

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 However, the *Tutorial* is very long and a little outdated; although its details remain correct, it presents a style of proof that has become increasingly obsolete with the advent of structured proofs and ever greater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
 - with precise definitions of concepts
 - and a formal deductive 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 λ -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: \emptyset , `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"
```

a new basic type

end-of-line comments

datatype

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

concrete syntax for commands

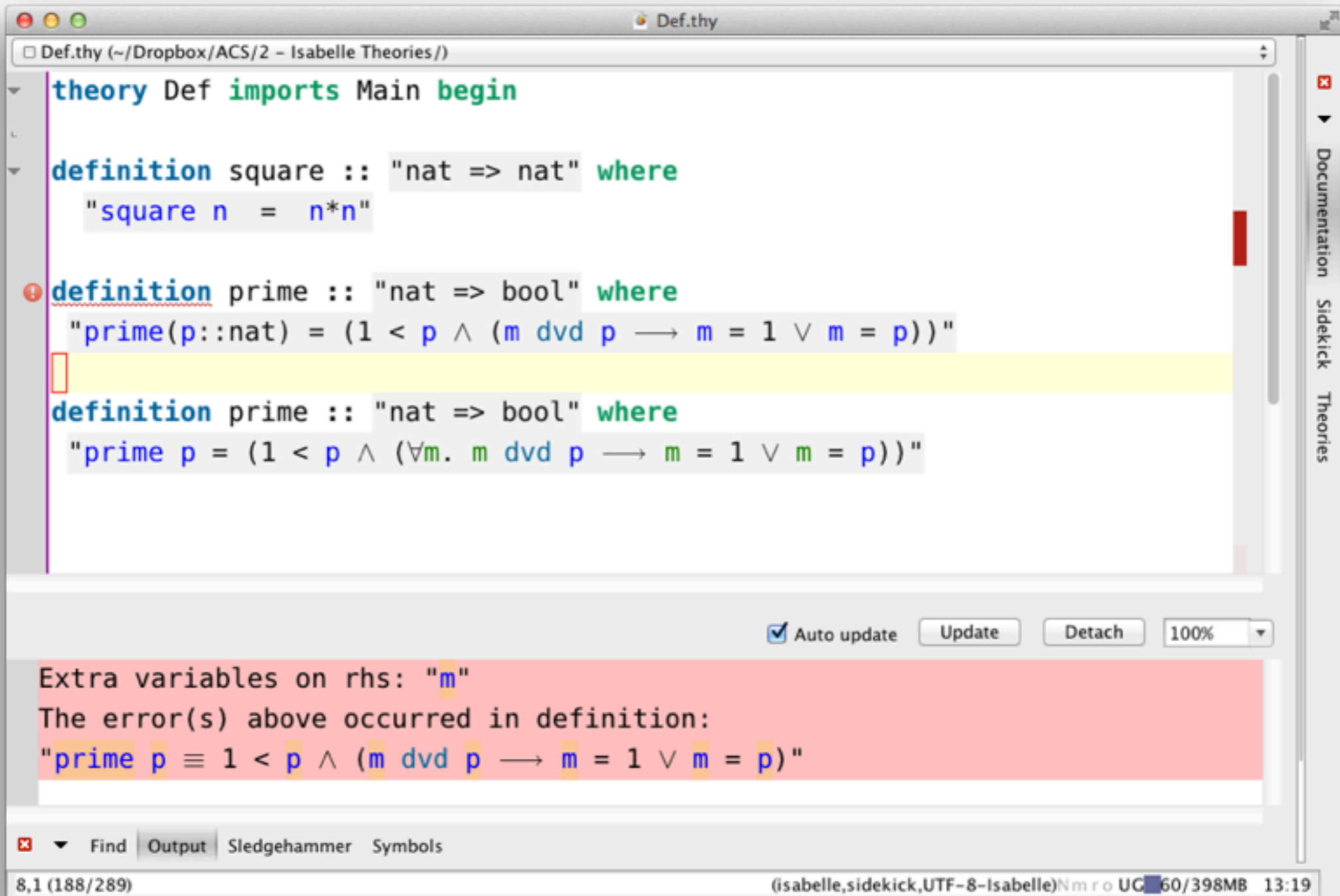
```
("_ ::= _" 60)
("_; _" [60, 60] 10)
("IF _ THEN _ ELSE _" 60)
("WHILE _ DO _" 60)
```

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



The screenshot shows the Isabelle/Isar IDE interface. The main window displays the following code in a file named `Def.thy`:

```
theory Def imports Main begin

definition square :: "nat => nat" where
  "square n = n*n"

definition prime :: "nat => bool" where
  "prime(p::nat) = (1 < p ∧ (m dvd p → m = 1 ∨ m = p))"

definition prime :: "nat => bool" where
  "prime p = (1 < p ∧ (∀m. m dvd p → m = 1 ∨ m = p))"
```

The second definition of `prime` is highlighted in yellow. Below the code, an error message is displayed in a red box:

```
Extra variables on rhs: "m"
The error(s) above occurred in definition:
"prime p ≡ 1 < p ∧ (m dvd p → m = 1 ∨ m = p)"
```

The IDE interface includes a sidebar on the right with options for Documentation, Sidekick, and Theories. At the bottom, there are tabs for Find, Output, Sledgehammer, and Symbols. The status bar at the bottom shows the version 8.1 (188/289) and the current file path.

See the *Tutorial*, Section 2.7.2 Constant Definitions.

Notes on Constant Definitions

- Basic definitions are *not* recursive.
- Every variable on the right-hand side must also appear on the left.
- In proofs, definitions are *not* expanded by default!
 - Defining the constant C to denote t yields the theorem C_def , asserting $C=t$.
 - Abbreviations can be declared through a separate mechanism.

Lists in Isabelle

- We illustrate data types and functions using a reduced Isabelle theory (one without 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!

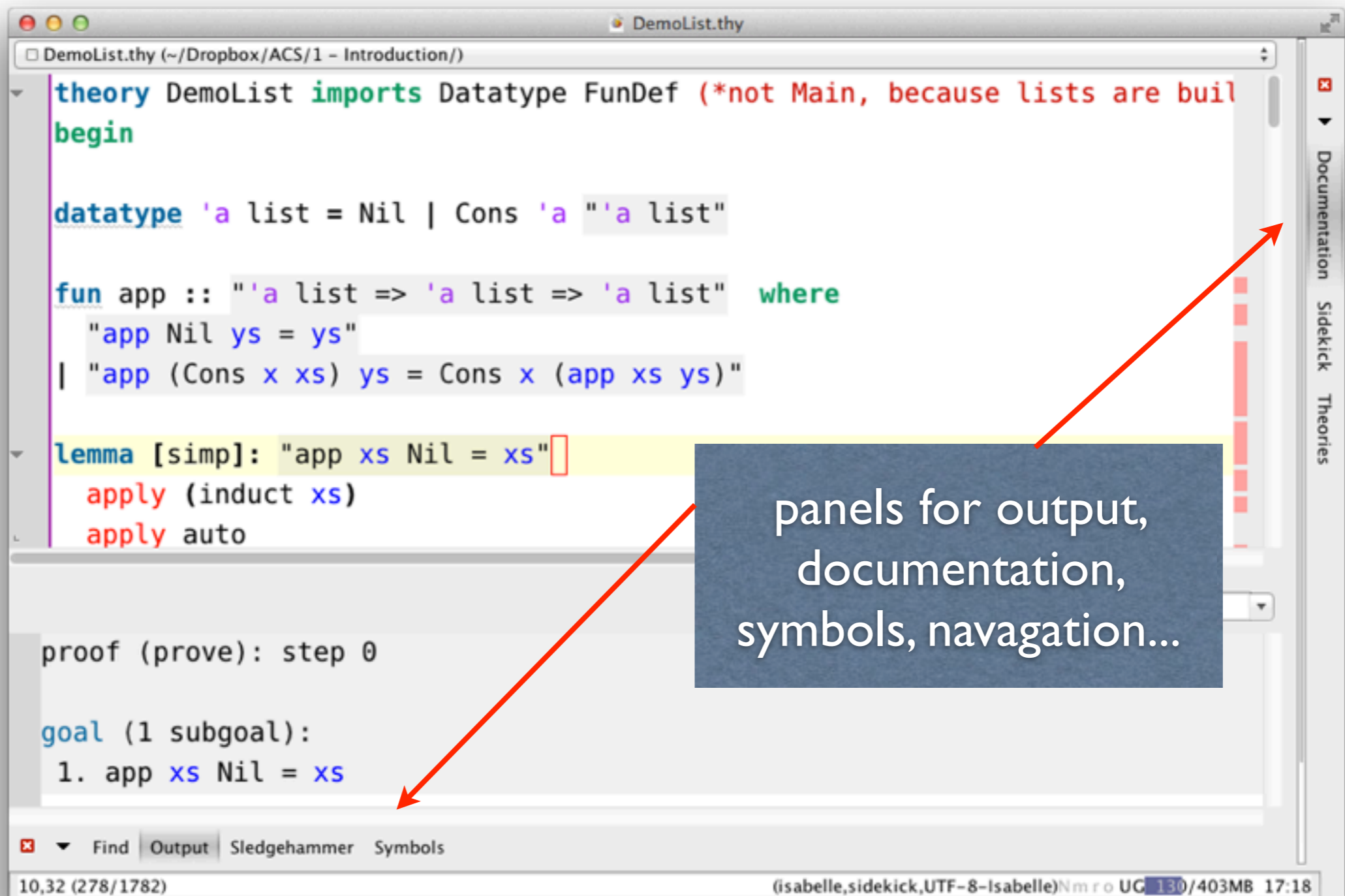
List Induction Principle

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 Nil or $\text{Cons}(x, xs)$ (for some x and xs).

Isabelle/jEdit

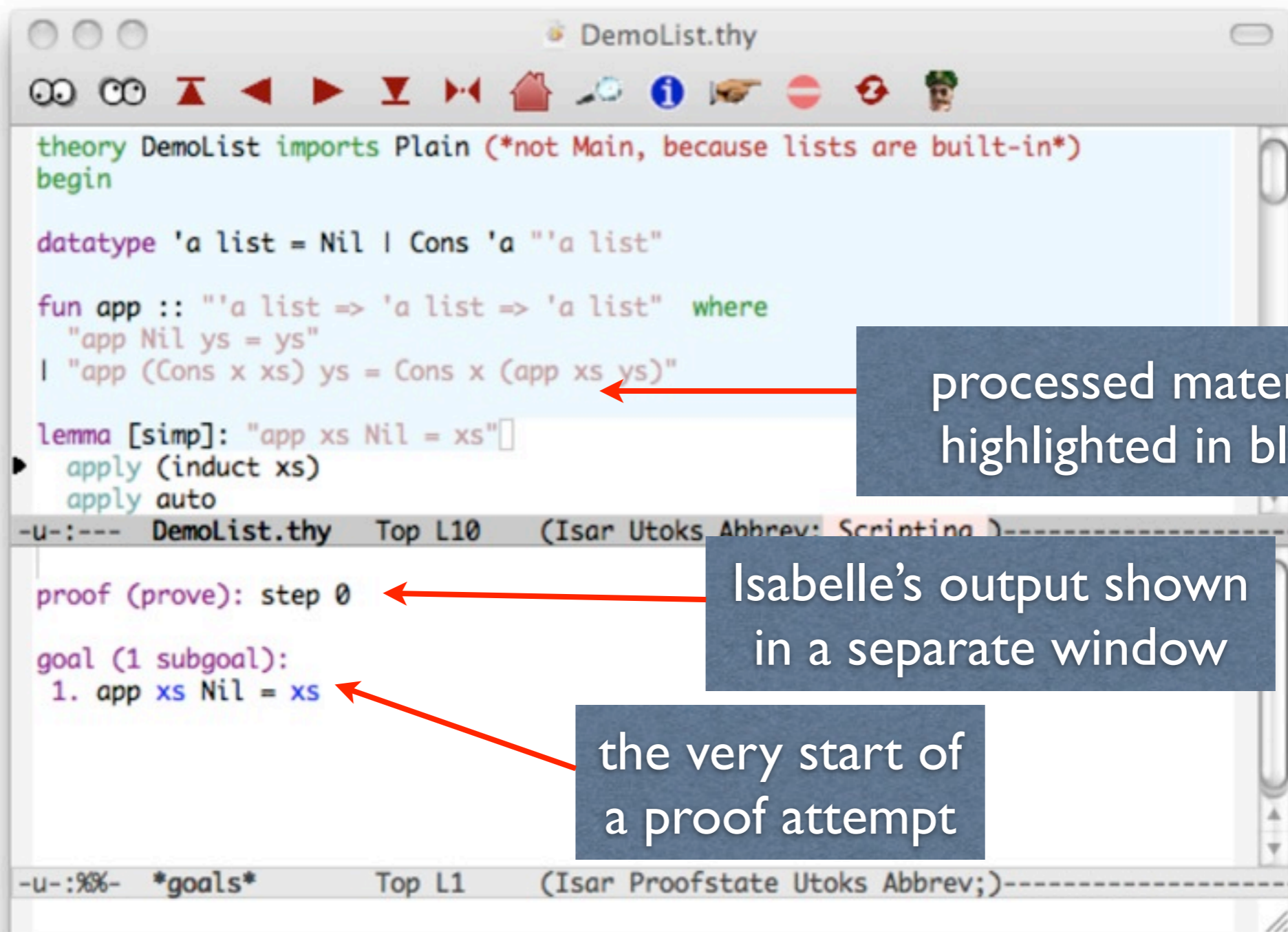


Isabelle's new user interface is the work of Makarius Wenzel. The entire proof document is processed as far as possible, errors and all. It allows inspection of proof states, identifier declarations, etc.

All documentation is accessible from the sidebar.

Launch using the command "isabelle jedit FILENAME"

Proof General



processed material
highlighted in blue

Isabelle's output shown
in a separate window

the very start of
a proof attempt

Isabelle's original 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

Launch using the command "isabelle emacs FILENAME"

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.  $\Lambda a xs. \text{app } xs \text{ Nil} = xs \implies \text{app } (\text{Cons } a \text{ xs}) \text{ Nil} = \text{Cons } a \text{ xs}$ 

```

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

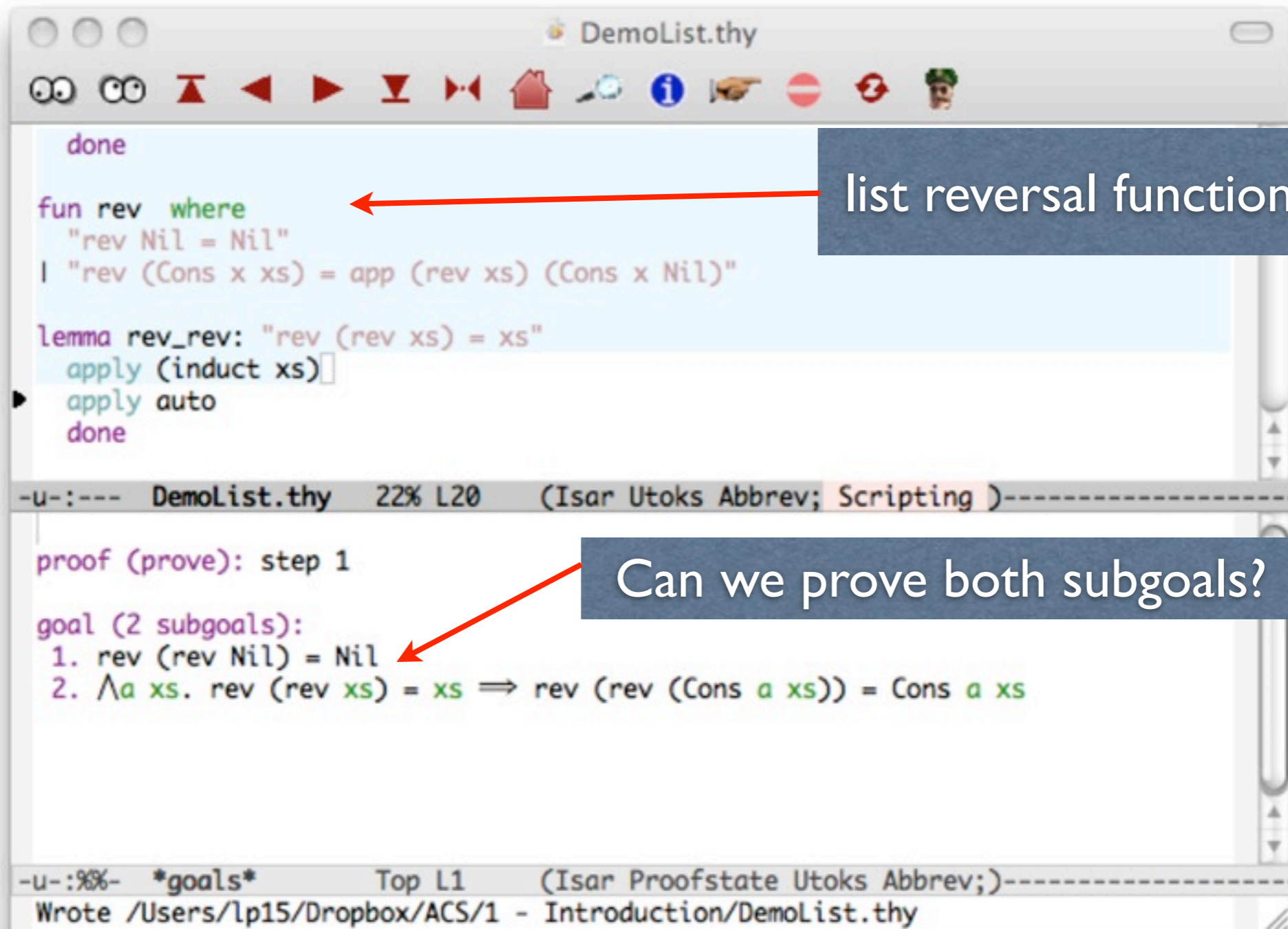
proof (prove): step 2
goal:
No subgoals!
```

auto proves both 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.

Another Proof Attempt



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

The proof state shows:

```
proof (prove): step 1

goal (2 subgoals):
  1. rev (rev Nil) = Nil
  2.  $\Lambda a xs. rev (rev xs) = xs \implies rev (rev (Cons a xs)) = Cons a xs$ 
```

Below the proof state is another status bar: `-u-:%%- *goals* Top L1 (Isar Proofstate Utoks Abbrev;)-----`

At the bottom, it says: `Wrote /Users/lp15/Dropbox/ACS/1 - Introduction/DemoList.thy`

Two red arrows point from text boxes to the code. The first arrow points from the text "list reversal function" to the `fun rev` definition. The second arrow points from the text "Can we prove both subgoals?" to the first subgoal in the goal list.

Stuck!

```
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

proof (prove): step 2

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

-u-:--- DemoList.thy 22% L22 (Isar Utoks Abbrev; Script
-u-:%%- *goals* Top L1 (Isar
tool-bar next
```

auto made progress
but didn't finish

looks like we need a lemma
relating rev and app!

Stuck Again!

