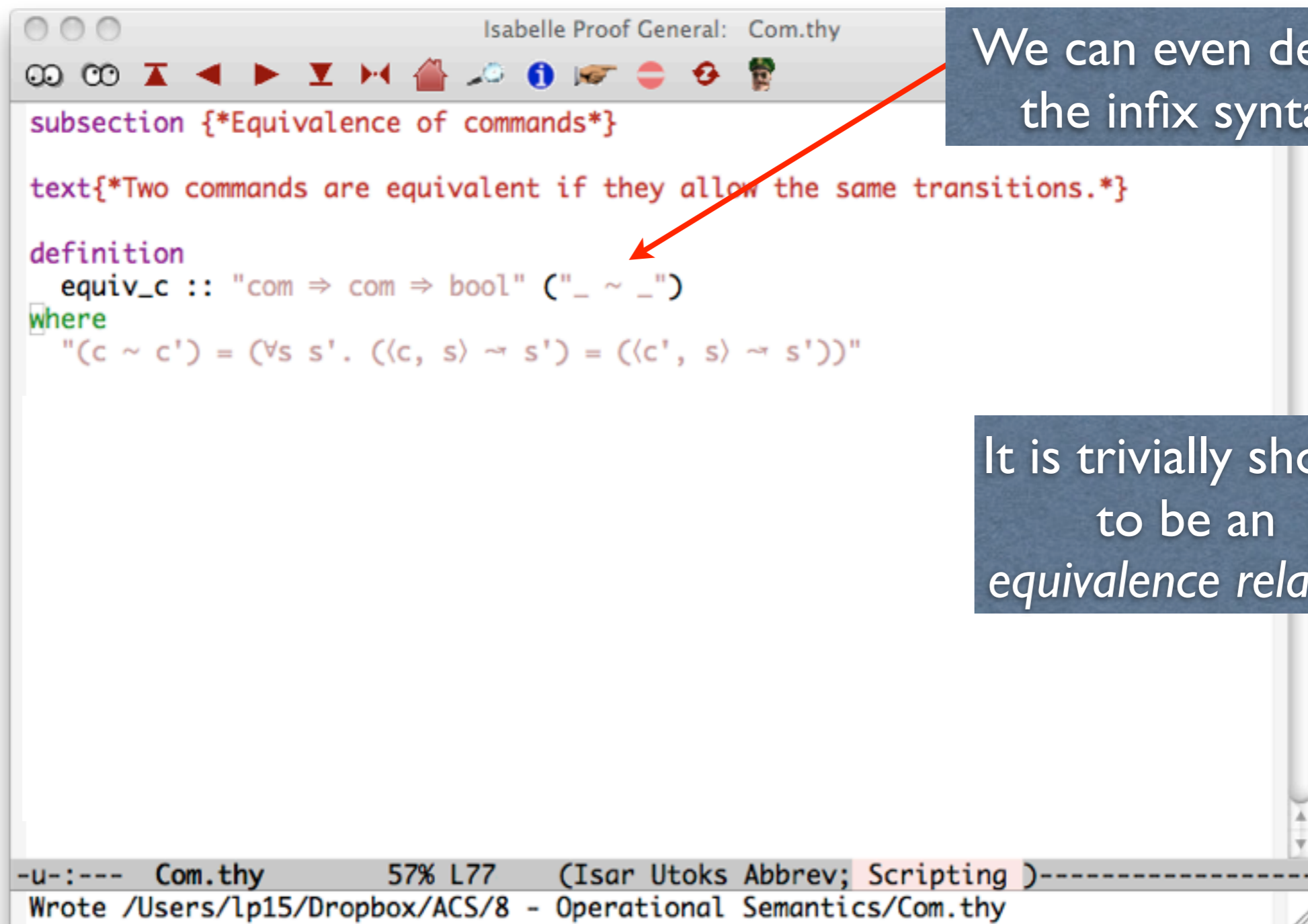
```
proof (prove): step 2  
goal:  
No subgoals!
```

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

call blast multiple times  
(here auto is too slow)

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



```
Isabelle Proof General: Com.thy
subsection {*Equivalence of commands*}
text{*Two commands are equivalent if they allow the same transitions.*}
definition
  equiv_c :: "com  $\Rightarrow$  com  $\Rightarrow$  bool" ("_  $\sim$  _")
where
  "(c  $\sim$  c') = ( $\forall$ s s'. ((c, s)  $\rightarrow$  s') = ((c', s)  $\rightarrow$  s'))"
```

-u-:--- Com.thy 57% L77 (Isar Utoks Abbrev; Scripting )-----  
Wrote /Users/lp15/Dropbox/ACS/8 - Operational Semantics/Com.thy

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  $\sim$  c"
by (auto simp add: equiv_c_def)

lemma equiv_sym:
  "c1  $\sim$  c2  $\implies$  c2  $\sim$  c1"
by (auto simp add: equiv_c_def)

lemma equiv_trans:
  "c1  $\sim$  c2  $\implies$  c2  $\sim$  c3  $\implies$  c1  $\sim$  c3"
by (auto simp add: equiv_c_def)
```

# More Semantic Equivalence!

```
Isabelle Proof General: Com.thy

lemma equiv_semi:
  "c1 ~ c1'  $\implies$  c2 ~ c2'  $\implies$  (c1; c2) ~ (c1'; c2')"
by (force simp add: equiv_c_def)

lemma equiv_if:
  "c1 ~ c1'  $\implies$  c2 ~ c2'  $\implies$  (IF b THEN c1 ELSE c2) ~ (IF b THEN c1' ELSE c2')"
by (force simp add: equiv_c_def)
□
```

shorthand for a one-line proof

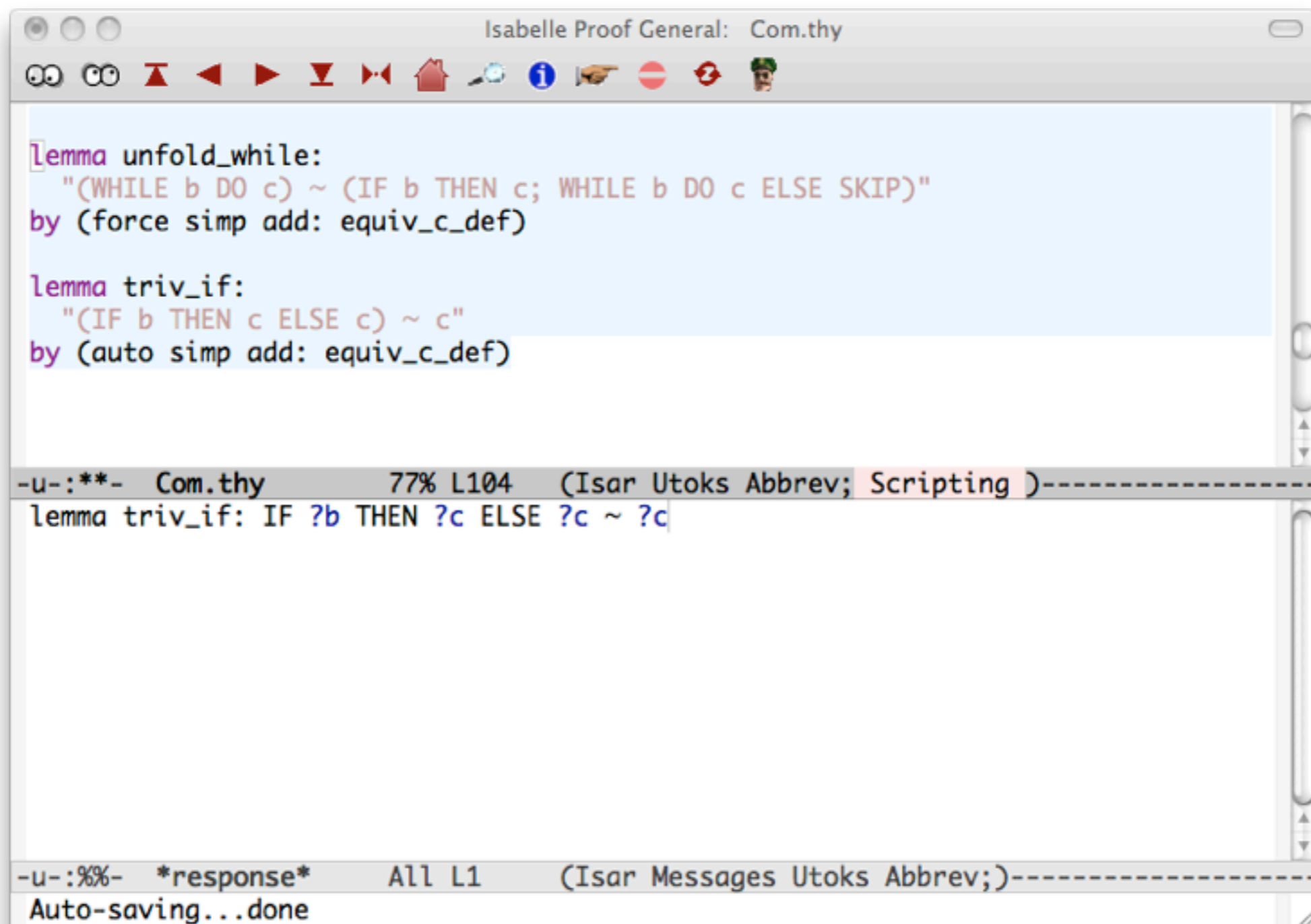
```
lemma
  equiv_if:
    [[?c1.0 ~ ?c1'; ?c2.0 ~ ?c2']]
     $\implies$  IF ?b THEN ?c1.0 ELSE ?c2.0 ~ IF ?b THEN ?c1' ELSE ?c2'|
```

commands built from equivalent commands are equivalent

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!!