```
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)

-u:--- DemoList.thy 21% L24 (Isar Utoks Abbrev; Scripting )-----

proof (prove): step 2

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

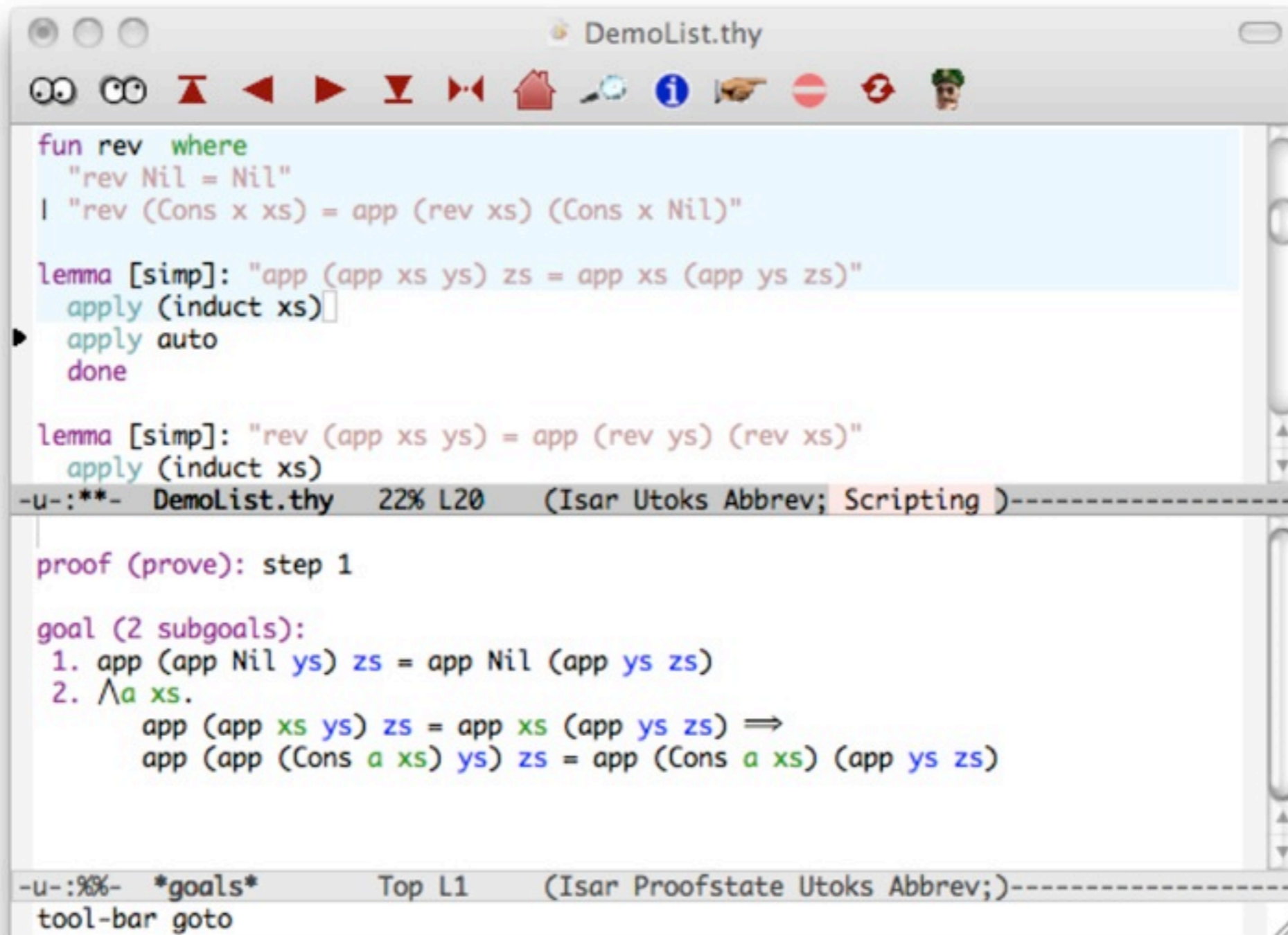
-u:%%- *goals* Top L1 (Isar Proofstate Utoks Abbrev;)-----
Wrote /Users/lp15/Dropbox/ACS/1 - Introduction/DemoList.thy
```

we dreamt up a lemma...

But it needs another lemma!
(Generalising this subgoal)

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 )-----

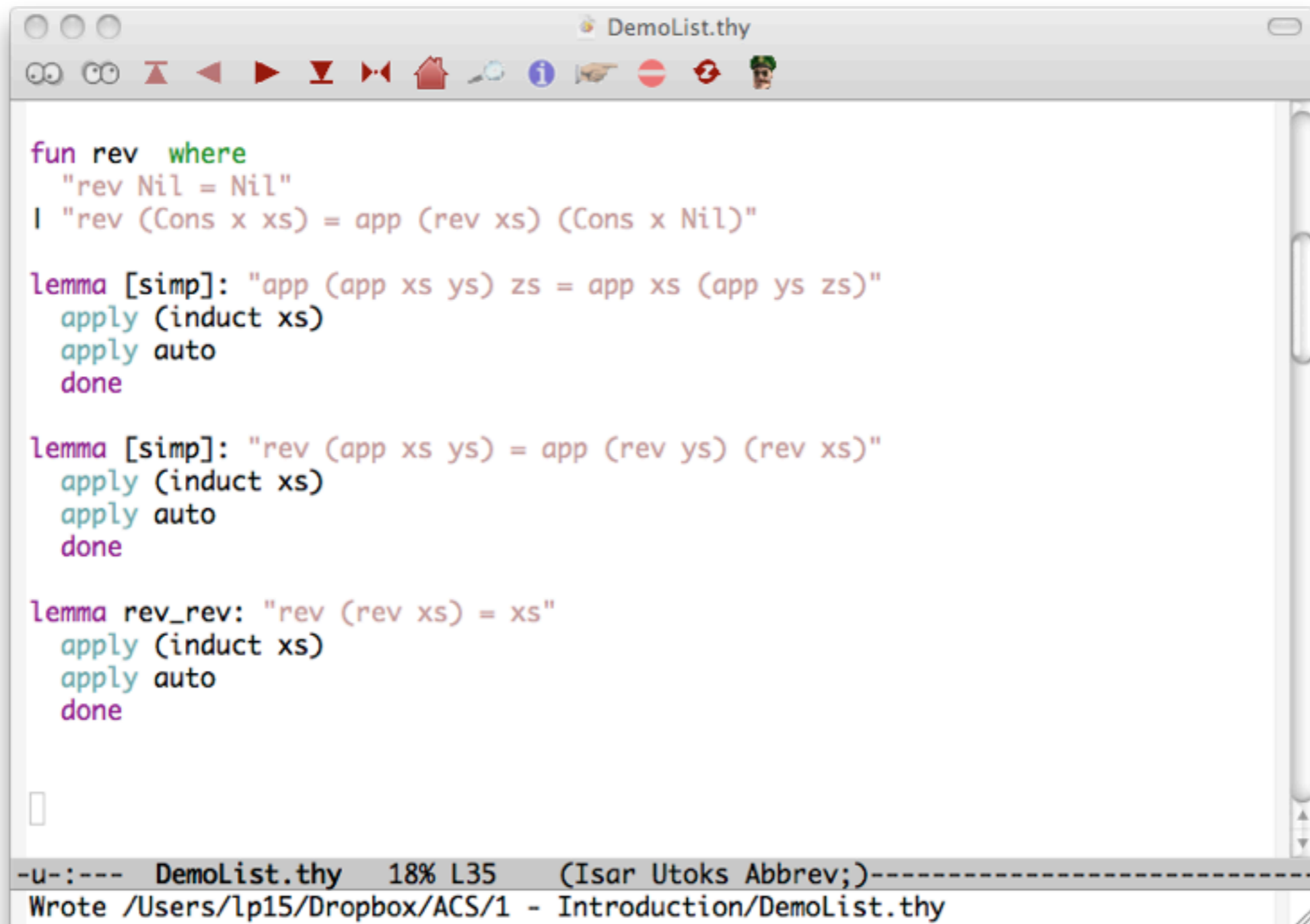
proof (prove): step 1

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)  $\implies$ 
   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

```
BT.thy (~/.Dropbox/ACS/2 - Isabelle Theories /)
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
```

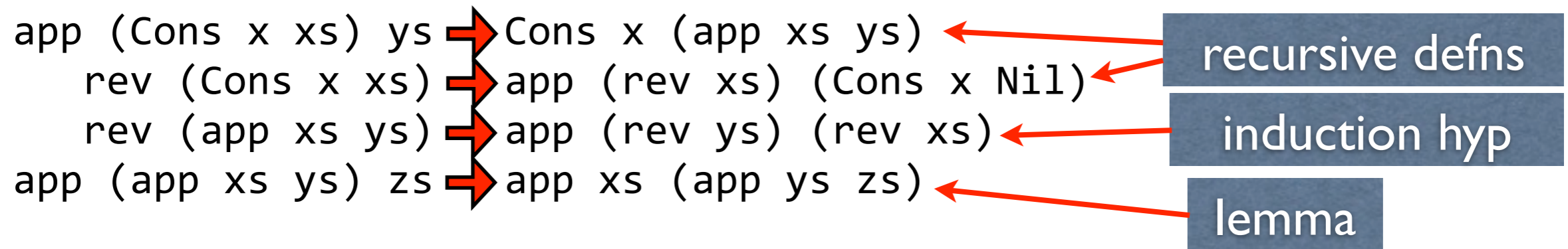
1. reflect (reflect Lf) = Lf
2. $\wedge a\ t1\ t2.$
 reflect (reflect t1) = t1 \implies
 reflect (reflect t2) = t2 \implies
 reflect (reflect (Br a t1 t2)) = Br a t1 t2

assumptions (two induction hypotheses)

parameters (arbitrary local variables)

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. But assumptions can also be split.
- Other kinds of 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: ...`

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

ignoring all assumptions

do `simp` for all subgoals

auto with options

Rules for Arithmetic

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

Example: `auto simp add: field_simps`

Basics of 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 Isabelle/jEdit

- Based on the jEdit text editor.
- Isabelle automatically processes the entire visible window, errors and all, using parallel threads.
- Identifiers and other elements can be *inspected* using hover-click.
- *Dockable panels* give access to the Isabelle output, theory structure, manuals, symbols, etc.

A View of Isabelle/jEdit

The screenshot shows the Isabelle/jEdit interface with several key components highlighted by blue callout boxes and red arrows:

- documentation panel**: A vertical sidebar on the right containing a tree view of documentation files, including 'Tutorials', 'Reference Manuals', and 'Examples'.
- more panels**: A label pointing to the top-right corner of the interface, indicating the presence of additional panels.
- popup inspector**: A small yellow box that appears over the code, displaying the type signature of a free variable: `free variable :: nat`.
- output panel**: A panel at the bottom of the window showing the current proof state, including the goal and its subgoals.

The main editor window displays the following code:

```
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

proof (prove): step 1
goal (3 subgoals):
1.  $\bigwedge n. n < \text{ack } 0 \ n$ 
2.  $\bigwedge m. 1 < \text{ack } m \ 1 \implies 0 < \text{ack } (\text{Suc } m) \ 0$ 
3.  $\bigwedge m \ n. n < \text{ack } (\text{Suc } m) \ n \implies$ 
```

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

```
fun itlen :: "'a list => nat => nat"
  "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
```

length of a list
(tail-recursive)

equivalent to the built-in length function?

May as well give up!

Mismatch between induction hypothesis and conclusion!

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

Generalising the Induction

The screenshot shows the Isabelle theorem prover interface with a file named `DemoList.thy`. The code defines a function `itlen` and a lemma. A blue box with the text "Insert a universal quantifier" has a red arrow pointing to the `itlen` function definition. A yellow highlight is under the `apply (induct xs)` line. A second blue box with the text "Induction hypothesis holds for all n" has a red arrow pointing to the induction hypothesis in the goal.

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

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

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
```

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: A Better Way

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

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

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

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

Two variables in the induction!

Two variables in the recursion!

A special induction rule!

The subgoals follow the recursion!

```
Primrec.thy
kermann's Func
fun ack :: "nat => nat => nat" where
  "ack 0 n = Suc n"
| "ack (Suc m) 0 = ack m 1"
| "ack (Suc m) (Suc n) = ack m (ack (Suc m) n)"

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

-u-:--- Primrec.thy 3% L16

proof (prove): step 1

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

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

Recursion: Key Points

- Recursion 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

The screenshot shows the Isabelle/HOL editor with the following content:

```
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
```

Annotations:

- A blue box with the text "recursive calls, guarded by conditions" has two red arrows pointing to the recursive call `merge xs (y#ys)` and `merge (x#xs) ys` in the function definition.
- A blue box with the text "2 induction hypotheses, guarded by conditions!" has a red arrow pointing to the first induction hypothesis in the proof.

The proof steps are:

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

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

The screenshot shows the Isabelle/Proof Assistant interface. At the top, a window titled 'MergeSort.thy' contains the following code:

```
lemma ordered_merge [simp]: "ordered (merge xs ys) =  
  ordered xs ^ (case ys of [] => True | ya # xs => y <= ya ^ ordered (ya # xs))" =>  
  apply (induct xs ys rule: merge.induct)  
  apply simp_all  
  apply (auto split: list.split)
```

A blue callout box with white text says "Automatic case splitting to the rescue!" with a red arrow pointing to the `split: list.split` argument in the `apply` block.

Below the code, the proof progress is shown. The first goal is expanded:

```
1.  $\wedge x \ xs \ y \ ys.$   
   $(x \leq y \implies$   
     $ordered \ (merge \ xs \ (y \ # \ ys)) =$   
     $ordered \ xs \ \wedge \ (case \ ys \ of \ [] \ \implies \ True \ | \ ya \ # \ xs \ \implies \ y \leq \ ya \ \wedge \ ordered \ (ya \ # \ xs))) \implies$   
   $x \leq y \implies$   
   $ordered \ (merge \ (x \ # \ xs) \ ys) =$   
   $(case \ xs \ of \ [] \ \implies \ True \ | \ y \ # \ xs \ \implies \ x \leq \ y \ \wedge \ ordered \ (y \ # \ xs)) \ \wedge \ ordered \ ys) \implies$   
   $(x \leq y \implies$   
   $(case \ merge \ xs \ (y \ # \ ys) \ of \ [] \ \implies \ True \ | \ y \ # \ xs \ \implies \ x \leq \ y \ \wedge \ ordered \ (y \ # \ xs)) =$   
   $(case \ xs \ of \ [] \ \implies \ True \ | \ y \ # \ xs \ \implies \ x \leq \ y \ \wedge \ ordered \ (y \ # \ xs)) \ \wedge$   
   $(case \ ys \ of \ [] \ \implies \ True \ | \ ya \ # \ xs \ \implies \ y \leq \ ya \ \wedge \ ordered \ (ya \ # \ xs)))) \ \wedge$   
   $(\neg \ x \leq \ y \implies$   
   $(case \ merge \ (x \ # \ xs) \ ys \ of \ [] \ \implies \ True \ | \ ya \ # \ xs \ \implies \ y \leq \ ya \ \wedge \ ordered \ (ya \ # \ xs)) =$   
   $((case \ xs \ of \ [] \ \implies \ True \ | \ y \ # \ xs \ \implies \ x \leq \ y \ \wedge \ ordered \ (y \ # \ xs)) \ \wedge$   
   $(case \ ys \ of \ [] \ \implies \ True \ | \ ya \ # \ xs \ \implies \ y \leq \ ya \ \wedge \ ordered \ (ya \ # \ xs))))$ 
```

Red arrows point from a blue callout box on the left, which says "Help! Look at all the case-splits!", to the various `case` expressions in the goal.

At the bottom of the window, the status bar shows "29,16 (666/1040)" and "(isabelle,sidekick,UTF-8-Isabelle)Nr o UG 227/401MB 14:42".

Help! Look at all the case-splits!

Automatic case splitting to the rescue!

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 proof editor window titled "MergeSort.thy". The editor contains the following code:

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

The last line of code is highlighted in yellow. Below the code editor, the proof progress is shown:

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

Two callout boxes with arrows point to the code and the proof status:

- A box pointing to the `apply (auto split: list.split) □` line contains the text: "But what is this? Risk of looping!"
- A box pointing to the "No subgoals!" line contains the text: "All solved, in <1 second."

The bottom status bar shows "30,31 (697/1040)" and "(isabelle,sidekick,UTF-8-Isabelle)Nm r o UG 230/401MB 14:44".

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

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

by

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

- It creates a case for each datatype constructor.
- Here it causes the simplifier to loop 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!

How to Trace the Simplifier

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

fun msort :: "'a list => 'a list"
```

Auto update Update Detach 95%

```
x ≤ a ∧ ordered (a # list)
[1]Applying instance of rewrite rule "??.unknown":
ordered [] ≡ True
[1]Rewriting:
ordered [] ≡ True
[1]Applying instance of rewrite rule "HOL.simp_thms_21":
?y ∧ True ≡ ?y
[1]Rewriting:
(x ≤ a ∧ ordered (a # list)) ∧ True ≡ x ≤ a ∧ ordered (a # list)
Tracing paused. Stop, or continue with next 100, 1000, 10000 messages?
```

trace appears in output panel

tracing is very slow

Find Output Symbols

31,1 (689/1062) (isabelle,sidekick,UTF-8-Isabelle)Nr o UG 140/397MB 14:56

Interactive Formal Verification

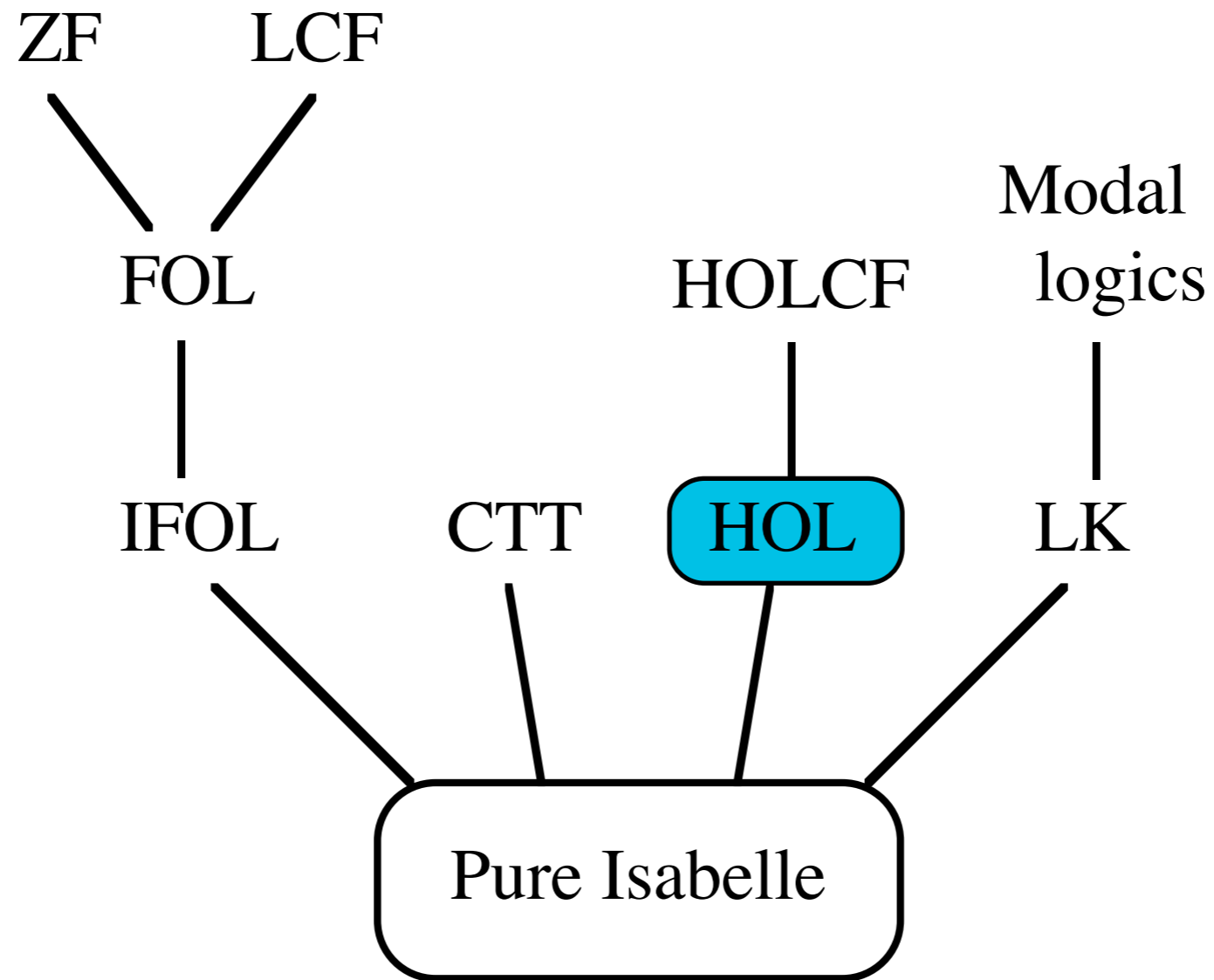
5: Logic in Isabelle

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

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

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

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

$$P \wedge Q \Rightarrow P$$

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

$$P \wedge Q \Rightarrow Q$$

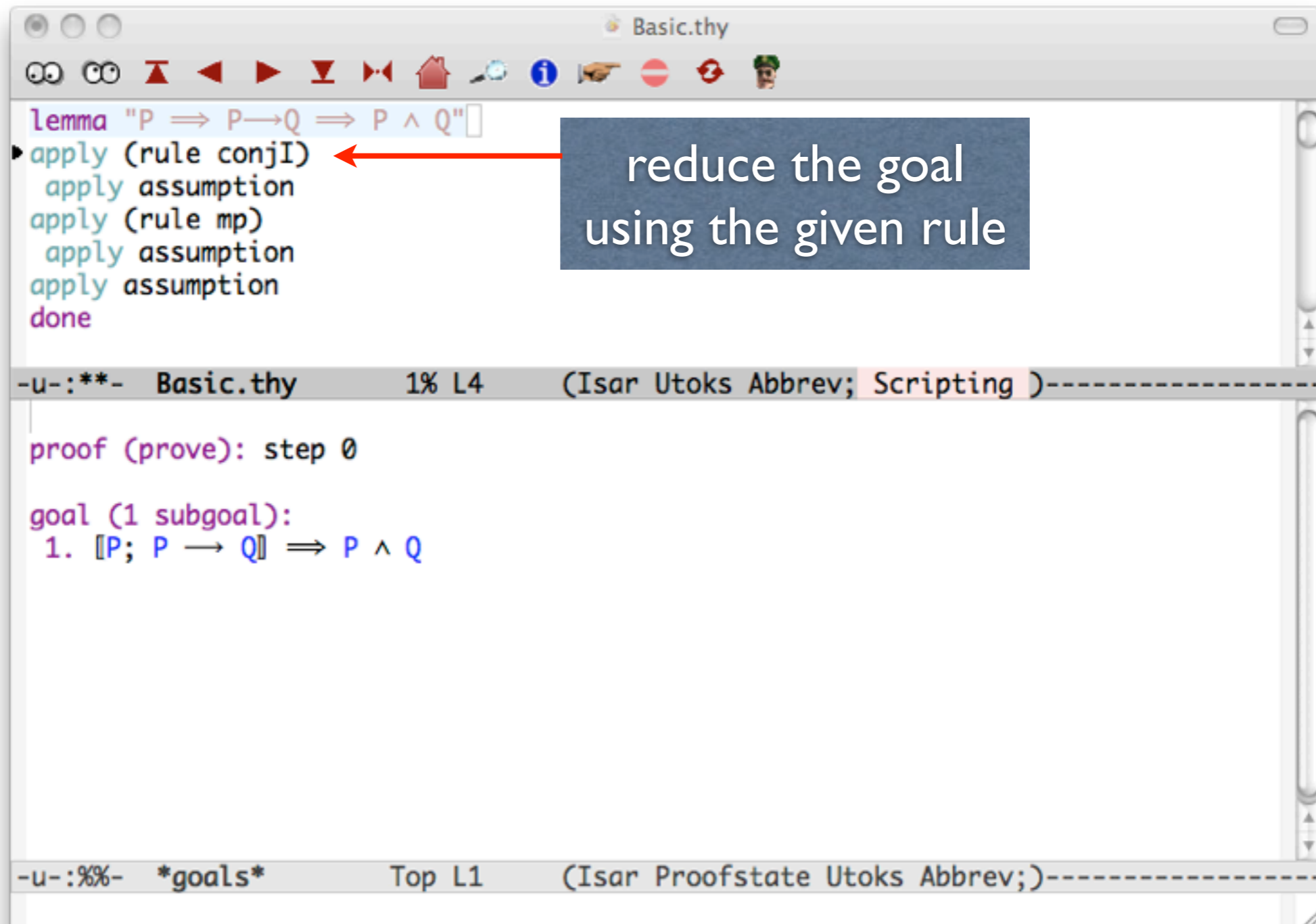
$$\frac{P \rightarrow Q \quad P}{Q}$$

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

Meta-implication

- The symbol \Rightarrow (or \implies) 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 $\llbracket P ; Q \rrbracket \Rightarrow R$

A Trivial Proof



The screenshot shows the Isabelle IDE interface. The top window displays the proof script for a lemma. A red arrow points from a text box to the `apply (rule conjI)` line. The bottom window shows the current state of the proof, with a goal that matches the lemma's conclusion.

```
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):
```

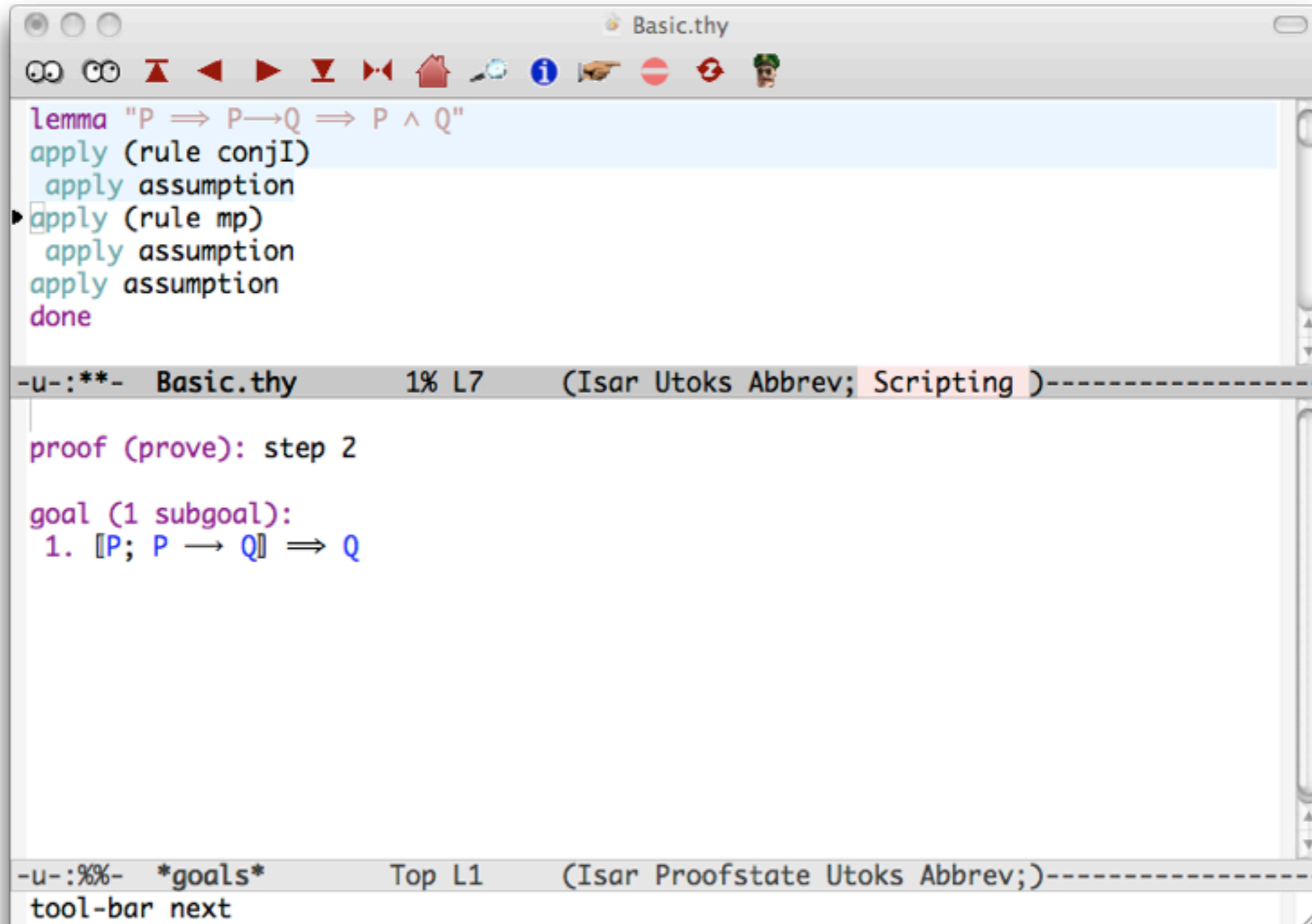
```
1. [[P; P  $\longrightarrow$  Q]]  $\implies$  P  $\wedge$  Q
```

```
-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 )-----

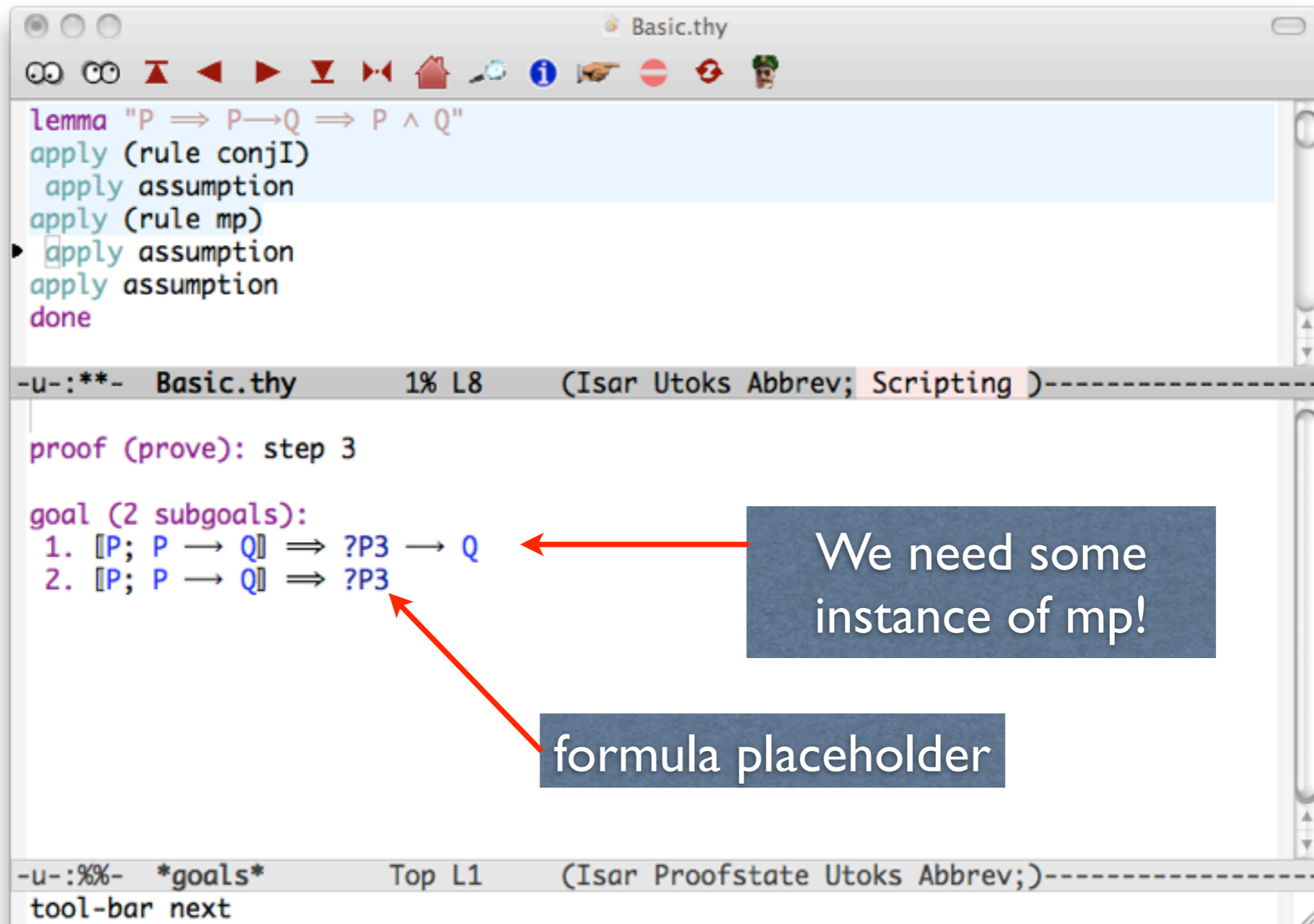
proof (prove): step 2

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

Unknowns and Unification

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

```
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 code is a status bar with the text: "-u-:***- Basic.thy 1% L9 (Isar Utoks Abbrev; Scripting)".

The lower part of the window shows a proof state:

```
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 goal statement. The status bar at the bottom of the window reads: "-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

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

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

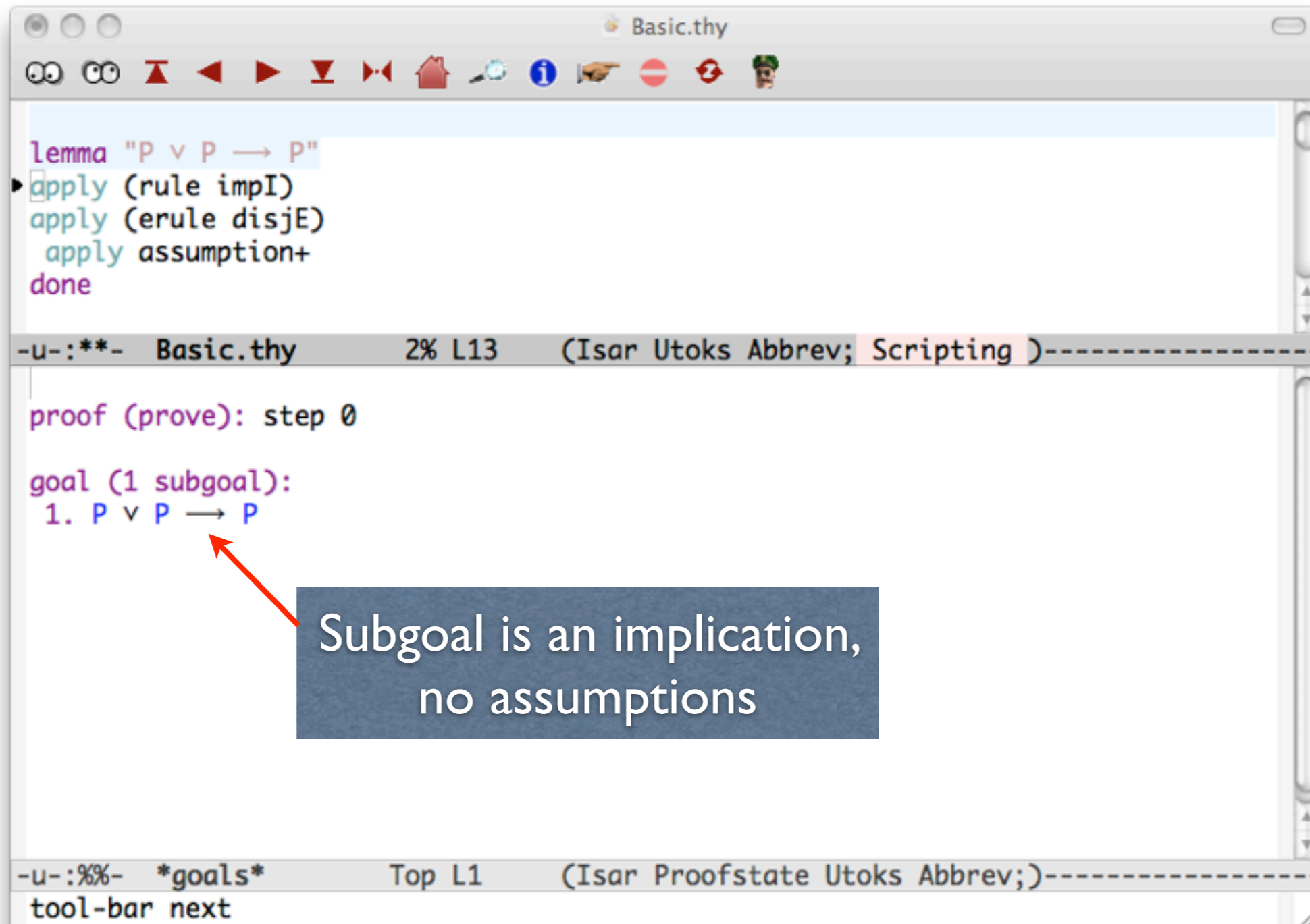
$$\frac{P \vee Q \quad \begin{array}{c} [P] \\ \vdots \\ R \end{array} \quad \begin{array}{c} [Q] \\ \vdots \\ R \end{array}}{R}$$

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

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**

-u-:%%- *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 `erule disjE` line to a dark blue callout box containing the text: "erule is good with elimination rules".

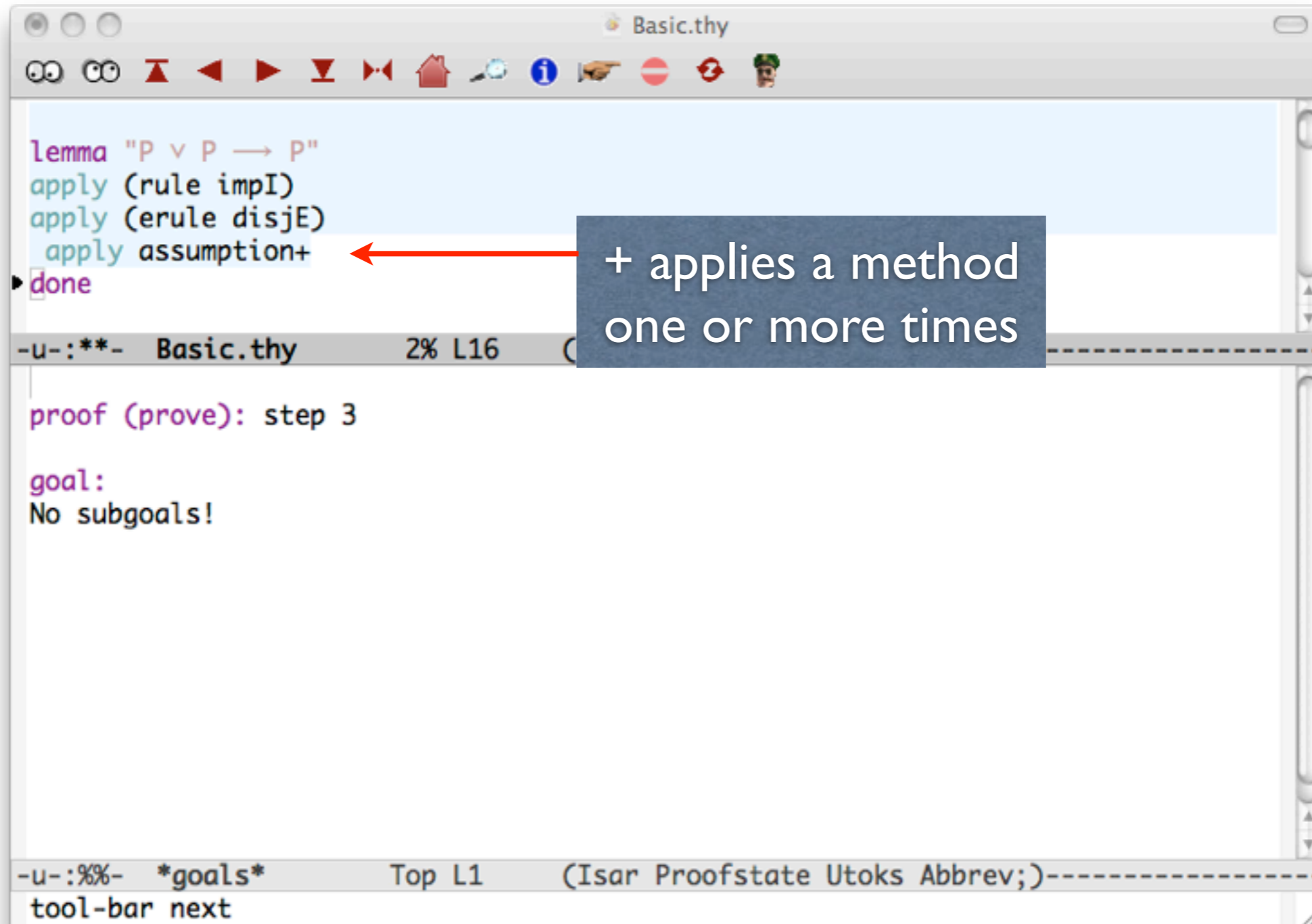
The bottom panel shows the goal state after the `erule disjE` command:

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

A red arrow points from the first goal `1. P ⇒ P` to another dark blue callout box containing the text: "An instance of ?P ∨ ?Q has been found".

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



The screenshot shows a theorem prover interface with a toolbar at the top and a main text area. The text area contains a lemma definition and a proof step. A red arrow points from a callout box to the '+' symbol in the 'apply assumption+' line of the lemma. The callout box contains the text '+ applies a method one or more times'. The proof step below shows 'proof (prove): step 3' and 'goal: No subgoals!'. The status bar at the bottom indicates the current goal and tool-bar settings.