```
Isabelle Proof General: Com.thy  
[lemma unfold_while:  
  "(WHILE b DO c) ~ (IF b THEN c; WHILE b DO c ELSE SKIP)"  
by (force simp add: equiv_c_def)  
  
lemma triv_if:  
  "(IF b THEN c ELSE c) ~ c"  
by (auto simp add: equiv_c_def)  
  
-u-:**- Com.thy          77% L104  (Isar Utoks Abbrev; Scripting )-----  
lemma triv_if: IF ?b THEN ?c ELSE ?c ~ ?c  
  
-u-:%%- *response*      All L1    (Isar Messages Utoks Abbrev;)-----  
Auto-saving...done
```

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”.

# Intro-Rule for Equivalence

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

declared like this

```
Isabelle Proof General: Com.thy
lemma equivI [intro!]:
  "( $\wedge s s'. \langle c, s \rangle \rightarrow s' = \langle c', s \rangle \rightarrow s'$ )  $\implies c \sim c'$ "
  by (auto simp add: equiv_c_def)

lemma commute_if:
  "(IF b1 THEN (IF b2 THEN c11 ELSE c12) ELSE c2)
  ~
  (IF b2 THEN (IF b1 THEN c11 ELSE c2) ELSE (IF b1 THEN c12 ELSE c2))"
  by blast
```

formalised like this

used *implicitly* like this

```
-u-:--- Com.thy
lemma
  commute_if:
    IF ?b1.0 THEN IF ?b2.0 THEN ?c11.0 ELSE ?c12.0 ELSE ?c2.0 ~ IF ?b2.0 THEN I
  IF ?b1.0 THEN ?c11.0 ELSE ?c2.0 ELSE IF ?b1.0 THEN ?c12.0 ELSE ?c2.0
```

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* can be 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 from Christian Urban's website: <http://www4.in.tum.de/~urbanc/publications.html>



# Interactive Formal Verification

## *I/O*: Structured Induction Proofs

Lawrence C Paulson  
Computer Laboratory  
University of Cambridge

# Structured Proofs: Review

- Structured Isar proofs are clearer than a series of commands, but verbose.
- The Isar language is rich and complex, supporting a great many proof styles.
- *Existential* reasoning is possible, naming entities that “exist”.
- Isar has syntax for proof by *induction*.
- No need to write out induction hypotheses.
- Cases given by name; bound variables named.
- And the same syntax works for *case analysis*.

# A Proof about Binary Trees

```
BT.thy
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"
proof (induct t)
  |
  proof (state): step 1
    goal (2 subgoals):
    1. reflect (reflect Lf) = Lf
    2.  $\wedge a t1 t2.$ 
       [[reflect (reflect t1) = t1; reflect (reflect t2) = t2]]
        $\Rightarrow$  reflect (reflect (Br a t1 t2)) = Br a t1 t2
```

Must we copy each case and such big contexts?



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

The screenshot shows the Emacs Isabelle interface. The menu bar includes 'Emacs', 'File', 'Edit', 'Options', 'Tools', 'Isabelle', 'Proof-General', 'Maths', 'Tokens', 'Buffers', and 'Help'. The 'Isabelle' menu is open, showing options like 'Logics', 'Commands', 'Show Me', 'Favourites', 'Settings', 'Start Isabelle', 'Exit Isabelle', 'Set Isabelle Command', and 'Help'. The 'Show Me' submenu is also open, listing various predefined cases and rules such as 'Cases', 'Facts', 'Term Bindings', 'Classical Rules', 'Induct/Cases Rules', 'Simplifier Rules', 'Theorems', 'Transitivity Rules', 'Antiquotations', 'Attributes', 'Commands', 'Inner Syntax', and 'Methods'. The code editor shows the following Isabelle code:

```
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 (ref

lemma reflect_reflect_ident: "reflect (reflect t) = t
proof (induct t)

cases:
Lf:
  let "?case" = "reflect (reflect Lf) = Lf"
Br:
  fix a_ t1_ t2_
  let "?case" = "reflect (reflect (Br a_ t1_ t2_)) = Br a_ t1_ t2_"
  assume
Br.hyps: "reflect (reflect t1_) = t1_" "reflect (reflect t2_) = t2_"
and Br.premis:
```

Annotations in the code editor include:

- A blue box labeled 'Built-in cases' with arrows pointing to the 'Lf:' and 'Br:' case definitions.
- A blue box labeled 'name of induction hyps' with an arrow pointing to the 'Br.hyps:' line.
- A blue box labeled 'abbreviation of conclusion' with an arrow pointing to the 'let "?case" = ...' line.

The status bar at the bottom shows '-u-:%%- \*response\* All L9 (Isar Messages Utoks Abbrev;)' and a menu bar 'Isabelle Show Me 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

```
BT.thy
[Lemma reflect_reflect_ident: "reflect (reflect t) = t"
proof (induct t)
  case Lf
  show ?case by simp
next
  case (Br a t1 t2)
  thus ?case by simp
qed
```

the two cases

list of bound variables

instances of the goal

```
-u-:**- BT.thy 56% L11 (Isar Utoks Abbrev; Scripting )-----
Successful attempt to solve goal by exported rule:
[[reflect (reflect ?t1.2) = ?t1.2; reflect (reflect ?t2.2) = ?t2.2]]
=> reflect (reflect (Br ?a2 ?t1.2 ?t2.2)) = Br ?a2 ?t1.2 ?t2.2
```

Isabelle has proved the induction step

```
-u-:%%- *response* All L3 (Isar Messages Utoks Abbrev;)
```

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

```
BT.thy
[ ]
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]

lemma "[| A ∈ Fin; B ⊆ A |] ==> B ∈ Fin"
proof (induct A arbitrary: B rule: Fin.induct)
-u-:--- BT.thy 29% L2 (Isar Utoks Abbrev; Scripting )-----
cases:
  emptyI:
    fix B
    let "?case" = "B ∈ Fin"
    assume emptyI.hyps: and emptyI.premis: "B ⊆ {}"
  insertI:
    fix A_ a_ B
    let "?case" = "B ∈ Fin"
    assume insertI.hyps: "A_ ∈ Fin" "∧B. B ⊆ A_ ==> B ∈ Fin" and
      insertI.premis: "B ⊆ insert a_ A_"
-u-:%%- *response* All L10 (Isar Messages Utoks Abbrev;)-
```

a named induction rule

an arbitrary variable

non-empty premises

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,  $B \subseteq A$ , instances of which are attached to both subgoals.

# Proving the Base Case

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

declare Fin.intros [intro]

lemma "[| A ∈ Fin; B ⊆ A |] ==> B ∈ Fin"
proof (induct A arbitrary: B rule: Fin.induct)
  case (emptyI B)
  thus "B ∈ Fin"

proof (prove): step 3
  using this:
    B ⊆ {}

goal (1 subgoal):
  1. B ∈ Fin
```

BT.thy 35%

BT.thy 35% | “arbitrary” variables must be named!

“thus” makes the premise available

\*goals\* Top L1 (Isar Proofstate Utoks Abbrev;)

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

```
declare Fin.intros [intro]
lemma "[| A ∈ Fin; B ⊆ A |] ⇒ B ∈ Fin"
proof (induct A arbitrary: B rule: Fin.induct)
  case (emptyI B)
  thus "B ∈ Fin"
  by auto
next
  case (insertI A a B)
  show "B ∈ Fin"
  proof (cases "B ⊆ A")
    case (emptyI B)
    thus "B ∈ Fin"
    by auto
  case (insertI A a B)
  thus "B ∈ Fin"
  by auto
qed
```

“arbitrary” variables must (again) be named!

case analysis on this formula

```
proof (state): step 8
goal (2 subgoals):
1. B ⊆ A ⇒ B ∈ Fin
2. ¬ B ⊆ A ⇒ B ∈ Fin
```

Here we know  $B \subseteq \text{insert } a \text{ } A$ , as it is the inherited premise of this case. But do we in fact know  $B \subseteq A$ ?



# The Complete Proof

```
BTplus.thy
lemma "[| A ∈ Fin; B ⊆ A |] ==> B ∈ Fin"
proof (induct A arbitrary: B rule: Fin.induct)
  case (emptyI B)
  thus "B ∈ Fin"
  by auto
next
  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
qed
```

Annotations:

- true and false cases
- induction hypothesis and premise
- the true case:  $B \subseteq A$
- direct quotation of a fact
- the false case:  $\neg B \subseteq A$

BTplus.thy 20% L50 (Isar Utoks Abbrev; Scripting )

Here is an outline of the proof. If  $B \subseteq A$ , 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  $B - \{a\} \in \text{Fin}$ , and therefore  $B = \text{insert } a (B - \{a\}) \in \text{Fin}$ .

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  $B - \{a\} \subseteq A$ .

# Which Theorems are Available?

The screenshot shows the Emacs Isabelle interface. The main window displays a proof script for a theorem `Ba`. The script includes a `have` statement, a `by auto` tactic, and a `hence` statement. Below the proof, the `facts` section lists the current facts, including `Ba` and `False`. The `assms` section lists the assumptions, including `insertI`. The `insertI.hyps` and `insertI.prem` sections show the hypotheses and premises for the `insertI` tactic. The `unnamed` section shows the current state of the proof, including the `False` assumption and the `Ba` fact.

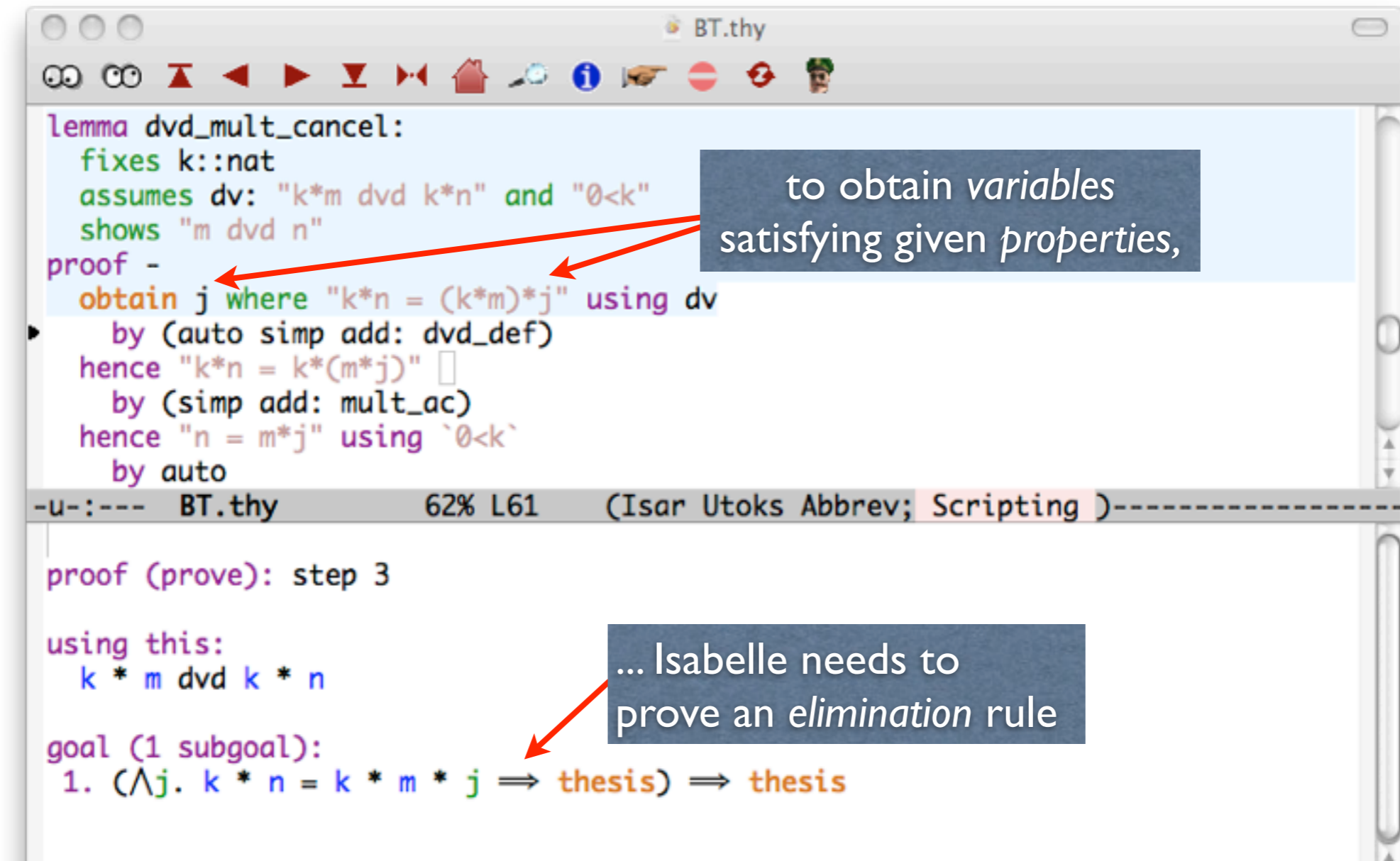
The menu on the right lists the available theorems, including:

- Logics
- Commands
- Show Me
- Favourites
- Settings
- Cases (C-c C-a <h> <c>)
- Facts (C-c C-a <h> <f>)
- Term Bindings (C-c C-a <h> <b>)
- Classical Rules (C-c C-a <h> <C>)
- Induct/Cases Rules (C-c C-a <h> <I>)
- Simplifier Rules (C-c C-a <h> <S>)
- Theorems (C-c C-a <h> <t>)
- Transitivity Rules (C-c C-a <h> <T>)
- Antiquotations (C-c C-a <h> <A>)
- Attributes (C-c C-a <h> <a>)
- Commands (C-c C-a <h> <o>)
- Inner Syntax (C-c C-a <h> <i>)
- Methods (C-c C-a <h> <m>)

Annotations in the image point to specific parts of the proof and the menu:

- a recently proved fact** points to the `Ba` fact in the `facts` section.
- the false case:  $\neg B \subseteq A$**  points to the `False` fact in the `facts` section.
- facts for the case insertI** points to the `insertI` assumption in the `assms` section.
- separate hyps and prems for insertI** points to the `insertI.hyps` and `insertI.prem` sections.