```
lemma "P ∨ P → P"  
  apply (rule impI)  
  apply (erule disjE)  
  apply assumption+  
done  
proof (prove): step 3  
goal:  
No subgoals!
```

+ applies a method
one or more times

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

Quantifiers

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

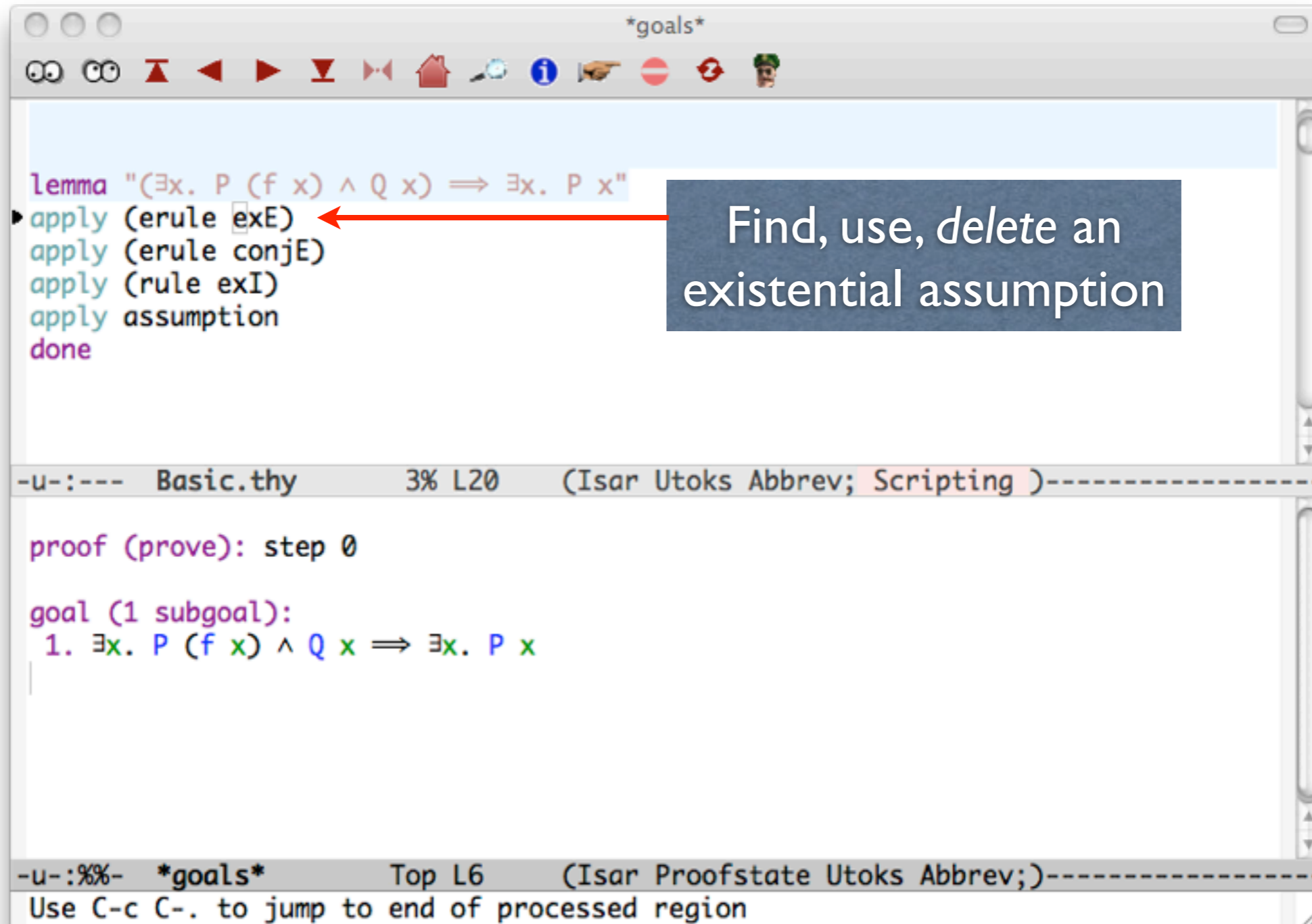
$$P(x) \Rightarrow \exists x. P(x)$$

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

$$\llbracket \exists x. P(x); \forall x. P(x) \Rightarrow Q \rrbracket \Rightarrow Q$$

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

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

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

proof (prove): step 2

goal (1 subgoal):
1.  $\wedge x. [P (f x); Q x] \Rightarrow \exists x. P x$ 

-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”

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

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

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

Below the code, the proof progress is shown:

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

Annotations in the image:

- A blue box with the text "We unfold the definition and get...?" has a red arrow pointing to the `auto simp add: dvd_def` line in the proof script.
- A blue box with the text "an assumption" has a red arrow pointing to the `ka` variable in the first subgoal.
- A blue box with the text "locally bound variables" has a red arrow pointing to the `ka` and `kb` variables in the second subgoal.
- A blue box with the text "A messy proof with two subgoals..." is positioned to the right of the subgoals.

The IDE interface includes a top bar with "Struct.thy (modified)", a left sidebar with "Documentation", "Sidekick", and "Theories", and a bottom status bar showing "94,31 (1685/3943)" and "(isabelle,sidekick,UTF-8-Isabelle)Nm r o UC 211/31 MB 23:01".

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

```
lemma "(k dvd (n + k)) = (k dvd (n::nat))"
proof (simp only: dvd_def, safe)
  fix m
  assume "n + k = k * m"
  then have "n = k * (m - 1)"
    by (metis diff_add_inverse diff_mult_distrib2 nat_add_commute nat_mult_1_right)
  then show "∃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
```

using this:
n + k = k * m

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

But how do you write them?

8,30 (182/3943) (isabelle,sidekick,UTF-8-Isabelle)N m r o UG 74/278MB 23:05

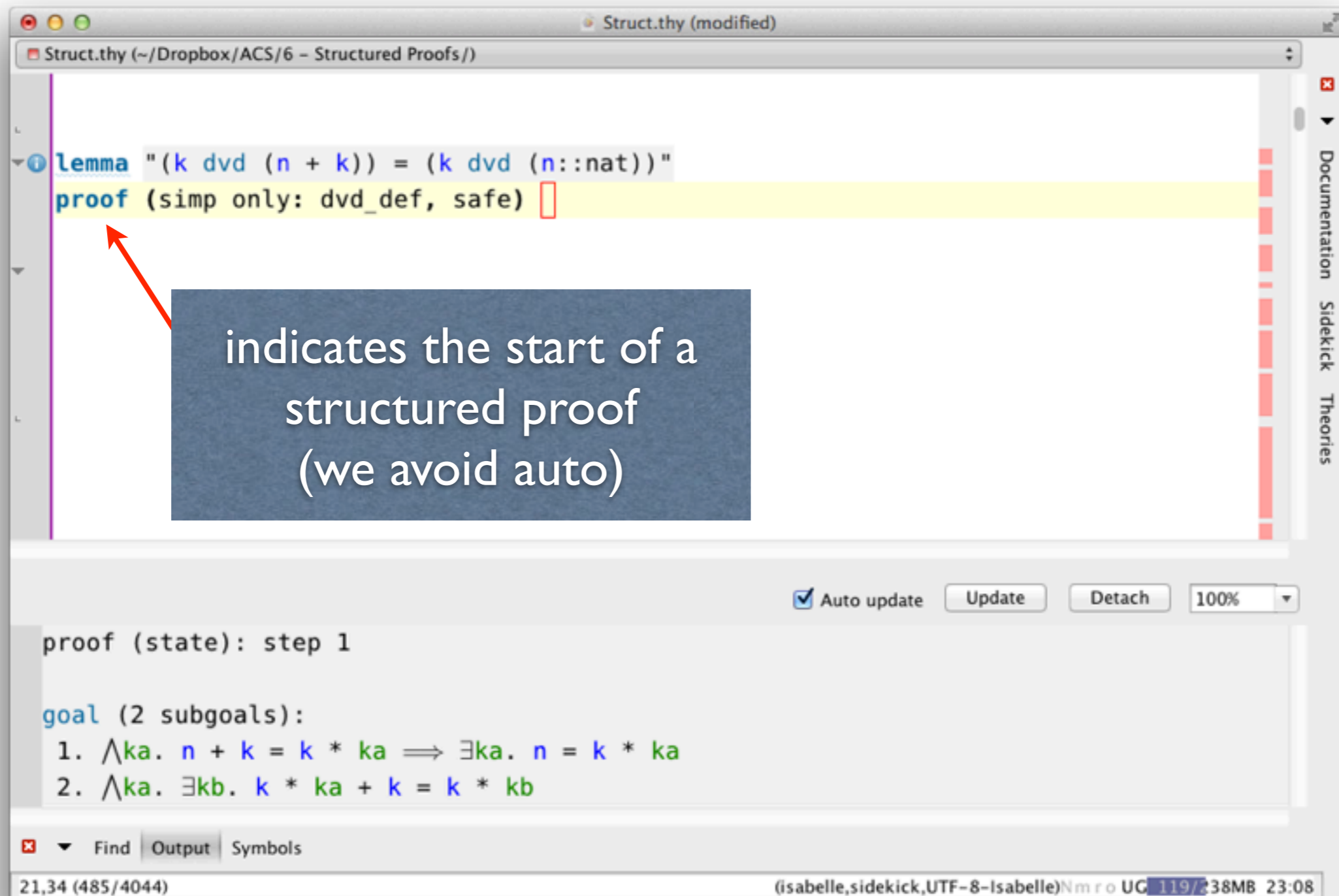
The Elements of Isar

- 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



```
Struct.thy (modified)
Struct.thy (~/.Dropbox/ACS/6 - Structured Proofs/)

lemma "(k dvd (n + k)) = (k dvd (n::nat))"
proof (simp only: dvd_def, safe) []

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

indicates the start of a structured proof (we avoid auto)

Auto update Update Detach 100%

Find Output Symbols

21,34 (485/4044) (isabelle,sidekick,UTF-8-Isabelle)Nm r o UC 119/238MB 23:08

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.

Replacing “auto” by “simp only: dvd_def, safe” produces a more robust proof, since these methods are much simpler and more stable than auto.

The Proof Skeleton

```
lemma "(k dvd (n + k)) = (k dvd (n::nat))"
proof (simp only: dvd_def, safe)
  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
```

assumption

conclusion

dummy proofs

a name for the bound variable

separates proofs of goals

terminates the proof

show $\exists m'. k * m + k = k * m'$
Successful attempt to solve goal by exported rule:
 $\exists m'. k * ?m2 + k = k * m'$
proof (state): step 9

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

```
Struct.thy (~/.Dropbox/ACS/6 - Structured Proofs/)  
lemma "(k dvd (n + k)) = (k dvd (n::nat))"  
proof (simp only: dvd_def, safe)  
  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
```

Labels for facts: assume 1, have 2

inserting a helpful fact: sorry

... and using it, a real proof: by blast

Output:
show ∃m'. k * m + k = k * m'
Successful attempt to solve goal by exported rule:
∃m'. k * ?m2 + k = k * m'
proof (state): step 13

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

The screenshot shows a theorem prover interface with a file named `Struct.thy (modified)`. The main window displays the following code:

```
lemma "(k dvd (n + k)) = (k dvd (n::nat))"
proof (simp only: dvd_def, safe)
  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
```

Annotations in the image include:

- A red arrow pointing to the line `show "∃m'. n = k * m'" using 2` with a text box: "This line came from sledgehammer".
- A red arrow pointing to the line `show "∃m'. k * m + k = k * m'"` with a text box: "sledgehammer does it again!".

Below the main window, a `proof (prove): step 8` section shows:

```
using this:
  n = k * (m - 1)
```

At the bottom, a `Sledgehammer` window is open, showing the following provers and their results:

```
Provers: e spass remote_vampire z3 remote_e_sine
[e] Try this: by (metis comm_semiring_1_class.normalizing_semiring_rules(24))
[spass] Try this: by (metis comm_semiring_1_class.normalizing_semiring_rules(
[remote_vampire] Try this: by (metis mult_Suc_right nat_add_commute) (7 ms).
[z3] Timed out.
[remote_e_sine] Try this: by (metis mult_Suc_right nat_add_commute) (10 ms).
```

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

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

Streamlining the Proof

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

→

```
assume "n + k = k * m"  
then have "n = k * (m - 1)"  
sorry  
then show "∃m'. n = k * m'"
```

using the previous fact without mentioning labels

- then have *or* hence
- then show *or* thus

There are numerous other tricks of this sort!

Another Proof Skeleton

The screenshot shows the Isabelle/Isar proof editor interface. The main window displays the following code for a lemma:

```
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
  then show "abs m = 1" using 0
    by auto
qed
```

Annotations with arrows point to specific parts of the code:

- specify m's type**: points to `fixes m :: int`
- declare a premise separately**: points to `assumes mn: "abs (m * n) = 1"`
- (-) is the null proof step**: points to `proof -`
- steps towards the result**: points to `have 0: "m ≠ 0" using mn by auto` and `have "~ (2 ≤ abs m)" sorry`
- now the conclusion is trivial**: points to `then show "abs m = 1" using 0 by auto`

The bottom panel shows the output of the `show |m| = 1` command:

```
show |m| = 1
Successful attempt to solve goal by exported rule:
  |m| = 1
proof (state): step 10
```

The status bar at the bottom indicates the current session information: `(isabelle,sidekick,UTF-8-Isabelle)Nm r o UG 106/199MB 14:06`.

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 displays the Isabelle/Proof General interface. The main editor shows a lemma definition:

```
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
  then show "abs m = 1" using 0
  by auto
qed
```

Annotations in the image:

- A purple box with the text "the default proof step (depends on the goal)" has an arrow pointing to the `proof` block within the `proof -` block.
- A blue box with the text "error symbols (syntax)" has arrows pointing to the red error icons on the left margin of the `then show` and `qed` lines.

The bottom panel shows the current state of the proof:

```
proof (state): step 6
goal (1 subgoal):
  1. 2 ≤ |m| ⇒ False
```

The status bar at the bottom indicates: 128,10 (1968/4043) Input/output complete (isabelle,sidekick,UTF-8-Isabelle)Nm r o UG 102/188MB 14:10

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

```
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"
      then show "False"
        sorry
    qed
  then show "abs m = 1" using 0
    by auto
qed
```

The screenshot shows a theorem prover interface with a file named 'Struct.thy'. The main window displays a lemma 'abs_m_1' with a nested proof structure. The code is color-coded: 'lemma' is blue, 'fixes' is green, 'assumes' is green, 'shows' is green, 'proof' is blue, 'have' is blue, 'by' is blue, 'assume' is blue, 'then show' is blue, 'sorry' is red, and 'qed' is blue. Two red arrows point from blue boxes labeled 'assumption' and 'conclusion' to the 'assume' and 'then show' lines, respectively. The interface also includes a sidebar with 'Documentation', 'Sidekick', and 'Theories' buttons, and a bottom panel with 'Auto update', 'Update', 'Detach', and '100%' buttons. The status bar at the bottom shows '137,17 (2053/4043)', 'Input/output complete', and '(isabelle,sidekick,UTF-8-Isabelle)Nm r o UG 97/15 SMB 14:15'.

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 θ are visible within this inner scope.

A Complete Proof

```
Struct.thy (~/.Dropbox/ACS/6 - Structured Proofs/)
```

```
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"  
    then have "2 * abs n ≤ abs m * abs n"  
      by (simp add: mult_mono)  
    then have "2 * abs n ≤ abs (m*n)"  
      by (simp add: abs_mult)  
    then have "2 * abs n ≤ 1"  
      by (auto simp add: mn)  
    then show "False" using 0  
      by auto  
  qed  
  then show "abs m = 1" using 0  
    by auto  
qed
```

a chain of steps leads to contradiction

such chains can be done as *calculations*

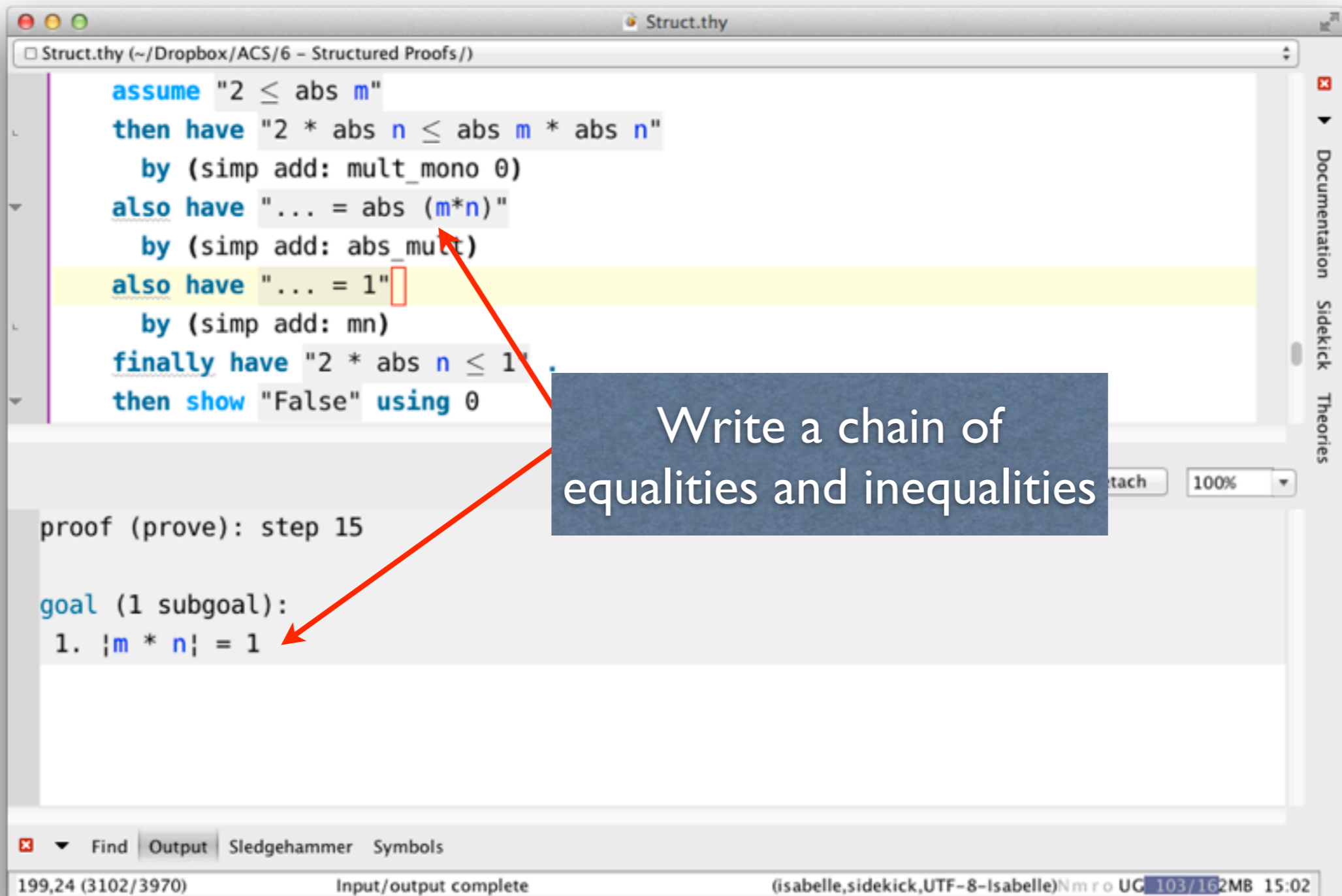
Find Output Sledgehammer Symbols

157,14 (2331/4043) Input/output complete (isabelle,sidekick,UTF-8-Isabelle)Nm r o UG 71/148MB 14:16

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



```
Struct.thy (~/.Dropbox/ACS/6 - Structured Proofs/)  
  
assume "2 ≤ abs m"  
then have "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"  
then show "False" using 0  
  
proof (prove): step 15  
goal (1 subgoal):  
1. |m * n| = 1
```

Write a chain 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

```
Struct.thy (~/.Dropbox/ACS/6 - Structured Proofs/)
```

```
assume "2 ≤ abs m"
then have "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"
  by (simp add: ...)
then show "False" using 0
```

proof (prove): step 12

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

... refers to the previous right-hand side

197,33 (3048/3970) (isabelle,sidekick,UTF-8-Isabelle)N m r o UG 88/162MB 15:02

The Internal Calculation

The screenshot shows the Isabelle/Isar proof editor interface. The top pane displays a proof script in Isar format:

```
assume "2 ≤ abs m"  
then have "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" .  
then show "False" using 0
```

Red arrows point from a blue box on the left labeled "structure of a calculation" to the `then have`, `also have`, `also have`, and `finally have` lines. The `finally have` line is highlighted in yellow.

The bottom pane shows the internal calculation chain:

```
calculation: 2 * |n| ≤ 1  
proof (chain) step 17  
picking this:  
  2 * |n| ≤ 1
```

A red arrow points from the `proof (chain)` line to a blue box on the right labeled "The calculation is displayed for also and finally".

At the bottom of the editor, there are tabs for "Find", "Output", "Sledgehammer", and "Symbols". The status bar at the bottom shows "(isabelle,sidekick,UTF-8-Isabelle)Nm r o UG 103/163MB 15:03".

structure
of a
calculation

The calculation is displayed for
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

```
Struct.thy (~/.Dropbox/ACS/6 - Structured Proofs/)
```

```
assume "2 ≤ abs m"
then have "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" .
then show "False" using 0
```

(.) denotes a trivial proof

```
have 2 * |n| ≤ 1
proof (state): step 19
this:
  2 * |n| ≤ 1
goal (1 subgoal):
1. 2 < !m! ⇒ False
```

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

201,35 (3161/3970) (isabelle,sidekick,UTF-8-Isabelle)Nm r o UG 112/16 MB 15:04

The last line of the proof is unnecessary, and merely restates what was proved by the calculation. That's why its proof is trivial. We could have concluded this proof fragment as follows, feeding the calculation straight into the desired conclusion:

```
finally show "False" using 0
  by auto
```

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 α 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: 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

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

And the analogous forms of intersections...

Once again, the logical equivalences are essentially definitions.

The third form of union is seldom seen.

A Simple Set Theory Proof

The screenshot shows the Isabelle/Isar IDE interface. The main window displays a theory named 'Examples' which imports 'Complex_Main'. A lemma is defined with the following statement:

$$\text{Lemma } "(\bigcap x \in A \cup B. C x \cup D) = ((\bigcap x \in A. C x) \cap (\bigcap x \in B. C x)) \cup D "$$

The proof is completed with the following commands:

```
Auto solve_direct: (  $\bigcup x \in A. f x \cup g x$  ) =  
   $\bigcup ( f ` A ) \cup$   
   $\bigcup ( g ` A )$  can be solved directly with  
Complete_Lattices.Un_Union_image:  
(  $\bigcup x \in ?C. ?A x \cup ?B x$  ) =  $\bigcup ( ?A ` ?C ) \cup \bigcup ( ?B ` ?C )$ 
```

Annotations in the image include:

- A blue callout box pointing to the lemma statement: "There's a popup window!"
- A blue callout box pointing to a small icon in the left-hand gutter: "Why the little icon?"

The IDE interface also shows a right-hand sidebar with 'Documentation', 'Sidekick', and 'Theories' panels, and a bottom status bar with '8,1 (184/373)' and '(isabelle,sidekick,UTF-8-Isabelle)Nm r o UG 159/208MB 16:27'.

Special symbols can be inserted using the Symbols panel. ASCII can simply be typed; auto-completions for symbols will be offered.

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

Also: look for icons in the left-hand "gutter", since they indicate errors, warnings or information.