# Existential Claims: “obtain”



```
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
```

to obtain *variables* satisfying given *properties*,

```
proof (prove): step 3
using this:
  k * m dvd k * n
goal (1 subgoal):
1. ( $\wedge j. k * n = k * m * j \Rightarrow thesis$ )  $\Rightarrow thesis$ 
```

... Isabelle needs to prove an *elimination rule*

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

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

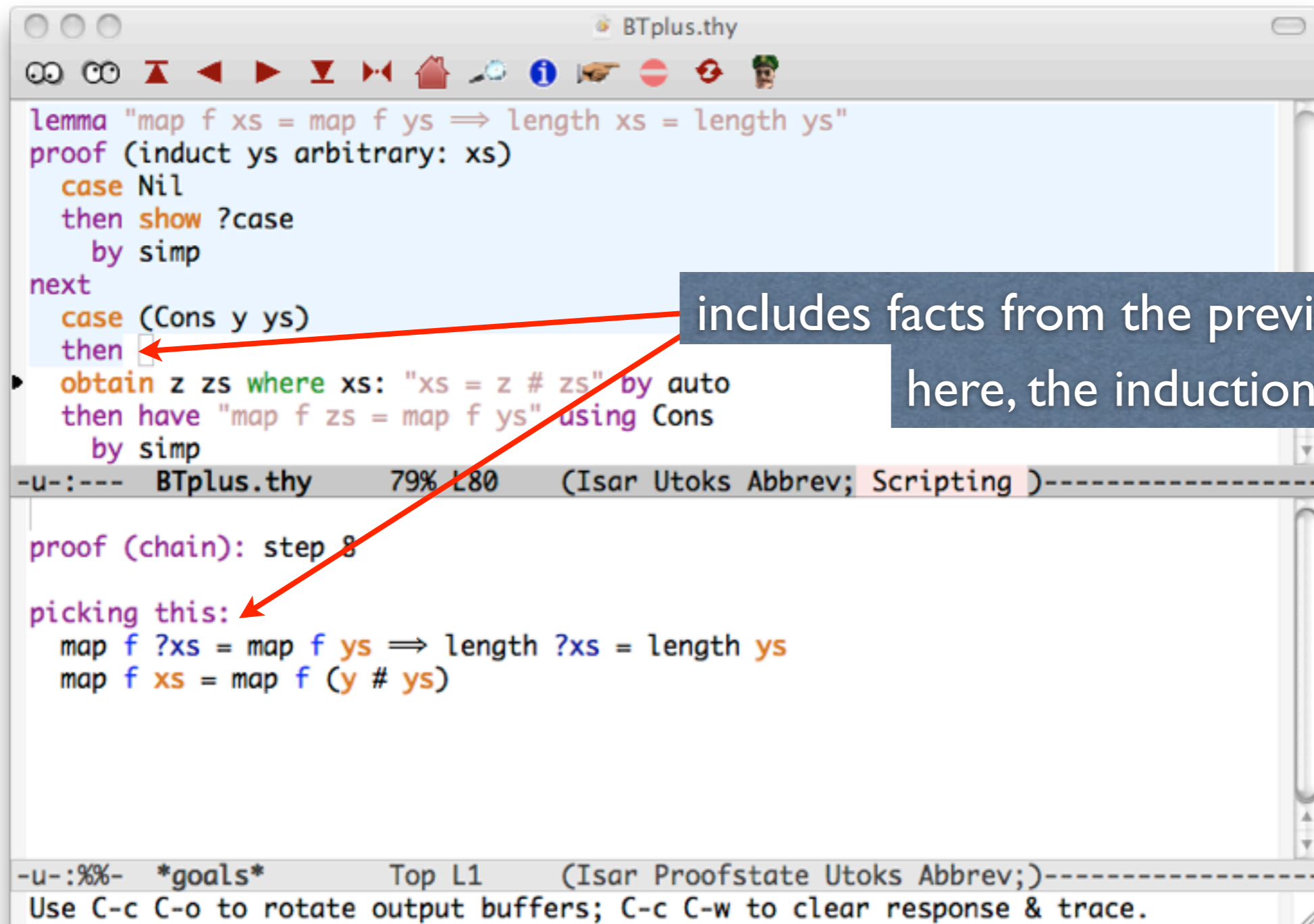
```
BT.thy
--
lemma dvd_mult_cancel:
  fixes k::nat
  assumes dv: "k*m dvd k*n" and "0<k"
  shows "m dvd n"
proof -
  obtain j where "k*n = (k*m)*j" using dv
    by (auto simp add: dvd_def)
  hence "k*n = k*(m*j)"
  by (simp add: mult_ac)
  hence "n = m*j" using `0<k`
    by auto
--
-u-:--- BT.thy 62% 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)
|
|-----
-u-:%%- *goals* Top L1 (Isar Proofstate Utoks Abbrev;)-----
| tool-bar next
```

we now have the key property of j

# 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` ← removing k from
    by auto                                     the equality
  thus "m dvd n"
    by (auto simp add: dvd_def)
qed]
-u:--- BT.thy 62% L54 (Isar Utoks Abbrev; Scripting )-----
proof (state): step 11
this:
  m dvd n
goal:
No subgoals!
-u:%%- *goals* Top L1 (Isar Proofstate Utoks Abbrev;)
```

# Introducing “then”



```
BTplus.thy
lemma "map f xs = map f ys => length xs = length ys"
proof (induct ys arbitrary: xs)
  case Nil
  then show ?case
    by simp
next
  case (Cons y ys)
  then
    obtain z zs where xs: "xs = z # zs" by auto
    then have "map f zs = map f ys" using Cons
      by simp
proof (chain): step 8
picking this:
  map f ?xs = map f ys => length ?xs = length ys
  map f xs = map f (y # ys)
```

includes facts from the previous step  
here, the induction context

\*goals\* Top L1 (Isar Proofstate Utoks Abbrev;)

Use C-c C-o to rotate output buffers; C-c C-w to clear response & trace.

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”

```
BTplus.thy
lemma "map f xs = map f ys => length xs = length ys"
proof (induct ys arbitrary: xs)
  case Nil
  then show ?case
    by simp
  next
  case (Cons y ys)
  then
  obtain z zs where xs: "xs = z # zs" by auto
  then have "map f zs = map f ys" using Cons
    by simp
-u-:--- BTplus.thy 79% L81 (Isar Utoks Abbrev; Scripting )-----

proof (prove): step 9
using this:
  map f ?xs = map f ys => length ?xs = length ys
  map f xs = map f (y # ys)
goal (1 subgoal):
1. (∧z zs. xs = z # zs => thesis) => thesis
```

we “obtain” two quantities

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

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

# Facts from Two Sources

```
BTplus.thy
lemma "map f xs = map f ys  $\implies$  length xs = length ys"
proof (induct ys arbitrary: xs)
  case Nil
  then show ?case
    by simp
next
  case (Cons y ys)
  then
  obtain z zs where xs: "xs = z # zs" by auto
  then have "map f zs = map f ys" using Cons
  by simp

```

BTplus.thy 79% L83 (Isar Utoks Abbrev; Scripting)

```
proof (prove): step 13
using this:
  xs = z # zs
  map f ?xs = map f ys  $\implies$  length ?xs = length ys
  map f xs = map f (y # ys)
goal (1 subgoal):
1. map f zs = map f ys

```

\*goals\* Top L1 (Isar Proofstate Utoks Abbrev;)

tool-bar next

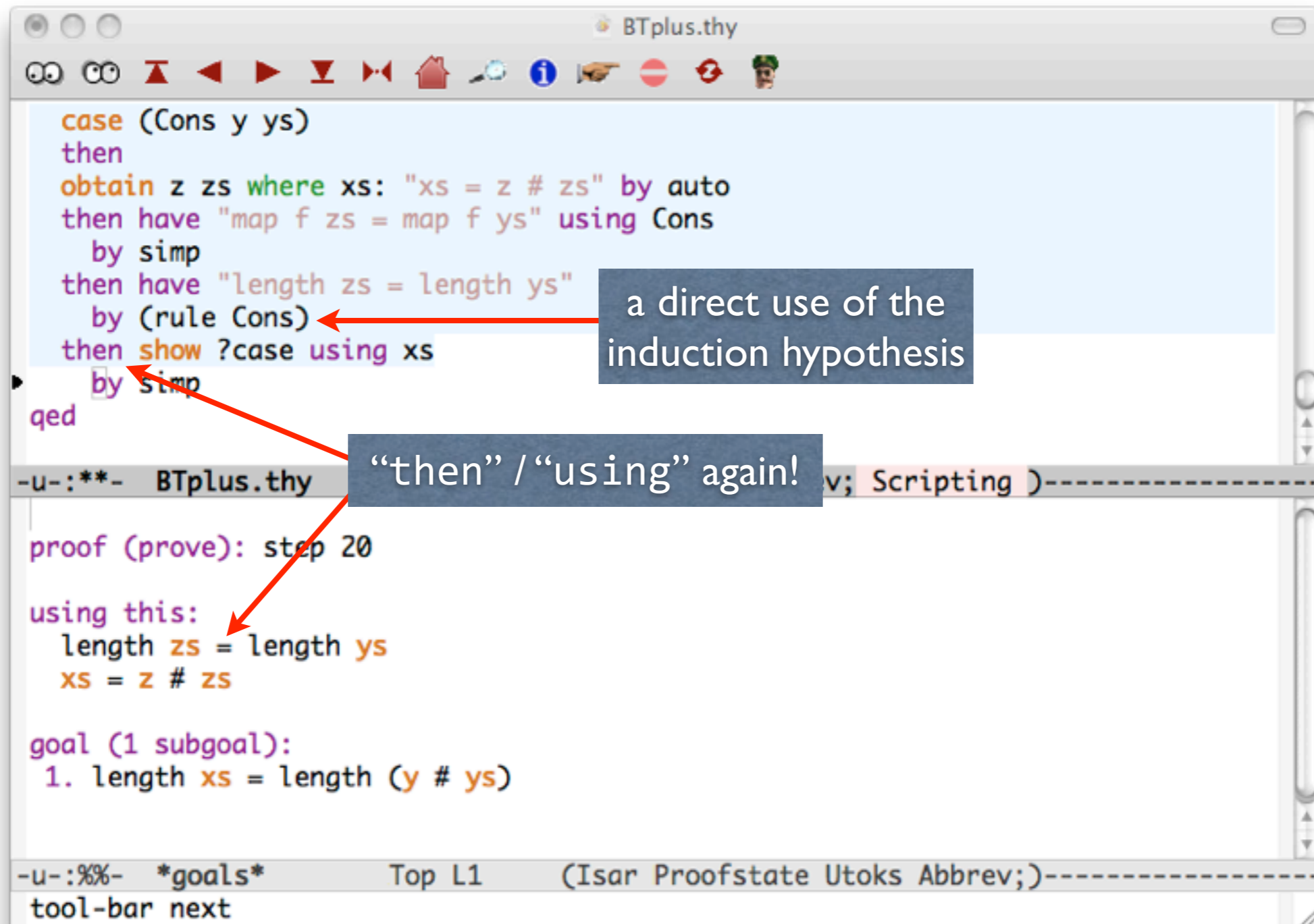
the effect of "then"

the effect of "using"

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