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

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

where

$$x \in \text{insert } a B \iff x = a \vee x \in B$$

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

```
Examples.thy
Examples.thy (~/.Dropbox/ACS/7 - Sets/)

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

proof (prove): step 0
goal (1 subgoal):
1. finite A  $\implies$  ( $\sum_{i \in A}. c * f i$ ) = c * setsum f A

```

a way to specify the types of variables

induction on the finite set, A

16,61 (276/373) (isabelle,sidekick,UTF-8-Isabelle)Nmr o UG 178/210M3 16:37

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
~/Dropbox/ACS/7 - Sets/

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

proof (prove): step 1

goal (2 subgoals):
  1. ( $\sum_{i \in \{ \}}$  c * f i) = c * setsum f { }
  2.  $\wedge x F. [ [ \text{finite } F; x \notin F; (\sum_{i \in F} c * f i) = c * \text{setsum } f F ] ]$ 
      $\implies (\sum_{i \in \text{insert } x F} c * f i) = c * \text{setsum } f (\text{insert } x F)$ 
```

base case: A is empty

inductive step: A = insert

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
Examples.thy (~/.Dropbox/ACS/7 - Sets/)

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

proof (prove): step 2

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

need a distributive law!

18,11 (325/373) (isabelle,sidekick,UTF-8-Isabelle)Nm r o UG 16 /498MB 16:59

Finished!

```
Examples.thy (modified)
Examples.thy (~/.Dropbox/ACS/7 - Sets/)

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

No need for the first "auto"...

proof (prove): step 3

goal:
No subgoals!

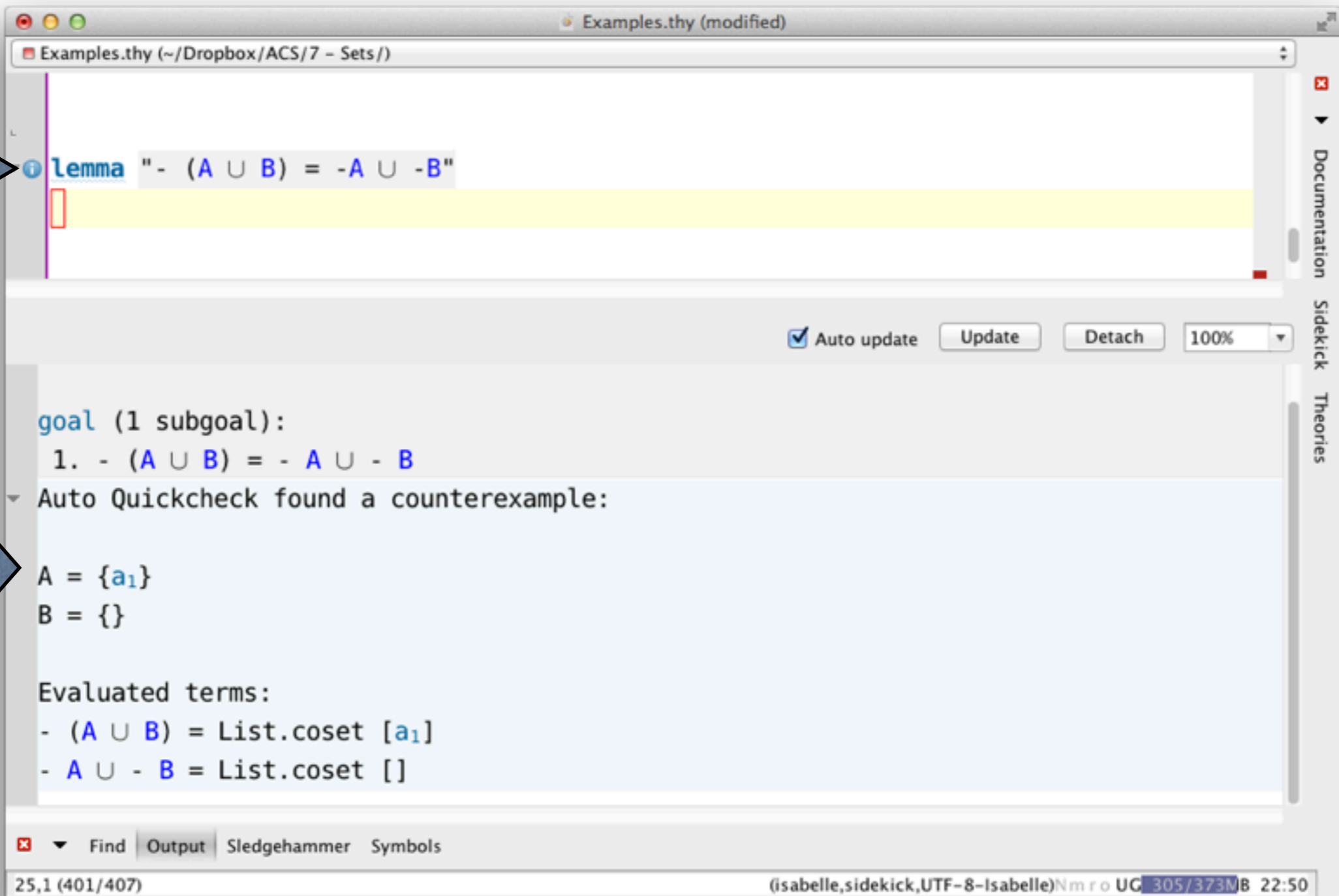
19,39 (364/374) (isabelle,sidekick,UTF-8-Isabelle)Nm r o UG 208/498MB 17:02

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

Counterexample Finding

- Don't waste time trying to prove false statements!
- Isabelle can find counterexamples quickly...
 - `quickcheck`: random testing of executable specifications (broadly interpreted)
 - `nitpick`: a general, SAT-based disprover
 - `try`: calls both of those (and `sledgehammer`)
- Type these commands right in the document.

Quickcheck Example



The screenshot shows the Isabelle/Quickcheck interface. At the top, a window titled "Examples.thy (modified)" contains a lemma: `Lemma "- (A ∪ B) = -A ∪ -B"`. A blue arrow points to the lemma's name. Below the lemma, the Quickcheck output is displayed in a light blue box. It shows a goal with one subgoal: `1. - (A ∪ B) = -A ∪ -B`. A message states: "Auto Quickcheck found a counterexample:". Below this, the counterexample is defined: `A = {a1}` and `B = {}`. Further down, "Evaluated terms:" are shown: `- (A ∪ B) = List.coset [a1]` and `- A ∪ - B = List.coset []`. A large blue arrow points to the counterexample section. The interface includes a sidebar with "Documentation", "Sidekick", and "Theories". At the bottom, there are tabs for "Find", "Output", "Sledgehammer", and "Symbols". The status bar at the bottom shows "25,1 (401/407)" and "(isabelle,sidekick,UTF-8-Isabelle)Nm r o UG 305/373MB 22:50".

```
Examples.thy (~/.Dropbox/ACS/7 - Sets/)  
  
Lemma "- (A ∪ B) = -A ∪ -B"  
  
goal (1 subgoal):  
  1. - (A ∪ B) = -A ∪ -B  
Auto Quickcheck found a counterexample:  
  
A = {a1}  
B = {}  
  
Evaluated terms:  
- (A ∪ B) = List.coset [a1]  
- A ∪ - B = List.coset []
```

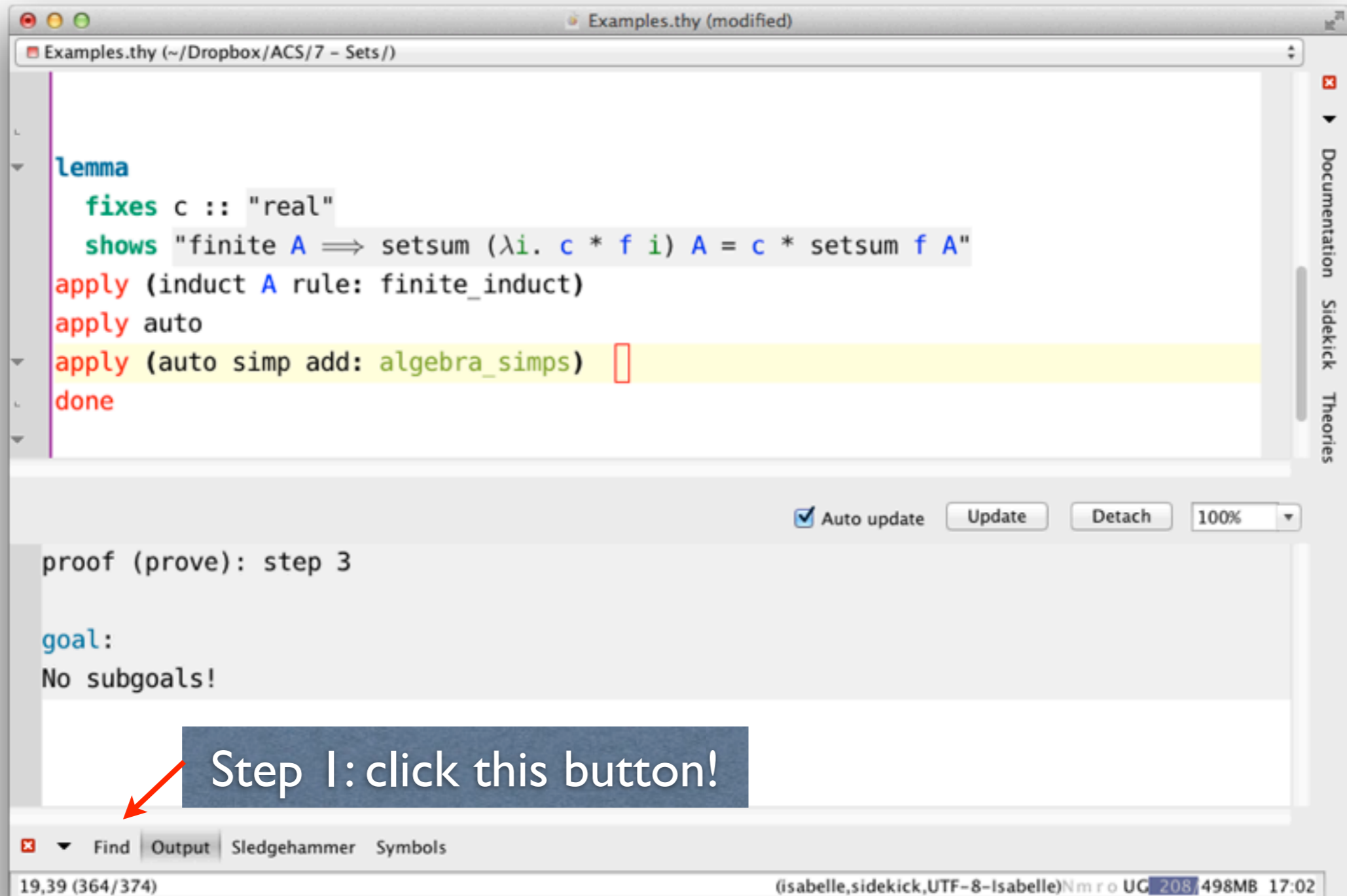
A minimal calls to quickcheck is performed automatically. Auto nitpick and even auto sledgehammer can be configured in the plugin options.

They work especially well for functional programs, but work in other domains, as we see here.

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



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

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

Below the code editor, the proof progress is shown:

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

At the bottom of the IDE, there is a toolbar with several buttons: Find, Output, Sledgehammer, and Symbols. The 'Find' button is highlighted with a red arrow pointing to it from a dark blue callout box containing the text "Step 1: click this button!".

The status bar at the bottom of the window shows: 19,39 (364/374) (isabelle,sidekick,UTF-8-Isabelle)Nm r o UG 208/498MB 17:02

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 the Isabelle/Isar IDE interface. The main editor displays a theorem proof:

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

The line `apply (auto simp add: algebra_simps)` is highlighted in yellow. A blue callout box with the text "press this button" has a red arrow pointing to the "Apply" button in the search criteria panel.

The search criteria panel shows the following settings:

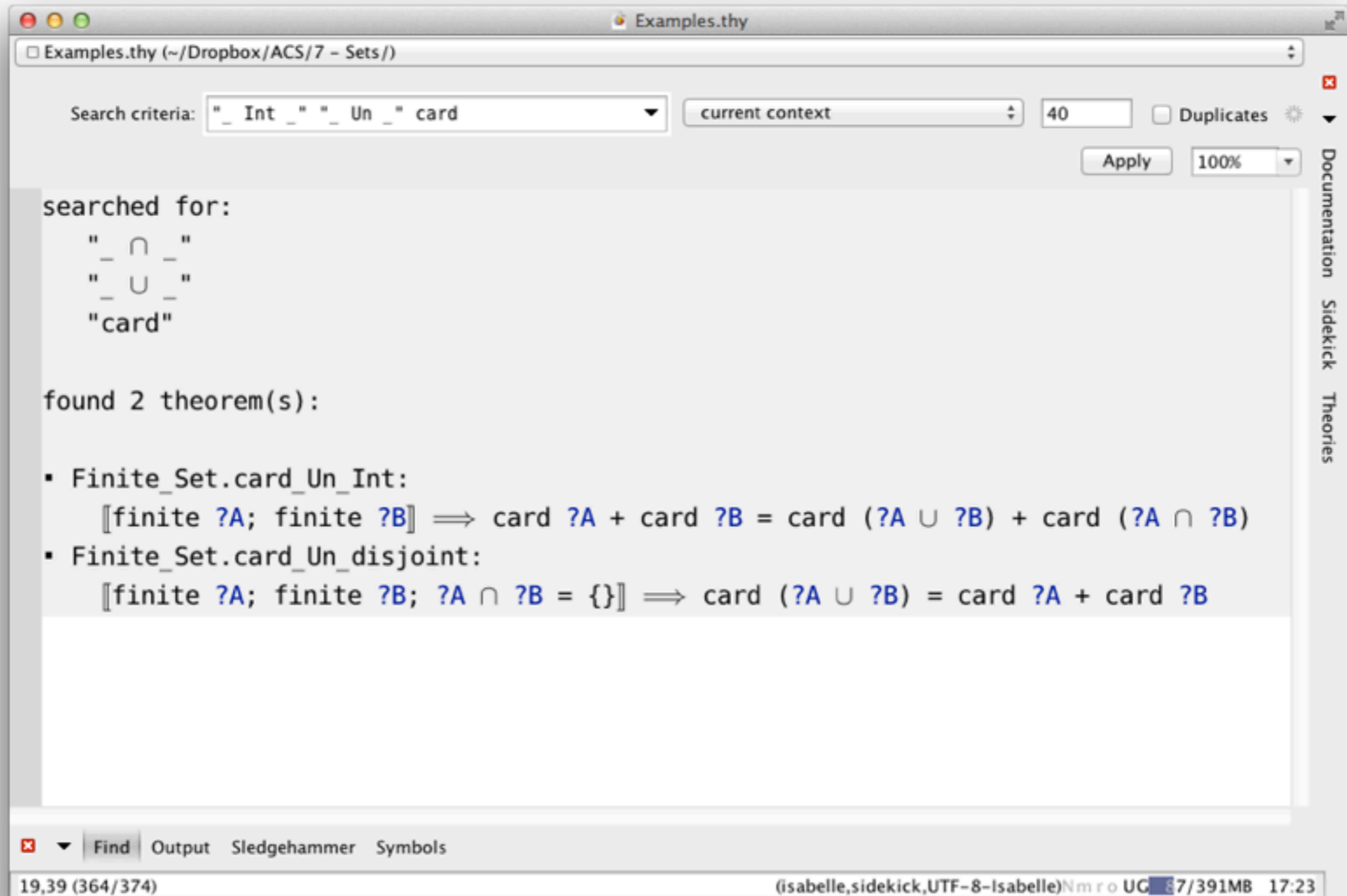
- Search criteria: `"_ Int _" "_ Un _" card`
- current context
- 40
- Duplicates
- Apply
- 100%

A blue callout box with the text "Step 2: type some patterns" has a red arrow pointing to the search criteria input field.

The bottom status bar shows: 19,39 (364/374) (isabelle,sidekick,UTF-8-Isabelle)Nm r o UG 137/371MB 17:21

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?



The Find panel, like all the other panels, can be detached or docked in various places so that it is always available.

Interactive Formal Verification

8: Inductive Definitions

Lawrence C Paulson
Computer Laboratory
University of Cambridge

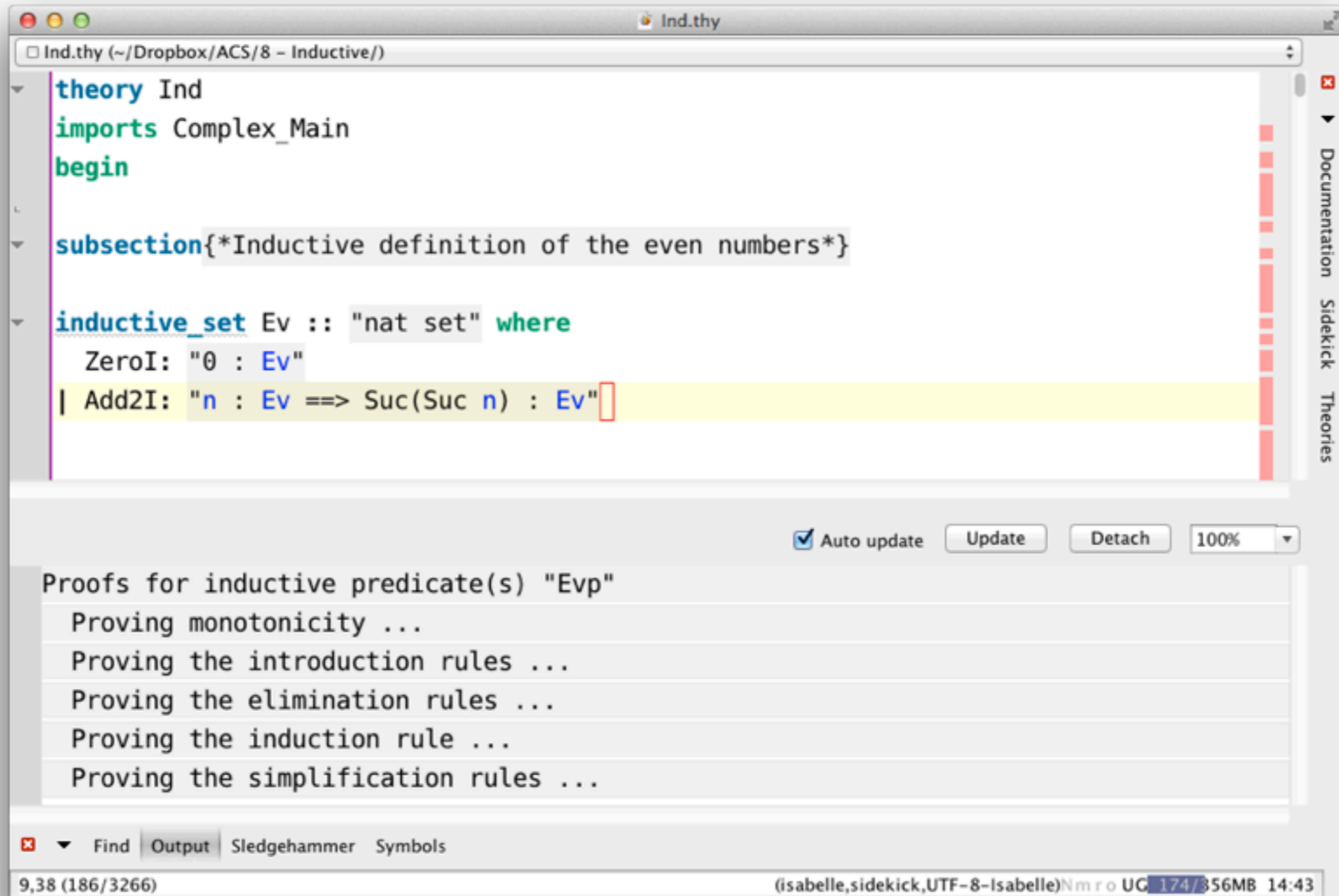
Overview

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



The screenshot shows the Isabelle IDE interface. The main window displays the following code in a file named `Ind.thy`:

```
theory Ind
  imports Complex_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"
```

The `Add2I` rule is highlighted in yellow. Below the code editor, the output window shows the progress of the inductive definition process:

```
Auto update Update Detach 100%
Proofs for inductive predicate(s) "Ev"
  Proving monotonicity ...
  Proving the introduction rules ...
  Proving the elimination rules ...
  Proving the induction rule ...
  Proving the simplification rules ...
```

The status bar at the bottom indicates the system is running Isabelle with Sidekick, using UTF-8 encoding, with 174 MB of memory used and 14:43 remaining.

Even Numbers Belong to Ev

The screenshot shows the Isabelle/Isar IDE 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
  apply (auto simp add: ZeroI Add2I)
  done
```

A callout box with a blue background and white text contains the text: "ordinary induction yields two subgoals". Two red arrows point from this box to the `induct k` line in the code and the `goal (2 subgoals):` line in the proof output.

The proof output shows the following goal:

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

The bottom status bar shows the file name `Ind.thy`, the location `(~/Dropbox/ACS/8 - Inductive/)`, and the system information `(isabelle,sidekick,UTF-8-Isabelle)Nr o UG 252/356MB 14:44`.

Proving Set Membership

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
```

The line `apply auto` is highlighted in yellow. Below the editor, the output window shows the result of the proof:

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

Two red arrows point from a text box to the `apply auto` line and the first subgoal. The text box contains the following text:

after simplification, the subgoals resemble the introduction rules

The interface includes a sidebar on the right with options for Documentation, Sidekick, and Theories. At the bottom, there are tabs for Find, Output, Sledgehammer, and Symbols. The status bar at the bottom shows the version 17.11 (280/3266), the encoding (isabelle,sidekick,UTF-8-Isabelle), and the memory usage (44/387MB) at 14:44.