```
case (Cons y ys)
then
  obtain z zs where xs: "xs = z # zs" by auto
  then have "map f zs = map f ys" using Cons
    by simp
  then have "length zs = length ys"
    by (rule Cons)
  then show ?case using xs
    by simp
qed
```

a direct use of the induction hypothesis

```
-u-:**- BTplus.thy v; Scripting )-----
```

```
proof (prove): step 20
using this:
  length zs = length ys
  xs = z # zs
goal (1 subgoal):
  1. length xs = length (y # ys)
```

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

Unusually, we prove  $\text{length } zs = \text{length } ys$  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

```
BTplus.thy
[Lemma "map f xs = map f ys  $\implies$  length xs = length ys"
proof (induct ys arbitrary: xs)
  case Nil
  then show ?case
    by simp
next
  case (Cons y ys)
  then
  obtain z zs where xs: "xs = z # zs" by auto
  then have "map f zs = map f ys" using Cons
    by simp
  then have "length zs = length ys"
    by (rule Cons)
  then show ?case using xs
    by simp
qed
```

“then have” = “hence”

“then show” = “thus”

```
-u-:**- BTplus.thy 79%
Successful attempt to solve goal by exported rule:
[[ $\wedge$ xs. map f xs = map f ?ysa2  $\implies$  length xs = length ?ysa2;
  map f ?xsa2 = map f (?y2 # ?ysa2)]]
 $\implies$  length ?xsa2 = length (?y2 # ?ysa2)
```

```
-u-:%%- *response* All L4 (Isar Messages Utoks Abbrev;)
```

# 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  $\langle facts \rangle$  ... = ... using  $\langle facts \rangle$

with  $\langle facts \rangle$  ... = then from  $\langle facts \rangle$  ...

(where ... is have / show / obtain)

# Interactive Formal Verification

## //: Modelling Hardware

Lawrence C Paulson  
Computer Laboratory  
University of Cambridge

# Outline

- General modelling techniques
- Hardware verification in higher-order logic
- Additional elements of the Isar language, for instantiating theorems

# 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.

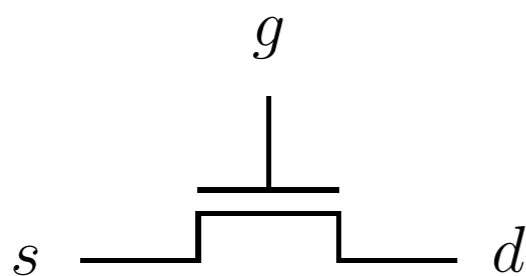
# Hardware Verification

- Pioneered by Prof. M. J. C. Gordon and his students, using successive versions of the HOL system.
- Works *hierarchically* from arithmetic units and memories right down to flip-flops and transistors.
- Used to model substantial hardware designs:
  - VIPER chip verification, by Avra Cohn (1988)
  - The ARM6 processor, by Anthony Fox (2003)
- Crucially uses *higher-order logic*, modelling signals as boolean-valued functions over time.

# Devices as Relations



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



$g \rightarrow s = d$

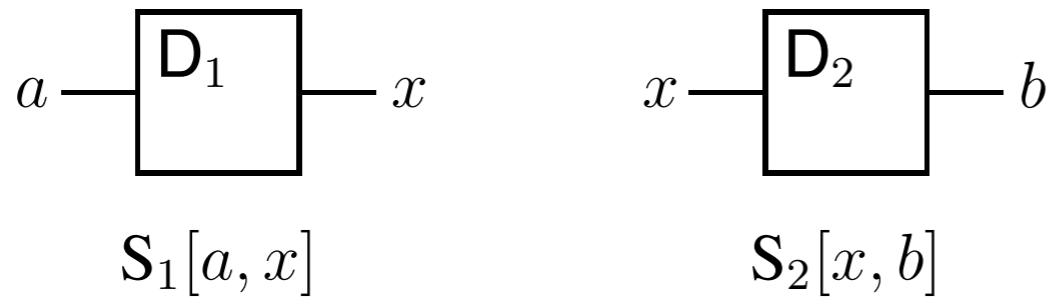
The relation describes the possible combinations of values on the ports.

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

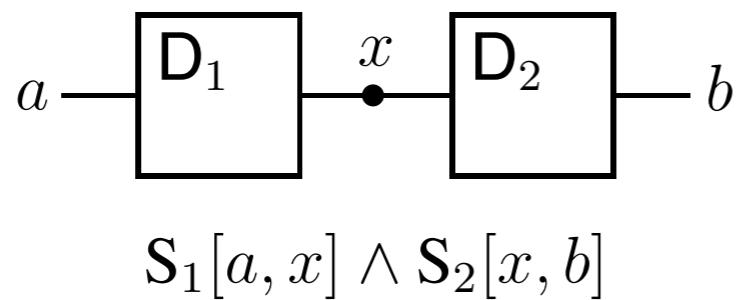
The second device on the slide above is an N-type field effect transistor, which can be conceived as a switch: when the gate goes high, the source and drain are connected. The logical implication shown next to the transistor formalises this behaviour. Note that the connection between the source and drain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



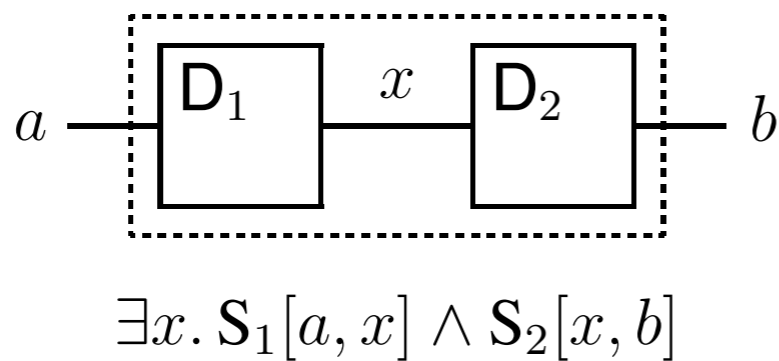
# Relational Composition



two devices modelled  
by two formulas



the connected ports  
have the *same* value



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*:  $Imp \Leftrightarrow Spec$ .
- The property  $Imp \Rightarrow Spec$  ensures that every possible behaviour of the *Imp* is permitted by *Spec*.

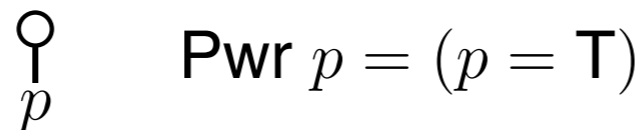
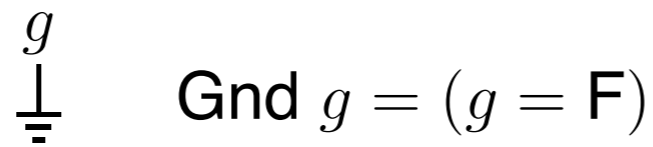
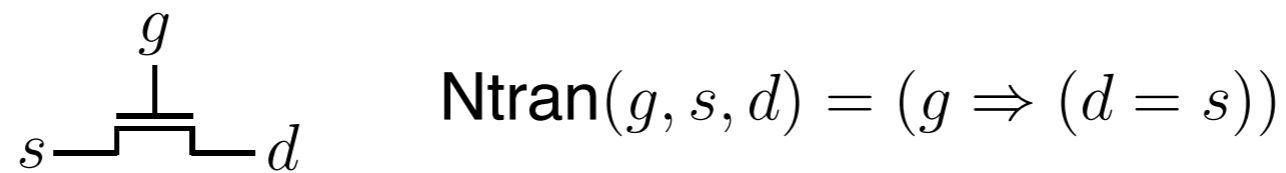
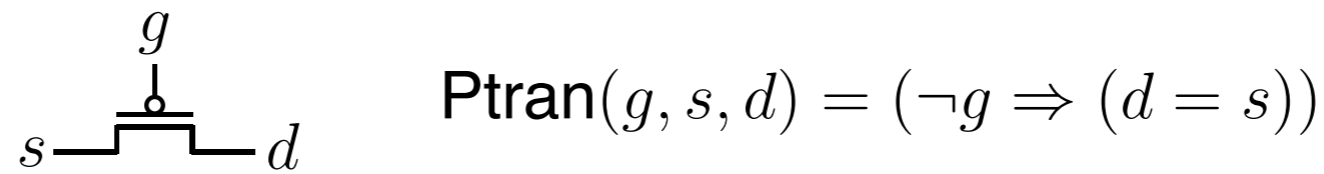
***Impossible implementations satisfy all specifications!***

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



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

```
text{* P and N transistors *}
```

```
definition "Ptran = ( $\lambda(g,a,b). (\sim g \longrightarrow a = b)$ )"
```

```
definition "Ntran = ( $\lambda(g,a,b). (g \longrightarrow a = b)$ )"
```

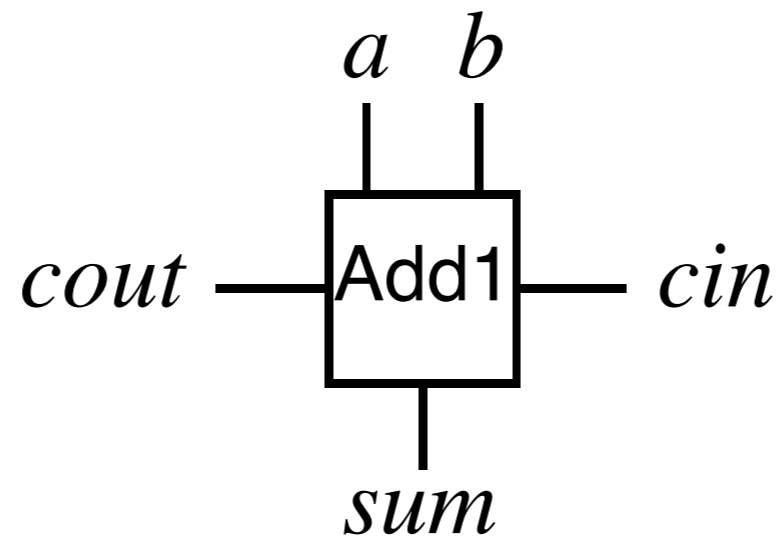
```
text{* Power and Ground*}
```

```
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



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

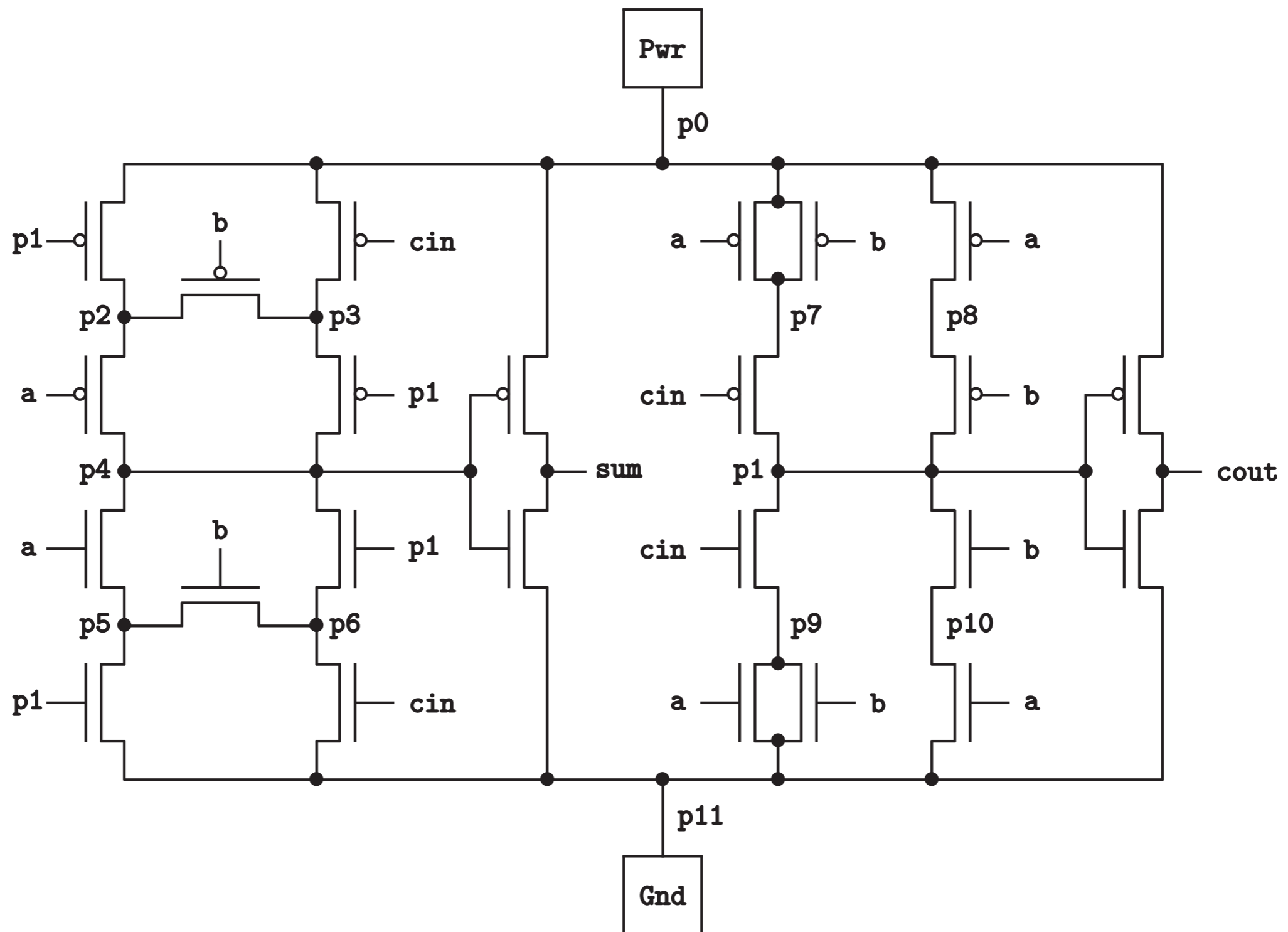
```
text{* 1-bit full adder specification *}

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

```
Adder.thy
text{* 1-bit CMOS full adder implementation *}

definition "Add1Imp = (λ(a,b,cin,sum,cout).
  ∃p0 p1 p2 p3 p4 p5 p6 p7 p8 p9 p10 p11.
    Ptran(p1,p0,p2)  ^  Ptran(cin,p0,p3)  ^
    Ptran(b,p2,p3)   ^  Ptran(a,p2,p4)   ^
    Ptran(p1,p3,p4)  ^  Ntran(a,p4,p5)   ^
    Ntran(p1,p4,p6)  ^  Ntran(b,p5,p6)   ^
    Ntran(p1,p5,p11) ^  Ntran(cin,p6,p11) ^
    Ptran(a,p0,p7)   ^  Ptran(b,p0,p7)   ^
    Ptran(a,p0,p8)   ^  Ptran(cin,p7,p1)  ^
    Ptran(b,p8,p1)   ^  Ntran(cin,p1,p9)  ^
    Ntran(b,p1,p10)  ^  Ntran(a,p9,p11)  ^
    Ntran(b,p9,p11)  ^  Ntran(a,p10,p11) ^
    Pwr(p0)          ^  Ptran(p4,p0,sum)  ^
    Ntran(p4,sum,p11) ^  Gnd(p11)        ^
    Ptran(p1,p0,cout) ^  Ntran(p1,cout,p11))"

text{* Verification of CMOS full adder *}
lemma Add1Correct:
  "Add1Imp(a,b,cin,sum,cout) = Add1Spec(a,b,cin,sum,cout)"
by (simp add: Pwr_def Gnd_def Ntran_def Ptran_def Add1Spec_def
    Add1Imp_def bit_val_def ex_bool_eq)

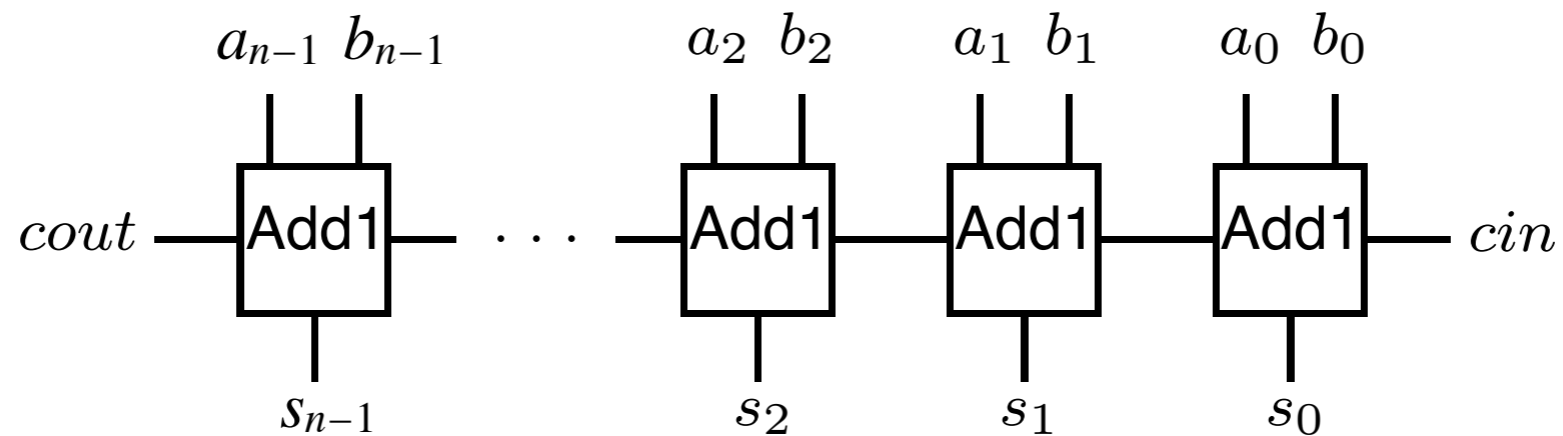
-u-:**- Adder.thy 27% L53 (Isar Utoks Abbrev; Scripting )-----
```

$$(\exists b. P b) = (P \text{ True} \vee P \text{ False})$$

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

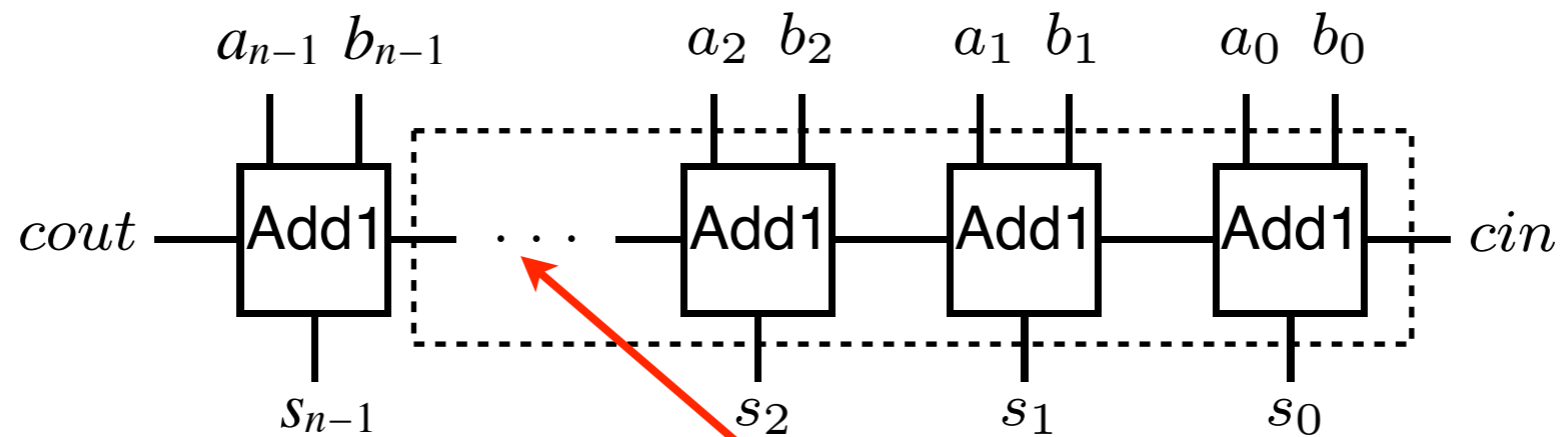
```
text{* Unsigned number denoted by bitstring f(n-1)...f(0) *}
fun bits_val where
  "bits_val f 0 = 0"
| "bits_val f (Suc n) = 2^n * bit_val(f n) + bits_val f n"

text{* Specification of an n-bit adder *}
□
definition
  "AdderSpec n = (λa, 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

```
Adder.thy
--u-:--- Adder.thy      53% L85      (Isar Utoks Abbrev; Scripting )-----
lemma AdderCorrect:
  "AdderImp n (a, b, cin, sum, cout)  $\Rightarrow$  AdderSpec n (a, b, cin, sum, cout)"
proof (induct n arbitrary: cout)
  case 0 thus ?case
    by (simp add: AdderSpec_def)
next
  case (Suc n)
  then obtain c
    where AddS: "AdderSpec n (a, b, cin, sum, c)"
    and Add1: "Add1Imp (a n, b n, c, sum n, cout)"
    by (auto intro: Suc)
--u-:%%- *goals*      5% L4      (Isar Proofstate Utoks Abbrev;)-----

this:
AdderImp n (a, b, cin, sum, ?cout)  $\Rightarrow$  AdderSpec n (a, b, cin, sum, ?cout)
AdderImp (Suc n) (a, b, cin, sum, cout)

goal (1 subgoal):
1.  $\wedge n$  cout.
  [[ $\wedge$ cout.
   AdderImp n (a, b, cin, sum, cout)  $\Rightarrow$ 
   AdderSpec n (a, b, cin, sum, cout);
   AdderImp (Suc n) (a, b, cin, sum, cout)]
 $\Rightarrow$  AdderSpec (Suc n) (a, b, cin, sum, cout)
```

assumptions

conclusion

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

```
lemma AdderCorrect:
  "AdderImp n (a, b, cin, sum, cout)  $\implies$  AdderSpec n (a, b, cin, sum, cout)"
proof (induct n arbitrary: cout)
  case 0 thus ?case
    by (simp add: AdderSpec_def)
next
  case (Suc n)
  then obtain c
    where AddS: "AdderSpec n (a, b, cin, sum, c)"
    and Add1: "Add1Imp (a n, b n, c, sum n, cout)"
    by (auto intro: Suc)
  have ( $\wedge$ c. [AdderSpec n (a, b, cin, sum, c);
    Add1Imp (a n, b n, c, sum n, cout)]
     $\implies$  ?thesis)  $\implies$ 
    ?thesis
```