Finishing the Proof

```
Ind.thy (~/.Dropbox/ACS/8 - Inductive/)
```

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

Isabelle also supports *introduction rules* (backward chaining)

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

27,33 (437/3266) (isabelle,sidekick,UTF-8-Isabelle)Nm r o UG 99/387MB 14:45

Rule Induction

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

Ev Has only Even Numbers

The screenshot shows the Isabelle/Proof General editor interface. The main window displays 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
```

Annotations in the image include:

- A yellow highlight on the lemma statement: `lemma "n ∈ Ev ⇒ ∃k. n = 2*k"`.
- A red box around the closing quote of the lemma statement.
- Two red arrows pointing from text boxes to the `rule: Ev.induct` part of the `apply` command.
- A text box containing "naming the induction rule" with an arrow pointing to `Ev.induct`.
- A text box containing "rule induction is needed!" with an arrow pointing to `induct n`.

The bottom panel of the editor shows the proof state:

```
proof (prove): step 0
goal (1 subgoal):
1. n ∈ Ev ⇒ ∃k. n = 2 * k
```

The status bar at the bottom indicates the current position in the document: 37,29 (521/3266) and the session information: (isabelle,sidekick,UTF-8-Isabelle)Nm r o UG 224/387MB 15:03.

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

The screenshot shows the Isabelle/Proof General interface with a file named `Ind.thy`. The source code in the editor is:

```
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
```

The proof window below shows the following state:

- Control buttons: Auto update, Update, Detach, 100%
- Current step: `proof (prove): step 1`
- Goal (2 subgoals):
 - $\exists k. 0 = 2 * k$
 - $\bigwedge n. [n \in \text{Ev}; \exists k. n = 2 * k] \implies \exists k. \text{Suc} (\text{Suc } n) = 2 * k$

Two blue callout boxes with red arrows provide annotations:

- The first box, labeled "base case: n replaced by 0", points to the first subgoal.
- The second box, labeled "induction step: n replaced by Suc (Suc n)", points to the second subgoal.

At the bottom of the window, the status bar shows: `(isabelle,sidekick,UTF-8-Isabelle)Nm r o UG 231/387MB 15:03`

One Tricky Goal Left!

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

```
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
```

The goal window shows the following text:

```
proof (prove): step 2  
goal (1 subgoal):  
1.  $\wedge k. 2 * k \in \text{Ev} \implies \exists ka. \text{Suc} (\text{Suc} (2 * k)) = 2 * ka$ 
```

A red arrow points from a blue callout box to the goal statement. The callout box contains the text: "a problem too difficult for auto".

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

The arith Proof Method

The screenshot shows the Isabelle/Proof General interface. The main editor displays a proof script for a lemma. The script is as follows:

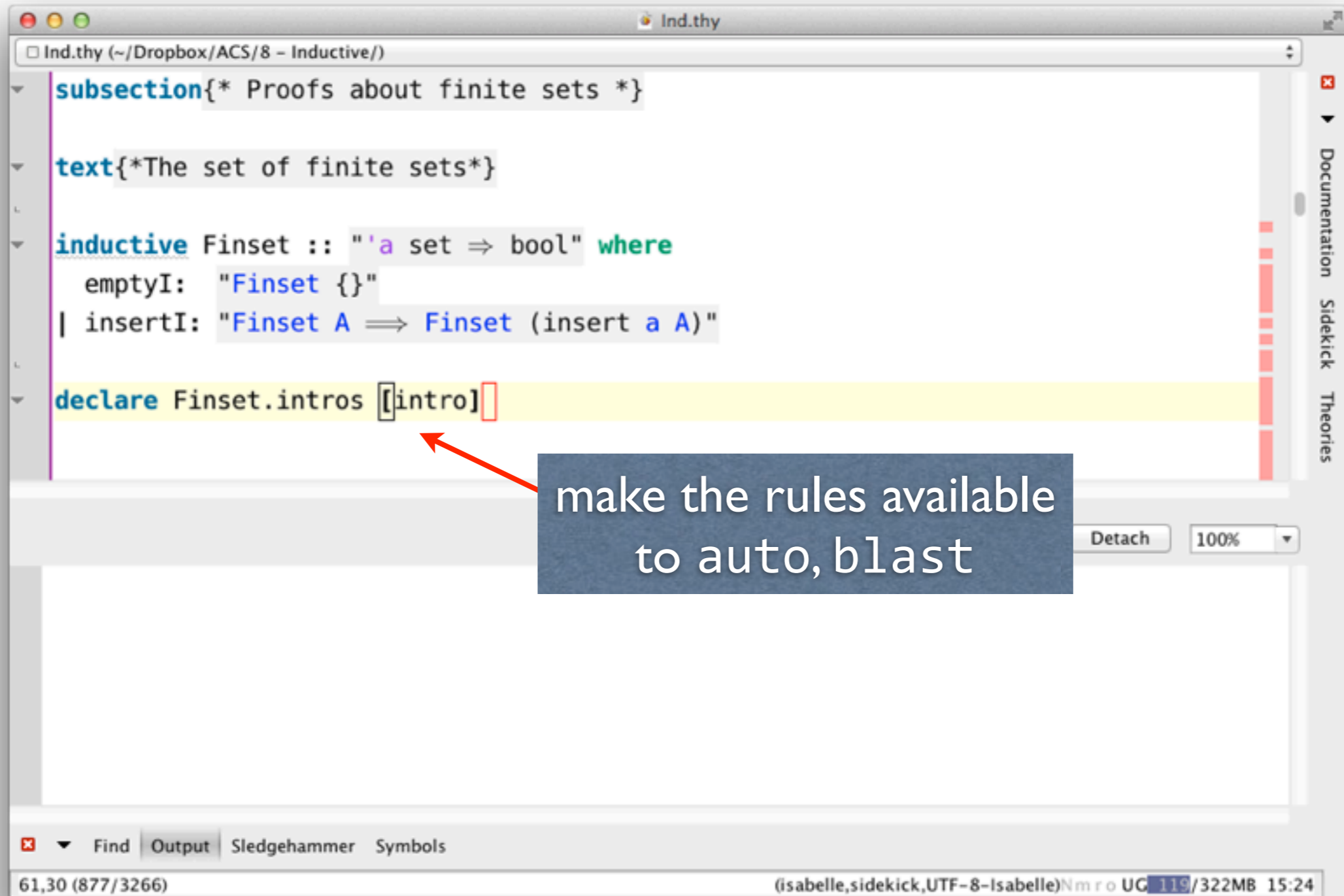
```
text{*All elements of this set are even.*}  
lemma "n ∈ Ev ⇒ ∃k. n = 2*k"  
apply (induct n rule: Ev.induct)  
apply auto  
apply arith  
done
```

The line `apply arith` is highlighted in yellow. A red arrow points from a dark blue box containing the text "for hard arithmetic subgoals" to the `arith` command. Below the editor, the proof progress is shown as "proof (prove): step 3" with a "goal:" section that says "No subgoals!". The bottom status bar indicates the current session is "(isabelle,sidekick,UTF-8-Isabelle)Nm r o UG 242/387MB 15:04".

Aside: Linear Arithmetic

- A *decidable* class of formulas
- Allowing the operators $+ - < \leq =$, and ...
- multiplication and division by *constants*:
 $\times 2, / 2$
- With slight variations, this class is decidable for the main arithmetic types.
- auto can solve simple arithmetic problems...
- arith handles logical connectives, quantifiers, etc.
- Decision procedures are necessary: proving arithmetic facts from first principles is too tedious.

Defining Finiteness



```
Ind.thy (~/.Dropbox/ACS/8 - Inductive/)  
subsection{* Proofs about finite sets *}  
  
text{*The set of finite sets*}  
  
inductive Finset :: "'a set => bool" where  
  emptyI: "Finset {}"  
| insertI: "Finset A => Finset (insert a A)"  
  
declare Finset.intros [intro]
```

make the rules available to auto, blast

61,30 (877/3266) (isabelle,sidekick,UTF-8-Isabelle)Nm r o UG 119/322MB 15:24

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

The Union of Two Finite Sets

```
Ind.thy (~/.Dropbox/ACS/8 - Inductive/)
```

```
lemma "[Finset A; Finset B] ==> Finset (A U B)"  
apply (induction A rule: Finset.induct)  
apply auto  
done
```

perform induction on A

```
proof (prove): step 1  
goal (2 subgoals):  
1. Finset B ==> Finset ({} U B)  
2.  $\wedge A a. [Finset A; Finset B ==> Finset (A U B); Finset B]$   
    $==> Finset (insert a A U B)$ 
```

67,40 (967/3266) (isabelle,sidekick,UTF-8-Isabelle)Nm r o UG 164/322MB 15:25

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 (modified)'. The main editor contains the following code:

```
lemma "[Finset A; B ⊆ A] ⇒ Finset B"  
apply (induction A arbitrary: B rule: Finset.induct) []  
apply auto
```

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

Below the main editor, there is a 'proof (prove): step 1' section. It shows a goal with two subgoals:

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

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".

At the bottom of the interface, there are tabs for 'Find', 'Output', 'Sledgehammer', and 'Symbols'. The status bar at the very bottom shows '78,53 (1080/3190)' on the left and '(isabelle,sidekick,UTF-8-Isabelle)Nm r o UG 70/384MB 15:26' on the right.

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

A Critical Point in the Proof

The screenshot shows the Isabelle/Proof General interface. The top window displays the following code:

```
lemma "[Finset A; B ⊆ A] ⇒ Finset B"  
apply (induction A arbitrary: B rule: Finset.induct)  
apply auto
```

The bottom window shows the current state of the proof:

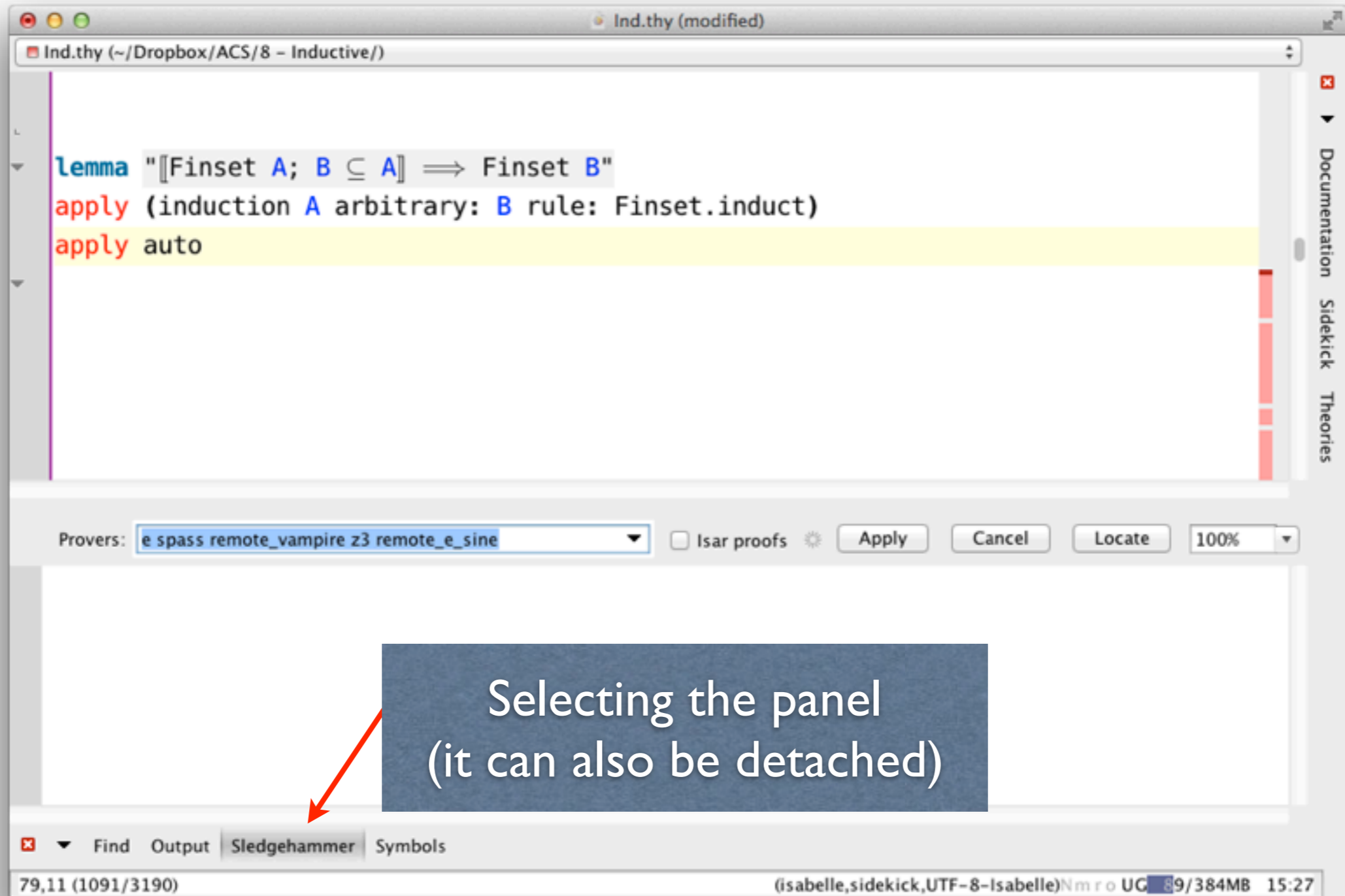
```
proof (prove): step 2  
goal (1 subgoal):  
1.  $\bigwedge A a B. [\bigwedge B. B \subseteq A \Rightarrow \text{Finset } B; B \subseteq \text{insert } a A] \Rightarrow \text{Finset } B$ 
```

A blue box with the text "now what??" is overlaid on the subgoal.

At the bottom of the interface, there are tabs for "Find", "Output", "Sledgehammer", and "Symbols". The status bar at the bottom shows "79,11 (1091/3190)" and "(isabelle,sidekick,UTF-8-Isabelle)Nm r o UG 5/384MB 15:26".

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 a window titled "Ind.thy (modified)" with a file path of "~/Dropbox/ACS/8 - Inductive/". The main text area contains the following code:

```
lemma "[Finset A; B ⊆ A] ⇒ Finset B"  
apply (induction A arbitrary: B rule: Finset.induct)  
apply auto
```

A yellow highlight is under the entire code block. Below the code is a "Provers:" dropdown menu with the text "e spass remote_vampire z3 remote_e_sine". To the right of the dropdown are buttons for "Apply", "Cancel", "Locate", and a "100%" zoom level. Below the dropdown, several lines of text are highlighted in green, each starting with a prover name and a "Try this:" prompt:

- "z3": Try this: by (metis Finset.insertI insert_subset mk_disjoint_insert subset_inse
- "spass": Try this: by (metis (full_types) Finset.insertI Set.set_insert insertI1 inse
- "e": Try this: by (metis (hide_lams, no_types) Finset.insertI dual_order.trans mk_dis
- "remote_vampire": Try this: by (metis Finset.simps Int_absorb2 Int_insert_right_if0 I
- "remote_e_sine": Try this: by (metis Collect_cong Collect_empty_eq Collect_mem_eq Col

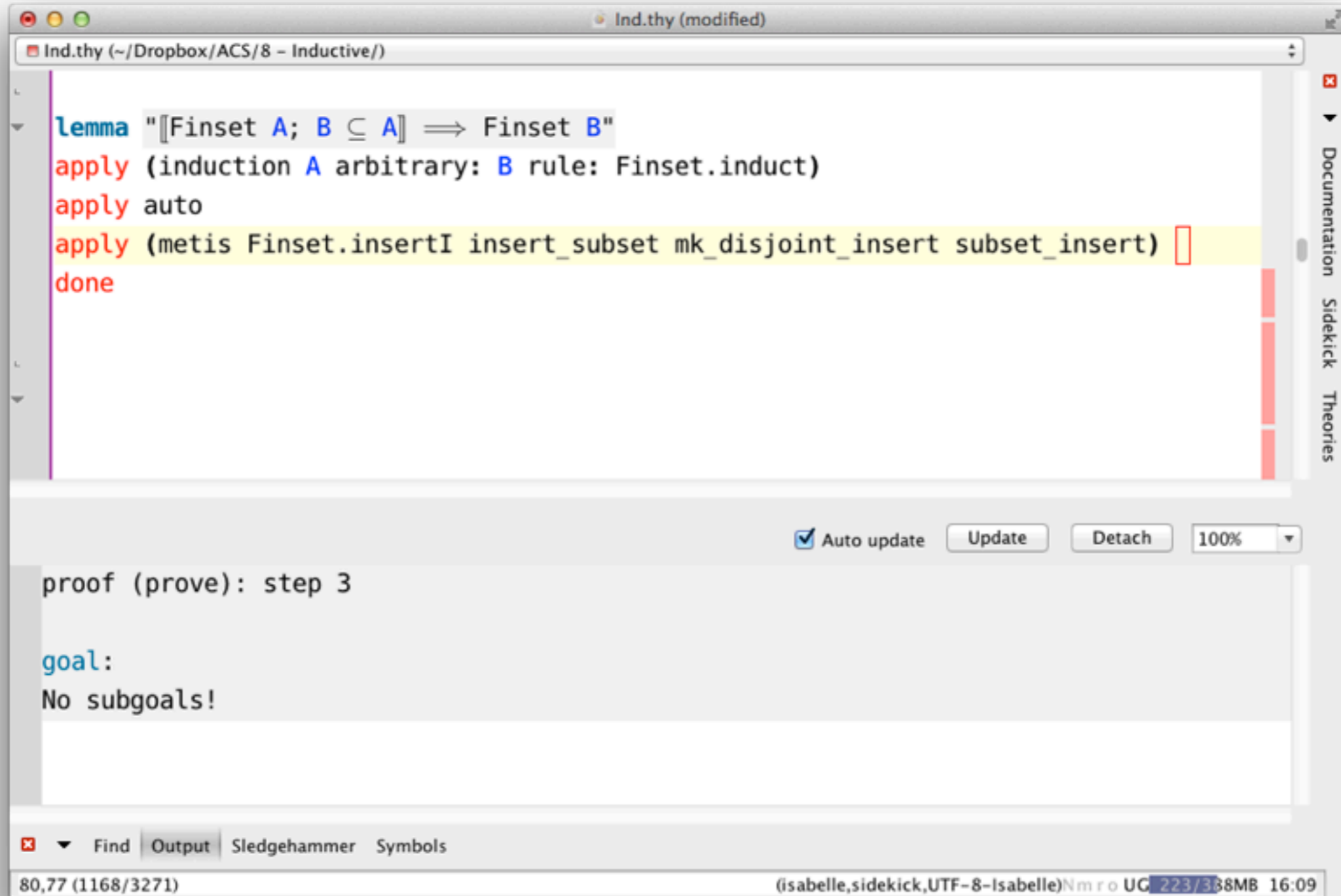
At the bottom of the highlighted text, it says "To minimize: sledgehammer min [remote_e_sine, provers = e spass remote_vampire z3 ren".

At the bottom of the window, there are tabs for "Find", "Output", "Sledgehammer", and "Symbols". The status bar at the very bottom shows "82,1 (1094/3189)" on the left and "UG 196/388MB 16:08" on the right.

this command should prove the goal

this one may return a more compact command

The Completed Proof



The screenshot shows the Isabelle/Proof General interface. The main editor displays a lemma and its proof:

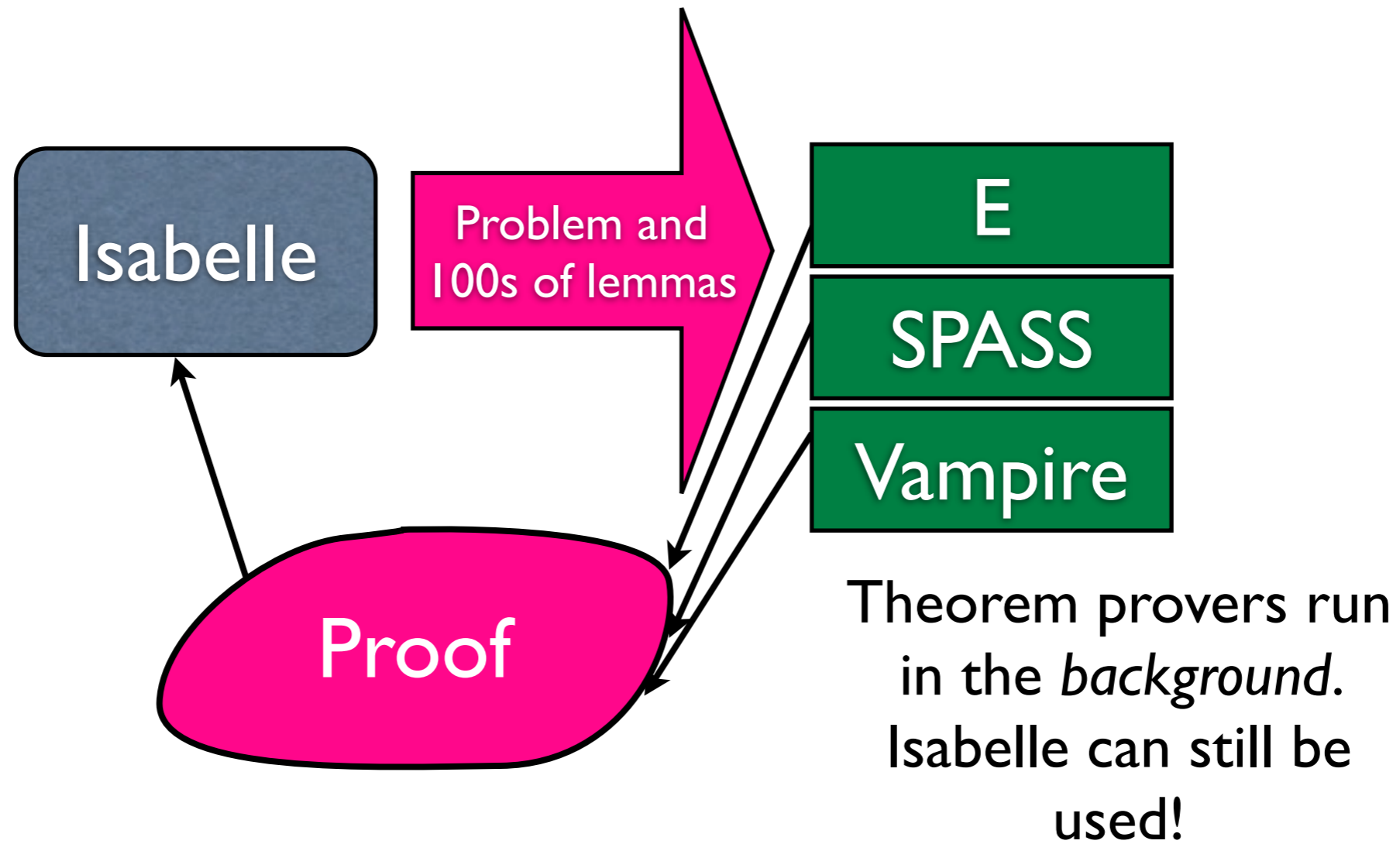
```
lemma "[Finset A; B ⊆ A] ⇒ Finset B"  
  apply (induction A arbitrary: B rule: Finset.induct)  
  apply auto  
  apply (metis Finset.insertI insert_subset mk_disjoint_insert subset_insert) []  
  done
```

The proof step is completed, and the output window shows:

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

The status bar at the bottom indicates the memory usage: 80,77 (1168/3271) and the system information: (isabelle,sidekick,UTF-8-Isabelle)Nm r o UG 223/338MB 16:09.

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` or `smt`. This command sometimes runs slowly, and `smt` calls can be fragile.
- The `minimise` option removes redundant theorems, increasing the likelihood of success.
- Calling `metis` or `smt` 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