-u-:--- Adder.thy 53% L80 (Isar ting )-----

-u-:%%- \*response\* All L4 (Isar Messages Utoks Abbrev;)-----

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

```
where AddS: "AdderSpec n (a, b, cin, sum, c)"
and Add1: "Add1Imp (a n, b n, c, sum n, cout)"
by (auto intro: Suc)
have "bit_val (sum n) * (2 ^ n) + bit_val cout * (2 * 2 ^ n) =
      (bit_val (sum n) + (bit_val cout * 2)) * (2 ^ n)"
by (simp add: algebra_simps)
also have "... = (bit_val c + (bit_val (a n) + bit_val (b n))) *
              (2 ^ n)"
using Add1 by (simp add: Add1Correct Add1Spec_def)
finally show "AdderSpec (Suc n) (a, b, cin, sum, cout)" u
by (simp add: AdderSpec_def algebra_simps)
-u-:--- Adder.thy 57% L96 (Isar Utoks Abbrev; Scripting )-----
calculation:
bit_val (sum n) * 2 ^ n + bit_val cout * (2 * 2 ^ n) =
(bit_val c + (bit_val (a n) + bit_val (b n))) * 2 ^ n
-u-:%%- *response* All L3 (Isar Messages Utoks Abbrev;)-----
tool-bar next
```

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

```
text{* Partial correctness of ripple-carry adder for all n by induction *}
lemma AdderCorrect:
  "AdderImp n (a, b, cin, sum, cout)  $\Rightarrow$  AdderSpec n (a, b, cin, sum, cout)"
proof (induct n arbitrary: cout)
  case 0 thus ?case
    by (simp add: AdderSpec_def)
next
  case (Suc n)
  then obtain c
    where AddS: "AdderSpec n (a, b, cin, sum, c)"
    and Add1: "Add1Imp (a n, b n, c, sum n, cout)"
    by (auto intro: Suc)
  have "bit_val (sum n) * (2 ^ n) + bit_val cout * (2 * 2 ^ n) =
        (bit_val (sum n) + (bit_val cout * 2)) * (2 ^ n)"
    by (simp add: algebra_simps)
  also have "... = (bit_val c + (bit_val (a n) + bit_val (b n))) *
        (2 ^ n)"
    using Add1 by (simp add: Add1Correct Add1Spec_def)
  finally show "AdderSpec (Suc n) (a, b, cin, sum, cout)" using AddS
    by (simp add: AdderSpec_def algebra_simps)
qed
```



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

```

Adder.thy
[Lemma AdderSpec_Suc:
  "AdderSpec (Suc n) (a, b, cin, sum, cout) =
    (∃c. AdderSpec n (a, b, cin, sum, c) & Add1Spec (a n, b n, c, sum n, cout))"
apply (auto simp add: AdderSpec_def Add1Spec_def ex_bool_eq bit_val_def)
-u-:**- Adder.thy      82% L130  (Isar Utoks Abbrev; Scripting )-----
goal (16 subgoals):
  1. [[a n; b n; sum n; ¬ cout; cin;
      bits_val sum n = Suc (2 ^ n + (bits_val a n + bits_val b n))]
      ⇒ False
  2. [[a n; b n; sum n; ¬ cout; ¬ cin;
      bits_val sum n = 2 ^ n + (bits_val a n + bits_val b n))]
      ⇒ False
  3. [[a n; b n; ¬ sum n; ¬ cout; cin;
      bits_val sum n = Suc (2 ^ n + bits_val a n + (2 ^ n + bits_val b n))]
      ⇒ False
  4. [[a n; b n; ¬ sum n; ¬ cout; ¬ cin;
      bits_val sum n = 2 ^ n + bits_val a n + (2 ^ n + bits_val b n))]
      ⇒ False
  5. [[a n; ¬ b n; sum n; cout; cin;
      2 * 2 ^ n + bits_val sum n = Suc (bits_val a n + bits_val b n))]
      ⇒ False
  6. [[a n; ¬ b n; sum n; cout; ¬ cin;
      2 * 2 ^ n + bits_val sum n = bits_val a n + bits_val b n]]
-u-:%%-  *goals*      2% L4  (Isar Proofstate Utoks Abbrev;)-----

```

just need to prove this...

HELP!!

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

```
lemma bits_val_less: "bits_val f n < 2^n"
by (induct n, auto simp add: bit_val_def)

lemma AdderSpec_Suc:
  "AdderSpec (Suc n) (a, b, cin, sum, cout) =
    (∃c. AdderSpec n (a, b, cin, sum, c) & Add1Spec (a n, b n, c, sum n, cout))"
using bits_val_less [of a n] bits_val_less [of b n] bits_val_less [of sum n]
by (simp add: AdderSpec_def Add1Spec_def ex_bool_eq bit_val_def)

proof (prove): step 1
using this:
  bits_val a n < 2 ^ n
  bits_val b n < 2 ^ n
  bits_val sum n < 2 ^ n

goal (1 subgoal):
1. AdderSpec (Suc n) (a, b, cin, sum, cout) =
  (∃c. AdderSpec n (a, b, cin, sum, c) ^
    Add1Spec (a n, b n, c, sum n, cout))
```

a trivial upper bound on the value of a bit string

inserting three instances of that fact

now proof is trivial, by arithmetic

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



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

```
lemma AdderCorrect2:  
  "AdderSpec n (a, b, cin, sum, cout)  $\Rightarrow$  AdderImp n (a, b, cin, sum, cout)"  
apply (induct n arbitrary: cout)  
apply (simp add: AdderSpec_def)  
apply (auto simp add: AdderSpec_Suc Add1Correct)  
done
```

Below the code is a status bar with the text: "-u-:\*\*- Adder.thy 91% L148 (Isar Utoks Abbrev; Scripting )-----".

Below the status bar is a message window with the text: "lemma AdderCorrect2: AdderSpec ?n (?a, ?b, ?cin, ?sum, ?cout)  $\Rightarrow$  AdderImp ?n (?a, ?b, ?cin, ?sum, ?cout)".

At the bottom of the message window is a status bar with the text: "-u-:%%- \*response\* All L4 (Isar Messages Utoks Abbrev;)-----".

The implementation and specification are equivalent!

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<sub>1</sub> thm<sub>2</sub> thm<sub>3</sub>*]  
discharges premises from left to right
- *thm* [simplified]  
applies the simplifier to *thm*
- *thm* [*attr<sub>1</sub>, attr<sub>2</sub>, attr<sub>3</sub>*]  
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: `thm1 [THEN thm2]` joins the conclusion of `thm1` to the premise of `thm2`. Thus it is equivalent to `thm2 [THEN thm1]`. 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 *1/2*: The Mutilated Chess Board

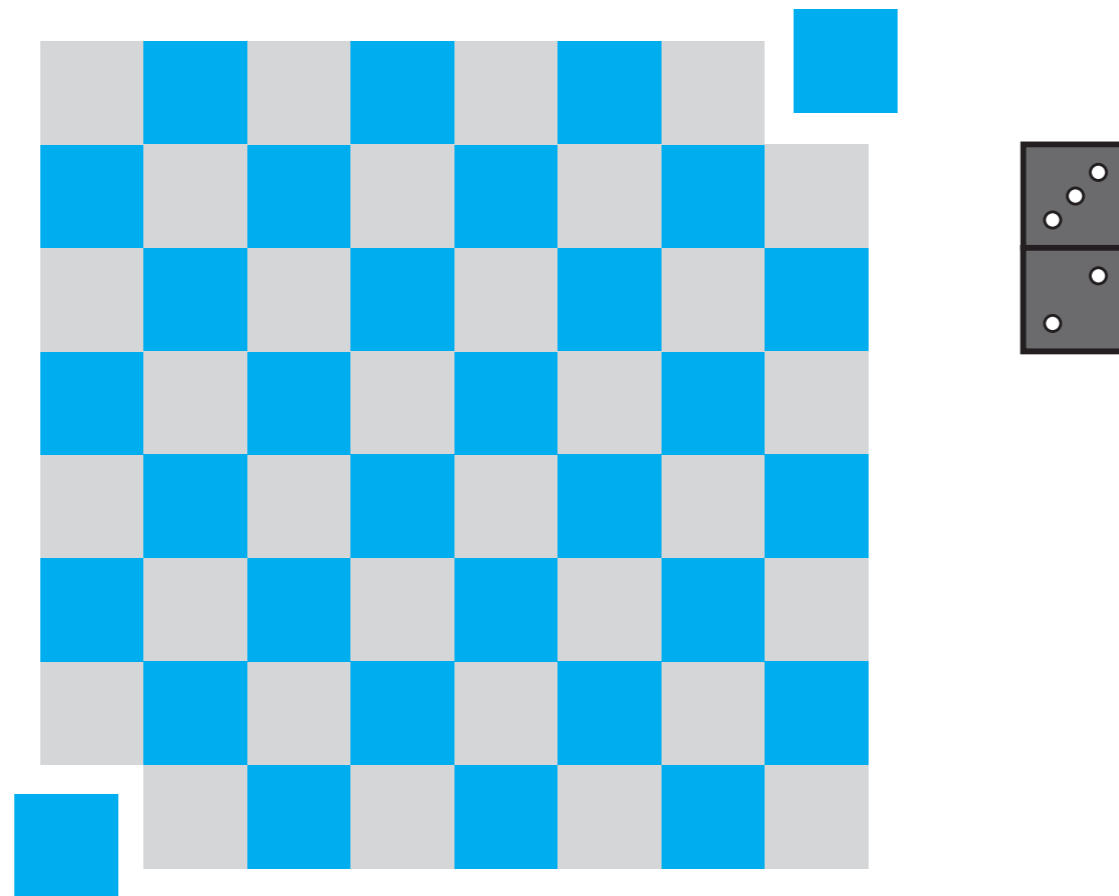
Lawrence C Paulson  
Computer Laboratory  
University of Cambridge

# Overview

- The mutilated chessboard: a classic example in modelling a problem intuitively.
- More techniques involving Isar.
- To conclude, brief references to other Isabelle tools and capabilities.

# The Mutilated Chess Board

Can this damaged board be tiled with 31 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

```
Tilings.thy
header { * The Mutilated Chess Board Cannot be Tiled by Dominoes * }
theory Tilings imports Main begin
text { * The originator of this problem is Max Black, according to J A
Robinson. It was popularized by J McCarthy. * }
section { * Inductive Tiling * }
inductive_set tiling :: "'a set set  $\Rightarrow$  'a set set" for A
where
  empty : "{}  $\in$  tiling A"
| Un : "[ a  $\in$  A; t  $\in$  tiling A; a  $\cap$  t = {} ]  $\Rightarrow$  a  $\cup$  t  $\in$  tiling A"
declare tiling.intros [intro]
-u-:**- Tilings.thy Top L1 (Isar Utoks Abbrev; Scripting )
-u-:%%- *response* All L1 (Isar Messages Utoks Abbrev;)
Beginning of buffer
```

given a set of tiles...

the empty set and  
a  $\cup$  t can be tiled

we give the introduction  
rules to auto and blast

# Simple Proofs about Tilings

```
Tilings.thy  
[lemma tiling_UnI [intro]:  
  "[ t1 ∈ tiling A; t2 ∈ tiling A; t1 ∩ t2 = {} ] ⇒ t1 ∪ t2 ∈ tiling A"  
by (induct rule: tiling.induct, auto simp add: Un_assoc)  
  
lemma tiling_finite:  
  assumes "∧a. a ∈ A ⇒ finite a"  
  shows "t ∈ tiling A ⇒ finite t"  
by (induct set: tiling, auto simp add: assms)  
-u-:***- Tilings.thy 11% L17 (Isar Utoks Abbrev; Scripting )-----  
lemma tiling_finite:  
  [∧a. a ∈ ?A ⇒ finite a; ?t ∈ tiling ?A] ⇒ finite ?t  
-u-:%%- *response* All L2 (Isar Messages Utoks Abbrev;)
```

for auto and blast...

referring to unnamed assumptions

another way to specify induction

a comma can join two methods

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

```
section{* Dominos and Colours *}

inductive_set domino :: "(nat × nat) set set" where
  horiz: "{(i, j), (i, Suc j)} ∈ domino"
| vertl: "{(i, j), (Suc i, j)} ∈ domino"

lemma domino_finite: "d ∈ domino ⇒ finite d"
by (cases set: domino, auto)

declare tiling_finite [OF domino_finite, simp]
```

each square is denoted by a pair of coordinates

a domino is horizontal or vertical

... and consists of two squares

combining two theorems

Tell the simplifier: every tiling using dominoes is finite

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

```
definition
  coloured :: "nat ⇒ (nat × nat) set" where
    "coloured b = {(i, j). (i + j) mod 2 = b}"

abbreviation
  whites :: "(nat × nat) set" where
    "whites ≡ coloured 0"

abbreviation
  blacks :: "(nat × nat) set" where
    "blacks ≡ coloured (Suc 0)"

text {*Every domino has a white square and a black square. *}

lemma domino_singletons:
  "d ∈ domino ⇒
   (∃i j. whites ∩ d = {(i,j)}) ∧ (∃m n. blacks ∩ d = {(m,n)})"
by (cases set: domino, auto simp add: coloured_def Int_insert_right mod_Suc)

--u-:--- Tilings.thy 28% L40 (Isar Utoks Abbrev; Scripting )-----
lemma
  domino_singletons:
    ?d ∈ domino ⇒
      (∃i j. whites ∩ ?d = {(i, j)}) ∧ (∃m n. blacks ∩ ?d = {(m, n)})
--u-:%%- *response* All L1 (Isar Messages Utoks Abbrev;)
```

colours defined using modular arithmetic

abbreviations provide notation

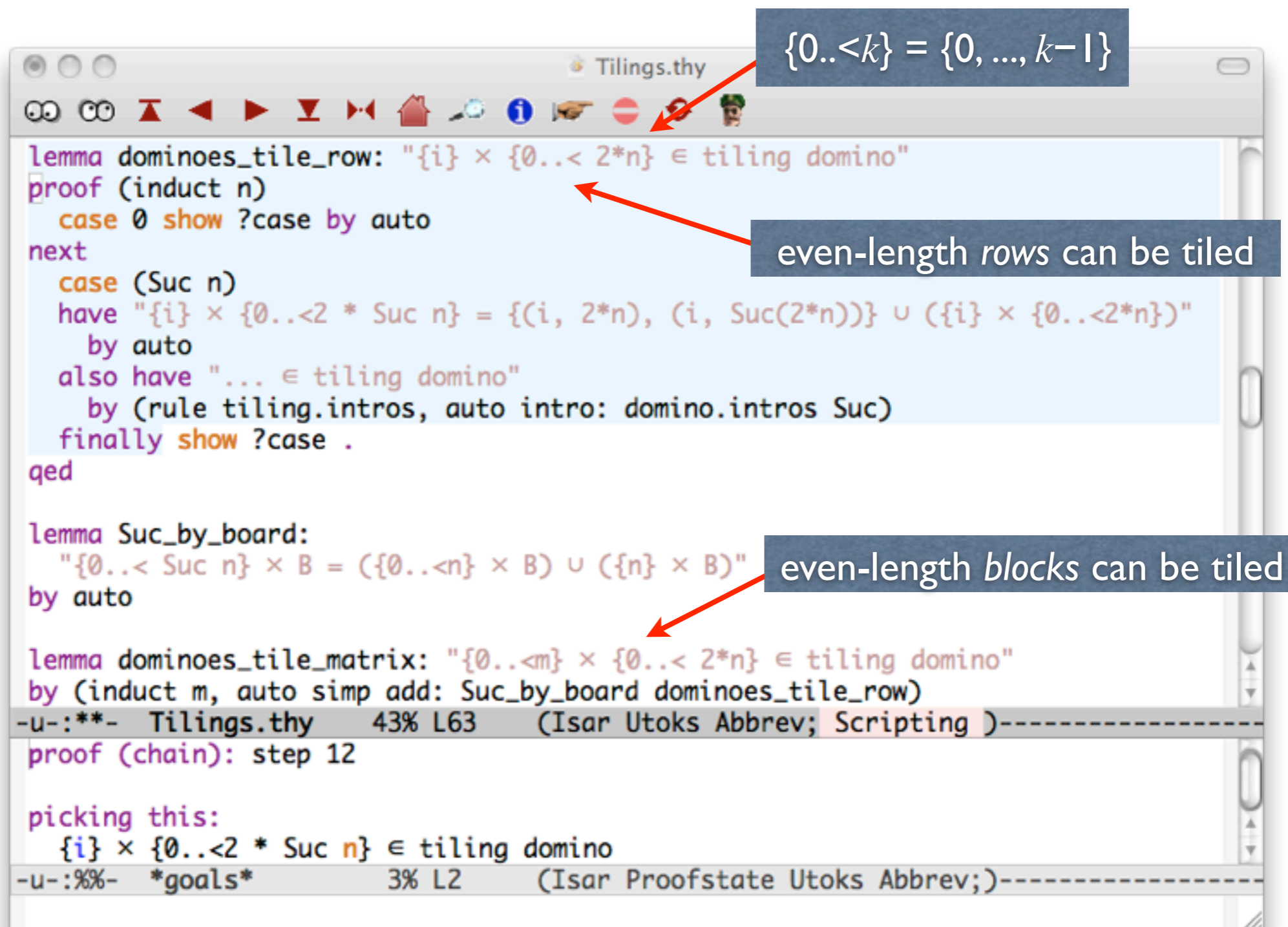
case analysis on the named set

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 screenshot shows a proof assistant window titled "Tilings.thy". The main text is a proof for the lemma `dominoes_tile_row`. The proof uses induction on `n`. The base case is `case 0 show ?case by auto`. The inductive step is `case (Suc n)`, where the goal is `"{i} × {0..< 2 * Suc n} = {(i, 2*n), (i, Suc(2*n))} ∪ ({i} × {0..< 2*n})"`. This is proven by `by auto` and `also have "... ∈ tiling domino"` using `rule tiling.intros` and `auto intro: domino.intros Suc`. The proof ends with `finally show ?case .` and `qed`.

Annotations with red arrows point to specific parts of the code:

- A box containing  $\{0..<k\} = \{0, \dots, k-1\}$  points to the set notation `{0..< 2*n}` in the lemma statement.
- A box containing "even-length rows can be tiled" points to the inductive step `case (Suc n)`.
- A box containing "even-length blocks can be tiled" points to the set notation `{0..< 2 * Suc n}` in the goal of the inductive step.

Below the main proof, there is a section for `proof (chain): step 12` showing a goal `{i} × {0..< 2 * Suc n} ∈ tiling domino` and a status bar indicating 3% L2 progress.

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

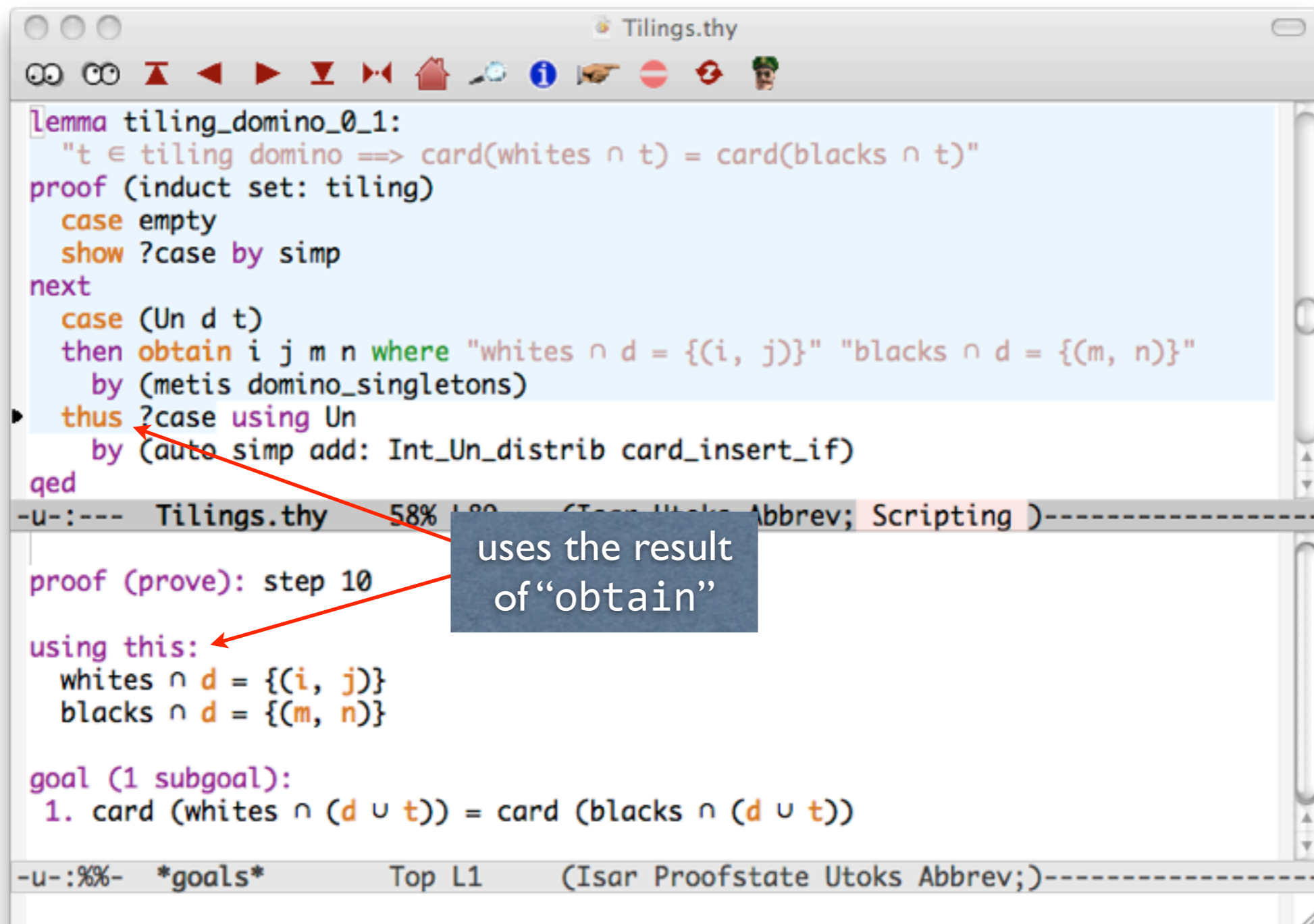
The alternative approach, of choosing rewrite rules that transform a term precisely as we wish, eliminates the need to write the intermediate stages of the transformation, but it can be more time-consuming overall. You know this other approach has been adopted if you see this sort of command:

```
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



```
Lemma tiling_domino_0_1:
  "t ∈ tiling domino ==> card(whites ∩ t) = card(blacks ∩ t)"
proof (induct set: tiling)
  case empty
  show ?case by simp
next
  case (Un d t)
  then obtain i j m n where "whites ∩ d = {(i, j)}" "blacks ∩ d = {(m, n)}"
  by (metis domino_singletons)
  thus ?case using Un
  by (auto simp add: Int_Un_distrib card_insert_if)
qed
```

-u-:--- Tilings.thy 58% 189 (Isar Utoks Abbrev; Scripting )-----

```
proof (prove): step 10
using this:
  whites ∩ d = {(i, j)}
  blacks ∩ d = {(m, n)}

goal (1 subgoal):
1. card (whites ∩ (d ∪ t)) = card (blacks ∩ (d ∪ t))
```

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

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

```
theorem gen_mutil_not_tiling:
  □ assumes "t ∈ tiling domino" "sqsq ⊆ whites ∩ t" "sqsq ≠ {}"
    shows "(t - sqsq) ∉ tiling domino"
  proof
    assume tm: "t - sqsq ∈ tiling domino"
    have fsqs: "finite sqsq" using assms
      by (metis Int_subset_iff finite_subset tiling_finite [OF domino_finite])
    hence c: "0 < card sqsq" "0 < card (whites ∩ t)" using assms
      by (auto simp add: card_gt_0_iff)
    have "card (whites ∩ (t - sqsq)) = card ((whites ∩ t) - sqsq)"
      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 - sqsq))"
      proof -
        have "blacks ∩ (t - sqsq) = 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
```

default proof of a negation

accumulating some facts

$\text{card (whites } \cap (t - \text{sqsq})) < \text{card (blacks } \cap (t - \text{sqsq}))$

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...

```
theorem mutil_not_tiling:
  fixes m n
  defines "t ≡ {0..< 2 * Suc m} × {0..< 2 * Suc n}"
  shows "t -> [(0,0), (Suc(2*m), Suc(2*n))] ∉ tiling domino"
apply (rule gen_mutil_not_tiling)
apply (metis dominoes_tile_matrix t_def)
apply (auto simp add: coloured_def t_def)
done

end

proof (prove): step 1

goal (3 subgoals):
1. t ∈ tiling domino
2. {(0, 0), (Suc (2 * m), Suc (2 * n))} ⊆ whites ∩ t
3. {(0, 0), (Suc (2 * m), Suc (2 * n))} ≠ {}

tool-bar next
```

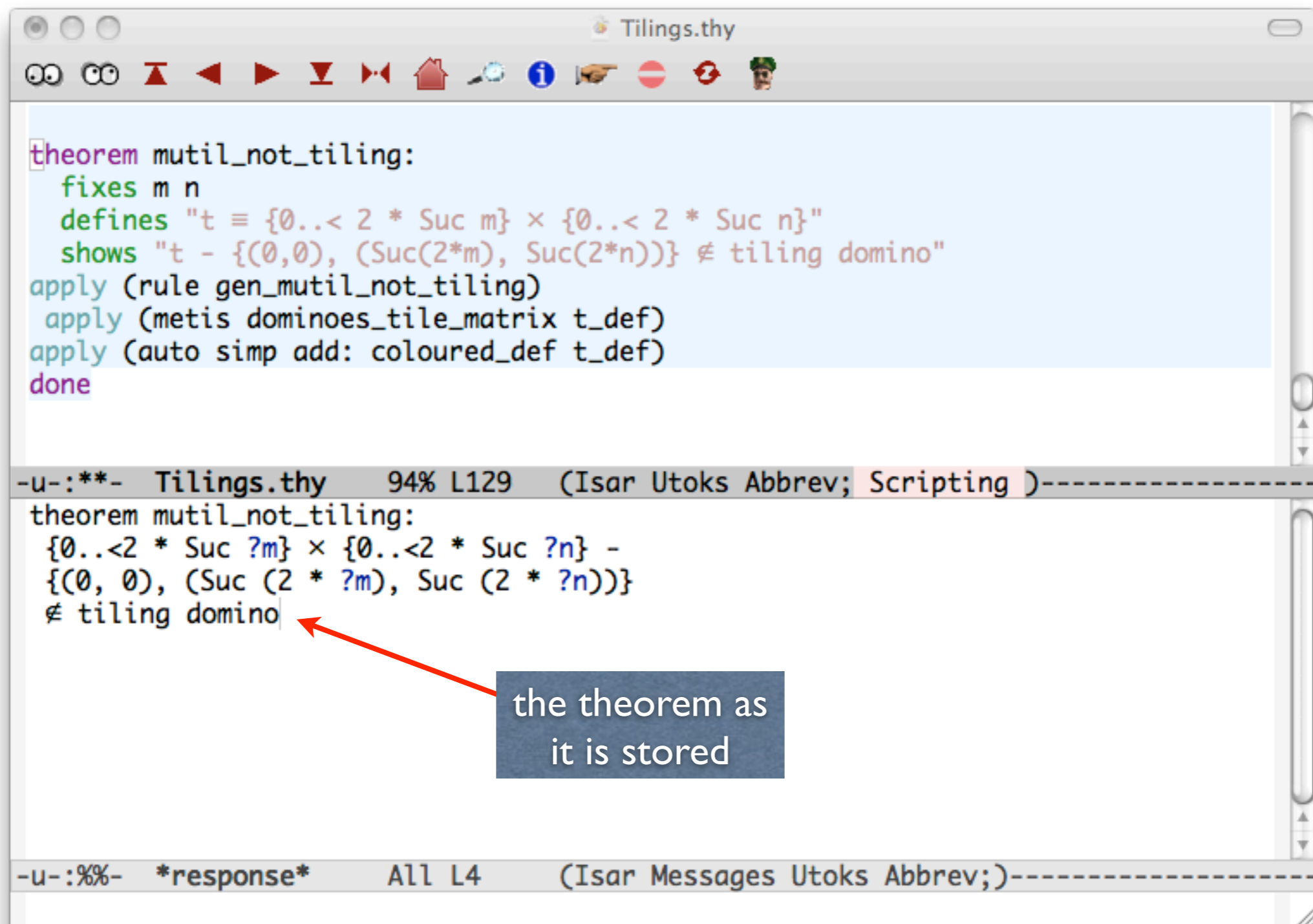
a local constant, t

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



The screenshot shows a theorem prover interface with a source editor and a message window. The source editor contains the following code:

```
theorem mutil_not_tiling:
  fixes m n
  defines "t ≡ {0..<2 * Suc m} × {0..<2 * Suc n}"
  shows "t - {(0,0), (Suc(2*m), Suc(2*n))} ∉ tiling domino"
apply (rule gen_mutil_not_tiling)
  apply (metis dominoes_tile_matrix t_def)
apply (auto simp add: coloured_def t_def)
done
```

The message window shows the theorem as it is stored in the system:

```
-u-:***- Tilings.thy 94% L129 (Isar Utoks Abbrev; Scripting )-----
theorem mutil_not_tiling:
  {0..<2 * Suc ?m} × {0..<2 * Suc ?n} -
  {(0, 0), (Suc (2 * ?m), Suc (2 * ?n))}
  ∉ tiling domino
```

A red arrow points from a text box to the theorem statement in the message window.

the theorem as  
it is stored

-u-:%%- \*response\* All L4 (Isar Messages Utoks Abbrev;)

# Finding Structured Proofs

```
assumes "\a. a \in A \implies finite a"
shows "t \in tiling A \implies finite t"
proof (induct set: tiling)
  case empty
  show ?case by simp
next
  case (Un d t)
  thus ?case
  apply (rule finite_UnI)
- u-:***- Tilings.thy 20% L28 (Isar
*** empty result sequence -- proof comm
*** At command "apply" (line 35 of "/Us
board/Tilings.thy").
```

It's okay to fool around with apply, but what if this keeps happening?

```
assumes "\a. a \in A \implies finite a"
shows "t \in tiling A \implies finite t"
proof (induct set: tiling)
  case empty
  show ?case by simp
next
  case (Un d t)
  thus ?case
  apply -
- u-:***- Tilings.thy 20% L36 (Isar Utoks Abbrev; Scripting )-----
proof (prove): step 8
goal (1 subgoal):
1. [[d \in A; t \in tiling A; finite t; d \cap t = {}]] \implies finite (d \cup t)
```

you need apply -

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

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.



# Counterexample Finding

- Don't waste time trying to prove impossible statements!
- Isabelle can find counterexamples quickly...
  - *quickcheck*: random testing of executable specifications (broadly interpreted)
  - *nitpick*: a more general, SAT-based counterexample finder
- Consider switching on “auto quickcheck” or “auto nitpick”, although they can be slow!

# Nitpick Example

The screenshot shows the Isabelle Aquamacs interface. The main window displays the following code in `BT.thy`:

```
lemma reflect_reflect_ident: "reflect (reflect t) = t"  
  apply (induct t)  
  apply auto  
  done  
  
lemma "reflect t = t"
```

The status bar indicates the cursor is at line 17, column 1 in `BT.thy`. Below the code, the Nitpick output is shown in a yellow background:

```
Nitpicking formula...  
  
Nitpick found a counterexample for card 'a = 4:  
  
Free variable:  
  t = Br a1 (Br a2 Lf Lf) Lf|
```

The status bar at the bottom shows the cursor is at line 6, column 30 in the `*response*` window.

# Other Facets of Isabelle

- *Document preparation*: you can generate L<sup>A</sup>T<sub>E</sub>X 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  $+$   $-$   $\times$   $/$   $^$   $\leq$  and even  $\text{gcd}$
- 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.

# The End

*You know my methods. Apply them!*

Sherlock Holmes