```
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"
```

```
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

a predicate with *special syntax*

```
inductive
  evalc :: "[com,state,state] => bool" ("⟨_,_⟩/ ~ _" [0,0,60] 60)
where
  Skip:    "⟨SKIP,s⟩ ~ s"
| Assign: "⟨x ::= a,s⟩ ~ s(x ::= a s)"
| Semi:   "⟨c0,s⟩ ~ s' => ⟨c1,s'⟩ ~ s' => ⟨c0; c1, s⟩ ~ s'"
| IfTrue: "b s => ⟨c0,s⟩ ~ s' => ⟨IF b THEN c0 ELSE c1, s⟩ ~ s'"
| IfFalse: "¬b s => ⟨c1,s⟩ ~ s' => ⟨IF b THEN c0 ELSE c1, s⟩ ~ s'"
| WhileFalse: "¬b s => ⟨WHILE b DO c,s⟩ ~ s"
| WhileTrue:  "b s => ⟨c,s⟩ ~ s' => ⟨WHILE b DO c, s'⟩ ~ s'
              => ⟨WHILE b DO c, s⟩ ~ s'"

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

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 code editor with several annotations:

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

```
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!

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

The inductive version considers
all possible commands

Non-Termination in Isabelle

7 subgoals, one for each rule of the definition

```
Com.thy (modified)
Com.thy (~/.Dropbox/ACS/9 - Operational Semantics/)
lemma while_never: " $\langle c, s \rangle \rightsquigarrow u \implies c \neq \text{WHILE } (\lambda s. \text{True}) \text{ DO } c1$ "
apply [(induct rule: evalc.induct)]
apply auto
```

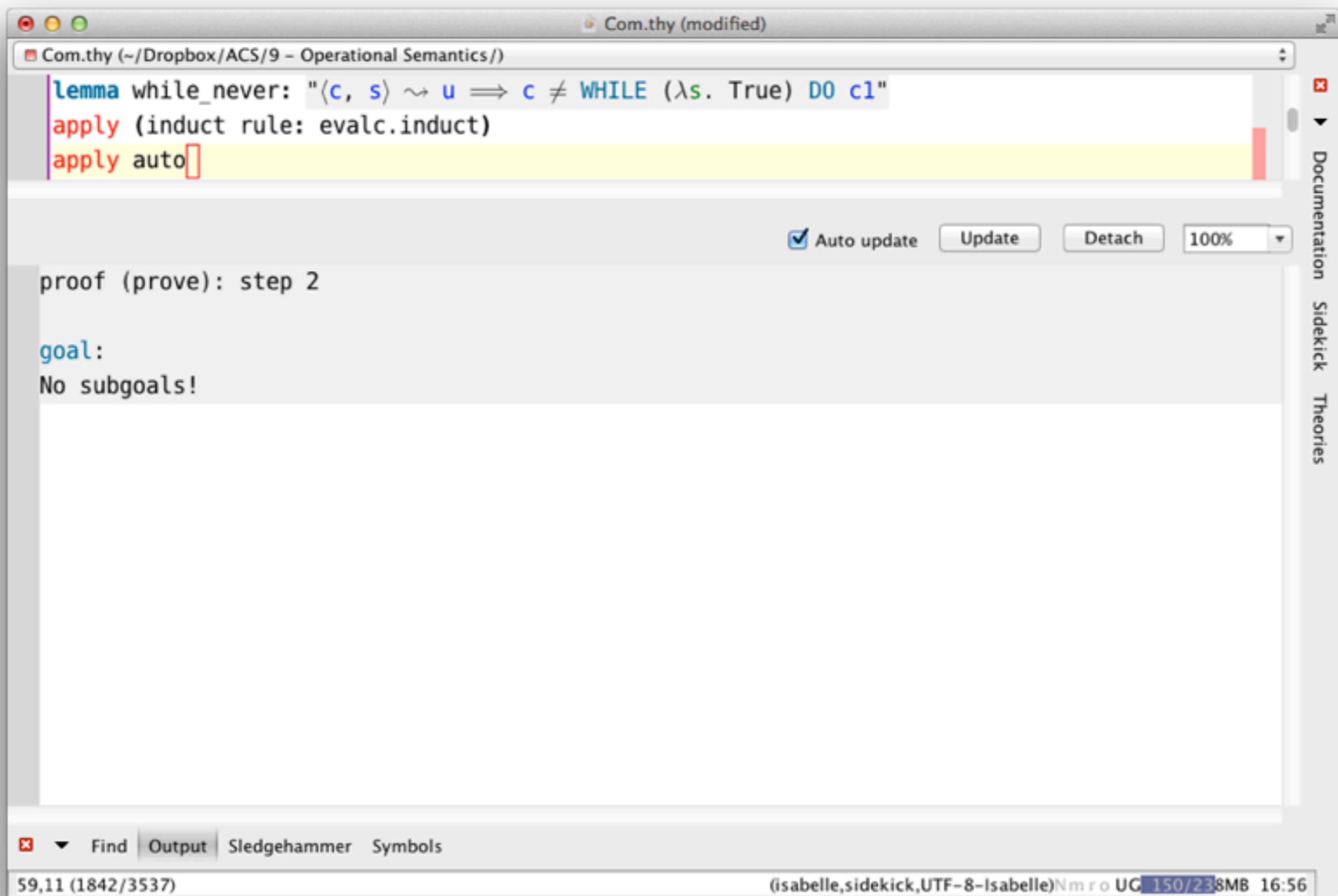
1. $\bigwedge s. \text{SKIP} \neq \text{WHILE } \lambda s. \text{True} \text{ DO } c1$
2. $\bigwedge x a s. x ::= a \neq \text{WHILE } \lambda s. \text{True} \text{ DO } c1$
3. $\bigwedge c0 s s' cla s'.$
 $[\langle c0, s \rangle \rightsquigarrow s'; c0 \neq \text{WHILE } \lambda s. \text{True} \text{ DO } c1; \langle cla, s' \rangle \rightsquigarrow s'; cla \neq \text{WHILE } \lambda s. \text{True} \text{ DO } c1]$
 $\implies c0; cla \neq \text{WHILE } \lambda s. \text{True} \text{ DO } c1$
4. $\bigwedge b s c0 s' cla.$
 $[b s; \langle c0, s \rangle \rightsquigarrow s'; c0 \neq \text{WHILE } \lambda s. \text{True} \text{ DO } c1]$
 $\implies \text{IF } b \text{ THEN } c0 \text{ ELSE } cla \neq \text{WHILE } \lambda s. \text{True} \text{ DO } c1$
5. $\bigwedge b s cla s' c0.$
 $[\neg b s; \langle cla, s \rangle \rightsquigarrow s'; cla \neq \text{WHILE } \lambda s. \text{True} \text{ DO } c1]$
 $\implies \text{IF } b \text{ THEN } c0 \text{ ELSE } cla \neq \text{WHILE } \lambda s. \text{True} \text{ DO } c1$
6. $\bigwedge b s c. \neg b s \implies \text{WHILE } b \text{ DO } c \neq \text{WHILE } \lambda s. \text{True} \text{ DO } c1$
7. $\bigwedge b s c s' s'.$
 $[b s; \langle c, s \rangle \rightsquigarrow s'; c \neq \text{WHILE } \lambda s. \text{True} \text{ DO } c1; \langle \text{WHILE } b \text{ DO } c, s' \rangle \rightsquigarrow s';$
 $\text{WHILE } b \text{ DO } c \neq \text{WHILE } \lambda s. \text{True} \text{ DO } c1]$
 $\implies \text{WHILE } b \text{ DO } c \neq \text{WHILE } \lambda s. \text{True} \text{ DO } c1$

58,34 (1831/3537) (isabelle,sidekick,UTF-8-Isabelle)Nm r o UG 128/256MB 16:54

Most are trivial, by distinctness

trivial for another reason

Done!



The screenshot shows the Isabelle/ML editor interface. The top window displays the source code for a lemma and its proof:

```
lemma while_never: " $\langle c, s \rangle \rightsquigarrow u \implies c \neq \text{WHILE } (\lambda s. \text{True}) \text{ DO } c1$ "  
apply (induct rule: evalc.induct)  
apply auto
```

The proof progress is shown in the main area:

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

The bottom status bar indicates the current state: 59,11 (1842/3537) (isabelle,sidekick,UTF-8-Isabelle) N m r o UG 150/238MB 16:56.

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...

allow the other state to vary

The screenshot shows the Isabelle/Isabelle IDE interface. At the top, a theorem is defined:

```
theorem com_det: "<math>\langle c, s \rangle \rightsquigarrow t \implies \langle c, s \rangle \rightsquigarrow u \implies u = t</math>"
```

The proof is completed with the following commands:

```
apply (induct arbitrary: u rule: evalc.induct)
apply blast+
```

Below the code, seven proof steps are listed, each representing a goal in the proof:

- $\bigwedge s u. \langle \text{SKIP}, s \rangle \rightsquigarrow u \implies u = s$
- $\bigwedge x a s u. \langle x := a, s \rangle \rightsquigarrow u \implies u = s(x := a s)$
- $\bigwedge c\theta s s' c1 s' u. \left[\langle c\theta, s \rangle \rightsquigarrow s'; \bigwedge u. \langle c\theta, s \rangle \rightsquigarrow u \implies u = s'; \langle c1, s' \rangle \rightsquigarrow s'; \bigwedge u. \langle c1, s' \rangle \rightsquigarrow u \implies u = s'; \langle c\theta ; c1, s \rangle \rightsquigarrow u \right] \implies u = s'$
- $\bigwedge b s c\theta s' c1 u. \left[b s; \langle c\theta, s \rangle \rightsquigarrow s'; \bigwedge u. \langle c\theta, s \rangle \rightsquigarrow u \implies u = s'; \langle \text{IF } b \text{ THEN } c\theta \text{ ELSE } c1, s \rangle \rightsquigarrow u \right] \implies u = s'$
- $\bigwedge b s c1 s' c\theta u. \left[\neg b s; \langle c1, s \rangle \rightsquigarrow s'; \bigwedge u. \langle c1, s \rangle \rightsquigarrow u \implies u = s'; \langle \text{IF } b \text{ THEN } c\theta \text{ ELSE } c1, s \rangle \rightsquigarrow u \right] \implies u = s'$
- $\bigwedge b s c u. \left[\neg b s; \langle \text{WHILE } b \text{ DO } c, s \rangle \rightsquigarrow u \right] \implies u = s$
- $\bigwedge b s c s' s' u. \left[b s; \langle c, s \rangle \rightsquigarrow s'; \bigwedge u. \langle c, s \rangle \rightsquigarrow u \implies u = s'; \langle \text{WHILE } b \text{ DO } c, s' \rangle \rightsquigarrow s'; \bigwedge u. \langle \text{WHILE } b \text{ DO } c, s' \rangle \rightsquigarrow u \implies u = s'; \langle \text{WHILE } b \text{ DO } c, s \rangle \rightsquigarrow u \right] \implies u = s'$

The IDE interface includes a status bar at the bottom with the following information: 65,47 (1946/3537) Text font size: 16 (isabelle,sidekick,UTF-8-Isabelle)N m r o UG 145/239MB 16:59

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. These subgoals are mostly trivial: rule inversion, which we set up previously, expresses precisely what we need: that if the given commands have executed, then corresponding intermediate states have been reached. The induction hypothesis allow us to assume the determinacy of the sub-commands.

Proved by Rule Inversion

```
Com.thy (modified)
Com.thy (~/.Dropbox/ACS/9 - Operational Semantics/)
theorem com_det: " $\langle c, s \rangle \rightsquigarrow t \implies \langle c, s \rangle \rightsquigarrow u \implies u = t$ "
  apply (induct arbitrary: u rule: evalc.induct)
  apply blast+
proof (prove): step 2
goal:
No subgoals!
```

call blast multiple times
(here auto is too slow)

66,13 (1959/3537) (isabelle,sidekick,UTF-8-Isabelle)N m r o UG 153/239MB 16:59

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

Semantic Equivalence

We can even define the infix syntax

```
Com.thy (modified)
Com.thy (~/.Dropbox/ACS/9 - Operational Semantics/)
text{*Two commands are equivalent if they allow the same transitions.*}

definition
  equiv_c :: "com  $\Rightarrow$  com  $\Rightarrow$  bool" (infixl "~" 50)
where
  "(c ~ c') = ( $\forall s s'$ . ( $\langle c, s \rangle \rightsquigarrow s'$ ) = ( $\langle c', s \rangle \rightsquigarrow s'$ ))"

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

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

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

It is trivially shown to be an equivalence relation

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

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

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

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


More Semantic Equivalence!

The screenshot shows a theorem prover interface with a window titled "Com.thy". The main editor contains two lemmas:

```
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)
```

A blue callout box on the right says "congruence laws: constructors preserve equivalence". A red arrow points from a blue callout box "by: gives a one-line proof" to the "by" keyword in the second lemma. Below the editor, a proof window shows:

```
proof (prove): step 0  
  
goal (1 subgoal):  
1. WHILE b DO c ~ IF b THEN (c ; WHILE b DO c) ELSE SKIP
```

At the bottom, there are tabs for "Find", "Output", "Sledgehammer", and "Symbols". The status bar shows "103,1 (2633/3515)", "Input/output complete", and "(isabelle,sidekick,UTF-8-Isabelle)Nm r o UG 109/359MB 16:19".

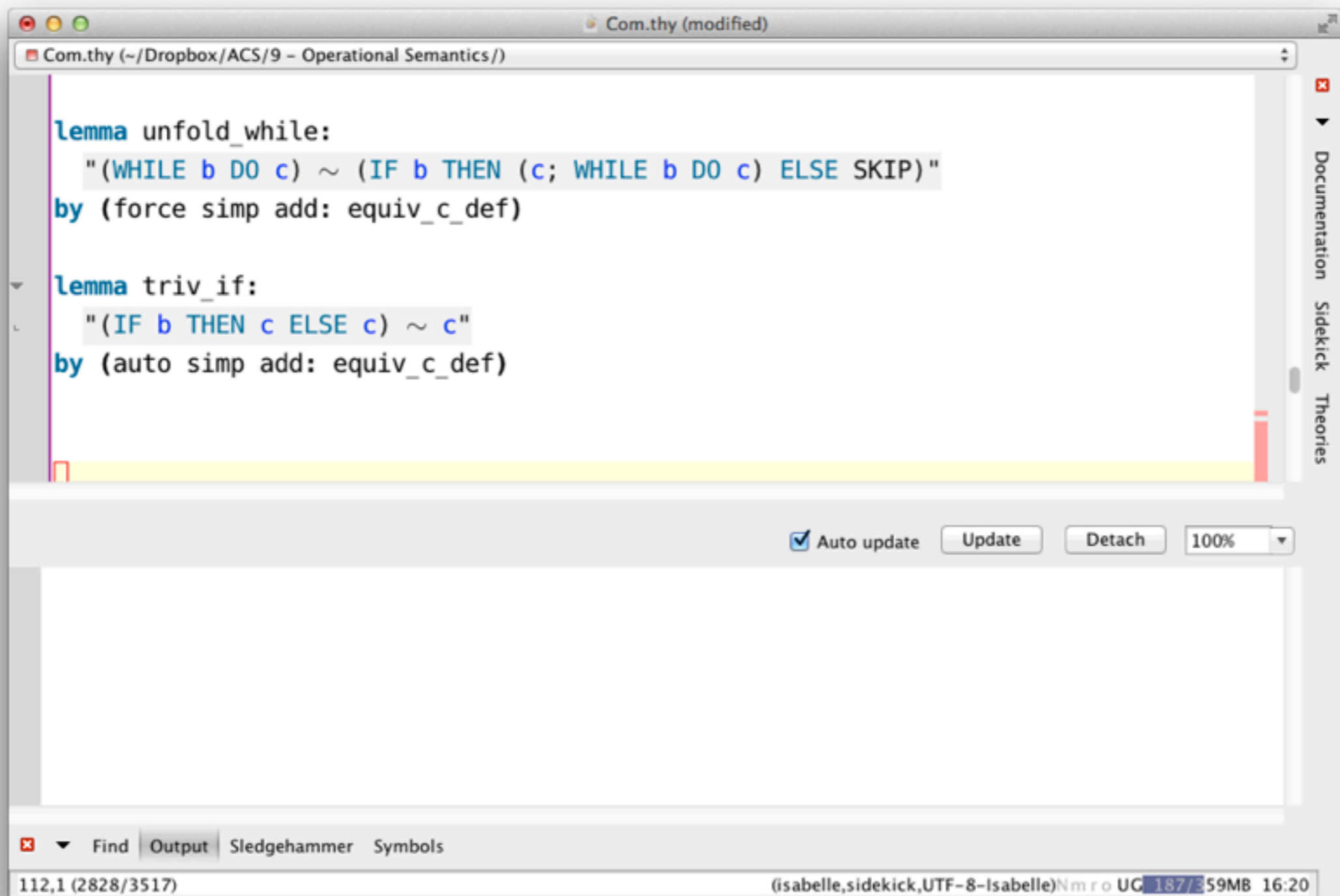
congruence laws:
constructors preserve
equivalence

by: gives a one-line proof

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!!



The screenshot shows a text editor window titled "Com.thy (modified)" with a file path of "~/Dropbox/ACS/9 - Operational Semantics/". The editor contains two lemmas:

```
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)
```

The editor interface includes a sidebar on the right with "Documentation", "Sidekick", and "Theories" options. At the bottom, there are controls for "Auto update" (checked), "Update", "Detach", and a zoom level of "100%". The status bar at the very bottom shows "112,1 (2828/3517)" and "(isabelle,sidekick,UTF-8-Isabelle)N m r o UG 187/3 59MB 16:20".

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

```
Com.thy (modified)
ACS/9 - Operational Semantics/

lemma equivI [intro!]:
  "(\s s'. \langle c, s \rangle \rightsquigarrow s' = \langle c', s \rangle \rightsquigarrow 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* in
blast/auto

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 (Isar) Proofs

- As we've already seen:
 - Structured proofs are clearer than a series of commands, but verbose.
 - The Isar language is rich and complex, supporting a great many proof styles.
- *But there's more!*
 - *Existential* reasoning: naming entities that “exist”.
 - 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

```
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 (induction 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]]
    $\implies$  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 a theorem prover interface with the following code and annotations:

```
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 (induction t)
print_cases
```

Annotations:

- Type a print_cases command!**: Points to the `print_cases` command in the proof script.
- Built-in cases**: Points to the `cases:` section in the proof script.
- abbreviation of conclusion**: Points to the `?case` placeholder in the `Lf` case.
- name of induction hyps**: Points to the `Br.IH` identifier in the `Br` case.

The `cases:` section contains the following code:

```
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.IH" : "reflect (reflect t1_) = t1_" "reflect (reflect t2_) = t2_" and
  "Br.premis" :
```

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.

It is unfortunately necessary to type the command `print_cases` right in your document.

Proof Using Named Cases

The screenshot displays the Isabelle/Proof General interface. The top window shows the source code for a lemma:

```
lemma reflect_reflect_ident: "reflect (reflect t) = t"
proof (induction t)
  case Lf
  show ?case by simp
next
  case (Br a t1 t2)
  then show ?case by simp
qed
```

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

- instances of the goal**: Points to the `show ?case` lines in both the `Lf` and `Br` cases.
- the two cases of the induction**: Points to the `case Lf` and `case (Br a t1 t2)` lines.
- list of bound variables**: Points to the list `(a t1 t2)` in the `Br` case header.

The bottom window shows the execution output for the `chain` proof:

```
proof (chain): step 7
picking this:
  ▪ reflect (reflect t1) = t1
  ▪ reflect (reflect t2) = t2
```

An annotation **the induction hypotheses** points to the two bullet points in the `picking this:` section.

The status bar at the bottom indicates: 17,7 (337/2592) Input/output complete (isabelle,sidekick,UTF-8-Isabelle)Nm r o UG 123/150MB 16:05

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 “then” with “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`.

Induction with a Context

The screenshot shows the Isabelle/HOL IDE interface. The top window displays the source code for an inductive definition and a proof. The code is as follows:

```
inductive Finset :: "'a set => bool" where
  emptyI: "Finset {}"
| insertI: "Finset A => Finset (insert a A)"

declare Finset.intros [intro]

lemma "[Finset A; B ⊆ A] => Finset B"
proof (induction A arbitrary: B rule: Finset.induct)
print_cases[]
```

The bottom window shows the generated proof script:

```
emptyI:
  fix B
  let "?case" = "Finset B"
  assume "emptyI.premis" : "B ⊆ {}"
insertI:
  fix A_ a_ B
  let "?case" = "Finset B"
  assume "insertI.hyps" : "Finset A_" and "insertI.IH" : "∧B. B ⊆ A_ => Finset B" and
  "insertI.premis" : "B ⊆ insert a_ A_"
```

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

- a named induction rule** points to the `rule: Finset.induct` argument in the `proof` block.
- an arbitrary variable** points to the `arbitrary: B` argument in the `proof` block and to the `fix B` line in the proof script.
- non-empty premises** points to the `insertI.hyps` and `insertI.premis` lines in the proof script.

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

```
BT.thy
BT.thy (~/.Dropbox/ACS/10 - Structured Induction/)
inductive Finset :: "'a set => bool" where
  emptyI: "Finset {}"
| insertI: "Finset A => Finset (insert a A)"

declare Finset.intros [intro]

lemma "[Finset A; B ⊆ A] => Finset B"
proof (induction A arbitrary: B rule: Finset.induct)
  case (emptyI B)
  then show "Finset B"
  by auto

proof (chain): step 3
  picking this:
  B ⊆ {}
```

“arbitrary” variables must be given names!

“then” gives the premise to the next step

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

```
BT.thy
BT.thy (~/.Dropbox/ACS/10 - Structured Induction/)

lemma "[Finset A; B ⊆ A] ⇒ Finset B"
proof (induction A arbitrary: B rule: Finset.induct)
  case (emptyI B)
  then show "Finset B"
    by auto
  next
  case (insertI A a B)
  show "Finset B"
  proof (cases "B ⊆ A")
    case True

```

“arbitrary” variables must (again) be given names!

Boolean case analysis on a formula

```
proof (state): step 9
goal (2 subgoals):
1. B ⊆ A ⇒ Finset B
2. ¬ B ⊆ A ⇒ Finset B
```

38,24 (751/2664) (isabelle,sidekick,UTF-8-Isabelle)Nm r o UG 166/207MB 16:36

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

The Complete Proof

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

```
lemma "[Finset A; B ⊆ A] ⇒ Finset B"
proof (induction A arbitrary: B rule: Finset.induct)
  case (emptyI B)
  then show "Finset B"
    by auto
  next
  case (insertI A a B)
  show "Finset B"
  proof [(cases "B ⊆ A")]
    case True
    show "Finset B" using insertI True
    by auto
  next
  case False
  have Ba: "B - {a} ⊆ A" using `B ⊆ insert a A`
    by auto
  then have "B = insert a (B - {a})" using False
    by auto
  then show "Finset B"
    by (metis Ba Finset.insertI insertI.IH)
  qed
qed
```

Annotations in the image:

- reference to the induction hypothesis and premise**: points to the `insertI` fact in the `True` case.
- reference to the true case: $B \subseteq A$** : points to the `True` case label.
- direct quotation of a fact, using backquotes**: points to the ``B ⊆ insert a A`` fact in the `False` case.
- reference to the false case: $\neg B \subseteq A$** : points to the `False` case label.
- true and false cases**: points to the `True` and `False` case labels.

The script is displayed in a window titled "BT.thy" with a sidebar on the right containing "Documentation", "Sidekick", and "Theories". The status bar at the bottom shows "38,24 (751/2664)" and "(isabelle,sidekick,UTF-8-Isabelle)Nm r o UG 117/208MB 16:37".

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 $B - \{a\} \in \text{Fin}$.

Additional Proof Structures

```
case (insertI A a B)
show "Finset B"
proof (cases "B ⊆ A")
  case True
  show "Finset B" using insertI True
  by auto
next
case False
have Ba: "B - {a} ⊆ A" using `B ⊆ insert a A`
  by auto
then have "B = insert a (B - {a})" using False
  by auto
then show "Finset B"
  by (metis Ba Finset.insertI insertI.IH)
qed
```

```
case (insertI A a B)
show "Finset B"
proof (cases "B ⊆ A")
  case True
  with insertI show "Finset B"
  by auto
next
case False
from `B ⊆ insert a A` have Ba: "B - {a} ⊆ A"
  by auto
with False have "B = insert a (B - {a})"
  by auto
with Ba insertI.IH show "Finset B"
  by (metis Finset.insertI)
qed
```

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

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

Viewing Available Facts

Type a `print_facts` command!

```
next
  case False
  have Ba: "B - {a} ⊆ A" using `B ⊆ insert a A`
  by auto
print_facts
```

a recently proved fact

the false case: $\neg B \subseteq A$

list of facts for the `insertI` case (and again, with distinct names)

```
▪ ?B ⊆ A ⇒ Finset ?B
▪ B - {a} ⊆ A
Ba: B - {a} ⊆ A
False: ¬ B ⊆ A
assms:
insertI:
  ▪ Finset A
  ▪ ?B ⊆ A ⇒ Finset ?B
  ▪ B ⊆ insert a A
insertI.IH: ?B ⊆ A ⇒ Finset ?B
insertI.hyps: Finset A
insertI.prem1: B ⊆ insert a A
this: B - {a} ⊆ A
```

It is unfortunately necessary to type the command `print_facts` right in your document.

Popups

The screenshot shows a theorem prover interface with a code editor on the left and a popup window on the right. The code editor contains the following text:

```
then show "Finset B"
  by auto
next
case (insertI A a B)
show "Finset B"
proof (cases "B ⊆ A")
  case True
  show "Finset B" using
  by auto
  next
  case False
  have Ba: "B - {a} ⊆ A"
  by auto
  then have "B = insert a (B - {a})"
  by auto print_fact
```

The text "Finset B" and "B ⊆ A" in the code editor has wavy underlining. A popup window is open over the code, displaying a list of facts:

```
facts:
<unnamed>:
  • Finset A
  • ¬ B ⊆ A
  • B ⊆ insert a A
  • ?B ⊆ A ⇒ Finset ?B
  • B - {a} ⊆ A
  • B = insert a (B - {a})
Ba: B - {a} ⊆ A
False: ¬ B ⊆ A
assms:
insertI:
  • Finset A
  • ?B ⊆ A ⇒ Finset ?B
  • B ⊆ insert a A
insertI.IH: ?B ⊆ A ⇒ Finset ?B
insertI.hyps: Finset A
insertI.prem1: B ⊆ insert a A
this: B = insert a (B - {a})
```

A blue box with white text is overlaid on the code editor, stating: "wavy underlining means output is available".

Simply hover with the mouse over any text where you see wavy underlining.

moreover / ultimately

```
notepad
begin

  have l1: "fact1" sorry
  have l2: "fact2" sorry
  have l3: "fact3" sorry
  from l1 l2 l3
```

oops

proof (chain): step 7

picking this:

- fact1
- fact2
- fact3

```
notepad
begin

  have "fact1" sorry
  moreover
  have "fact2" sorry
  moreover
  have "fact3" sorry
  ultimately
```

oops

calculation:

- fact1
- fact2
- fact3

proof (chain): step 9

Existential Claims: “obtain”

The screenshot shows the Isabelle/Proof General interface. The top window displays a proof script for the lemma `dvd_trans`. The script includes the following code:

```
lemma dvd_trans:
  fixes a::nat
  assumes ab: "a dvd b" and bc: "b dvd c"
  shows "a dvd c"
proof -
  obtain v where "b = a * v"
    by (metis ab dvd_def)
  moreover obtain w where "c = b * w"
    by (metis bc dvd_def)
  ultimately have "c = a * (v * w)"
    by (simp add: mult_assoc)
```

Annotations with red arrows point to the `obtain` command and the goal. A blue box explains that `obtain` is used to obtain variables satisfying given properties. Another blue box explains that Isabelle needs to prove an elimination rule for this.

The bottom window shows the goal for the `obtain` command:

```
proof (prove): step 2
goal (1 subgoal):
  1. ( $\wedge v. b = a * v \implies thesis$ )  $\implies thesis$ 
```

At the bottom of the slide, the mathematical definition of the divides relation is given:

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

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 . Proof attempts involving “obtain” can be difficult to understand, especially when they fail. Isabelle 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”.

Chaining Facts: “moreover”

The screenshot displays the Isabelle/Proof General interface. The top pane shows the following lemma and proof:

```
lemma dvd_trans:
  fixes a::nat
  assumes ab: "a dvd b" and bc: "b dvd c"
  shows "a dvd c"
proof -
  obtain v where "b = a * v"
    by (metis ab dvd_def)
  moreover obtain w where "c = b * w"
    by (metis bc dvd_def)
  ultimately have "c = a * (v * w)"
    by (simp add: mult_assoc)
```

A callout box points to the `moreover` line, stating: **moreover** retains the result as a *calculation*.

The bottom pane shows the state after the `moreover` step:

```
calculation: b = a * v
proof (state): step 4
this:
  b = a * v
goal (1 subgoal):
```

A second callout box points to the `b = a * v` line in the state, stating: we now have the key property of `j`.

At the bottom of the window, the status bar shows: 116,11 (2681/3891) (isabelle,sidekick,UTF-8-Isabelle)Nr o UG 66/357MB 15:50

Delivering Facts: “ultimately”

```
BT.thy
BT.thy (~/.Dropbox/ACS/10 - Structured Induction/)
lemma dvd_trans:
  fixes a::nat
  assumes ab: "a dvd b" and bc: "b dvd c"
  shows "a dvd c"
proof -
  obtain v where "b = a * v"
  by (metis ab dvd_def)
  moreover obtain w where "c = b * w"
  by (metis bc dvd_def)
  ultimately have "c = a * (v * w)"
  by (simp add: mult_assoc)
```

Auto update Update Detach 100%

calculation:

- $b = a * v$
- $c = b * w$

proof (chain): step 7

picking this:

- $b = a * v$
- $c = b * w$

Find Output Sledgehammer Symbols

118,13 (2748/3899) (isabelle,sidekick,UTF-8-Isabelle)Nr o UG 171/308MB 16:03

Documentation Sidekick Theories

calculation holds previous results

ultimately gives them to the next step

The Finished Proof

```
BT.thy (modified)
BT.thy (~/.Dropbox/ACS/10 - Structured Induction/)
lemma dvd_trans:
  fixes a::nat
  assumes ab: "a dvd b" and bc: "b dvd c"
  shows "a dvd c"
proof -
  obtain v where "b = a * v"
    by (metis ab dvd_def)
  moreover obtain w where "c = b * w"
    by (metis bc dvd_def)
  ultimately have "c = a * (v * w)"
    by (simp add: mult_assoc)
  then show ?thesis
    by (rule dvdI)
qed

Find Output Sledgehammer Symbols
117,27 (2735/4179) (isabelle,sidekick,UTF-8-Isabelle)Nr o UG 217/30 1MB 16:36
```

A Simpler Proof

The screenshot shows a theorem prover interface with the following content:

```
BT.thy (modified)
BT.thy (~/.Dropbox/ACS/10 - Structured Induction/)
lemma dvd_trans:
  fixes a::nat
  assumes "a dvd b" "b dvd c"
  shows "a dvd c"
proof -
  from assms obtain v w where "b = a * v" "c = b * w"
  by (metis dvd_def)
  then have "c = a * (v * w)"
  by (simp add: mult_assoc)
  then show ?thesis ..
qed
```

using this:

- a dvd b
- b dvd c

goal (1 subgoal):

1. $(\wedge v w. [b = a * v; c = b * w] \Rightarrow \text{thesis}) \Rightarrow \text{thesis}$

Annotations:

- “..” is the default proof step (here, dvdI)
- can **obtain** multiple vars, facts in one step

Any proof can be written in a variety of different ways. The concluding step is surprising. The mysterious .. symbol denotes the default proof step, which in this case happens to be a rule called dvdI. This rule exactly matches the given premise and conclusion. In practice, however, default proof steps are seldom used.

Advanced Proof Structures

The screenshot shows a theorem prover interface with two main panels. The top panel, titled "notepad", contains a proof script:

```
begin
  have "P ∨ Q ∨ R" sorry
  moreover {assume "P" have "S" sorry}
  moreover {assume "Q" have "S" sorry}
  moreover {assume "R" have "S" sorry}
  ultimately have "S" by blast
end
```

The bottom panel shows the execution output:

```
calculation:
  ▪ P ∨ Q ∨ R
  ▪ P ⇒ S
  ▪ Q ⇒ S
  ▪ R ⇒ S
proof (chain): step 21
```

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

- A blue box labeled "local proof blocks" points to the three `moreover` lines in the script.
- A blue box labeled "these facts are available to blast" points to the three `▪ P ⇒ S` lines in the output.

The interface includes a status bar at the bottom with the following information: "175,13 (3521/4611)", "Input/output complete", "(isabelle,sidekick,UTF-8-Isabelle)Nm r o UG 111/200MB 17:20".

Here we see a three-way case distinction. Local blocks have many other uses.

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.

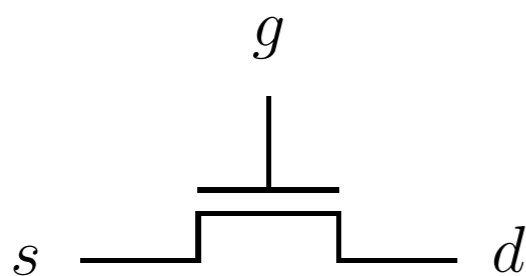
Hardware Verification

- 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



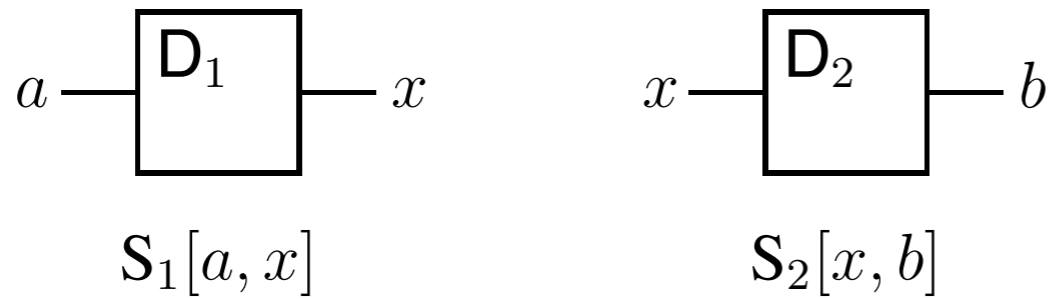
$g \rightarrow s = d$

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

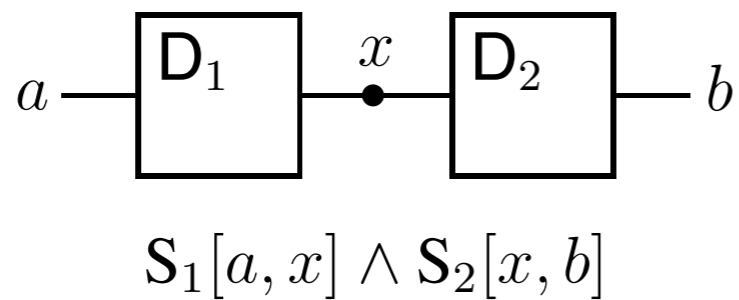
Values could be bits, words, signals (functions from time to bits), etc

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.

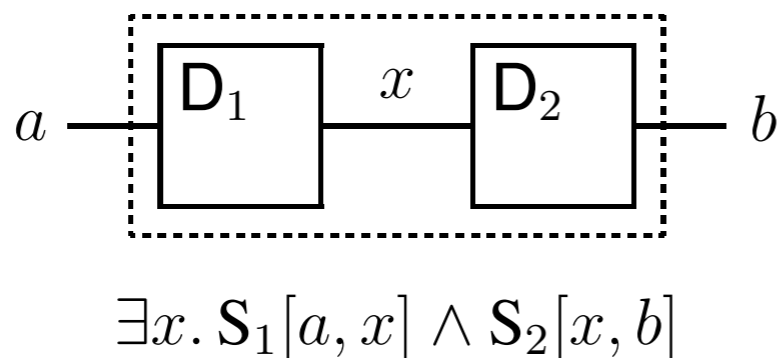
Relational Composition



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*.

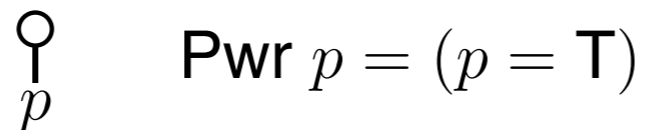
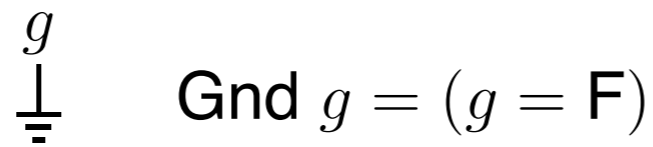
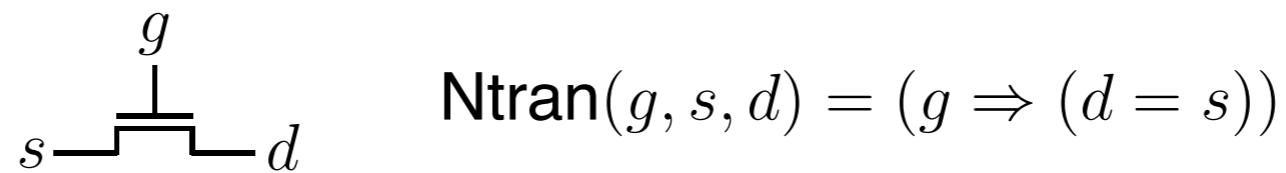
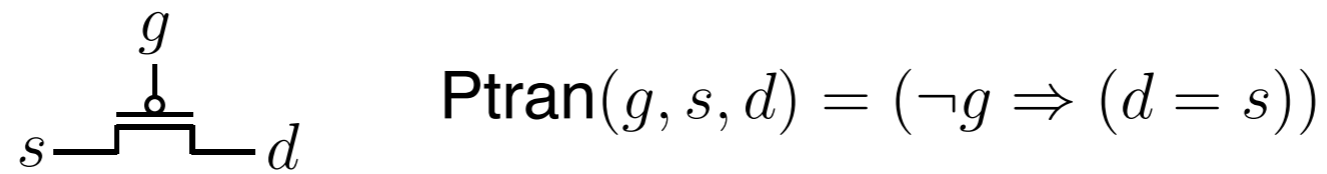
Impossible implementations satisfy all specifications!

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

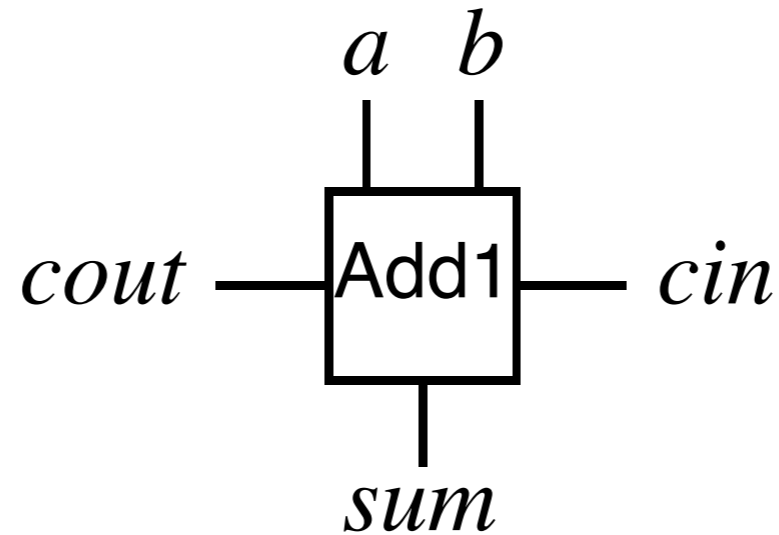
```
text{* Power and Ground*}
```

```
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



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

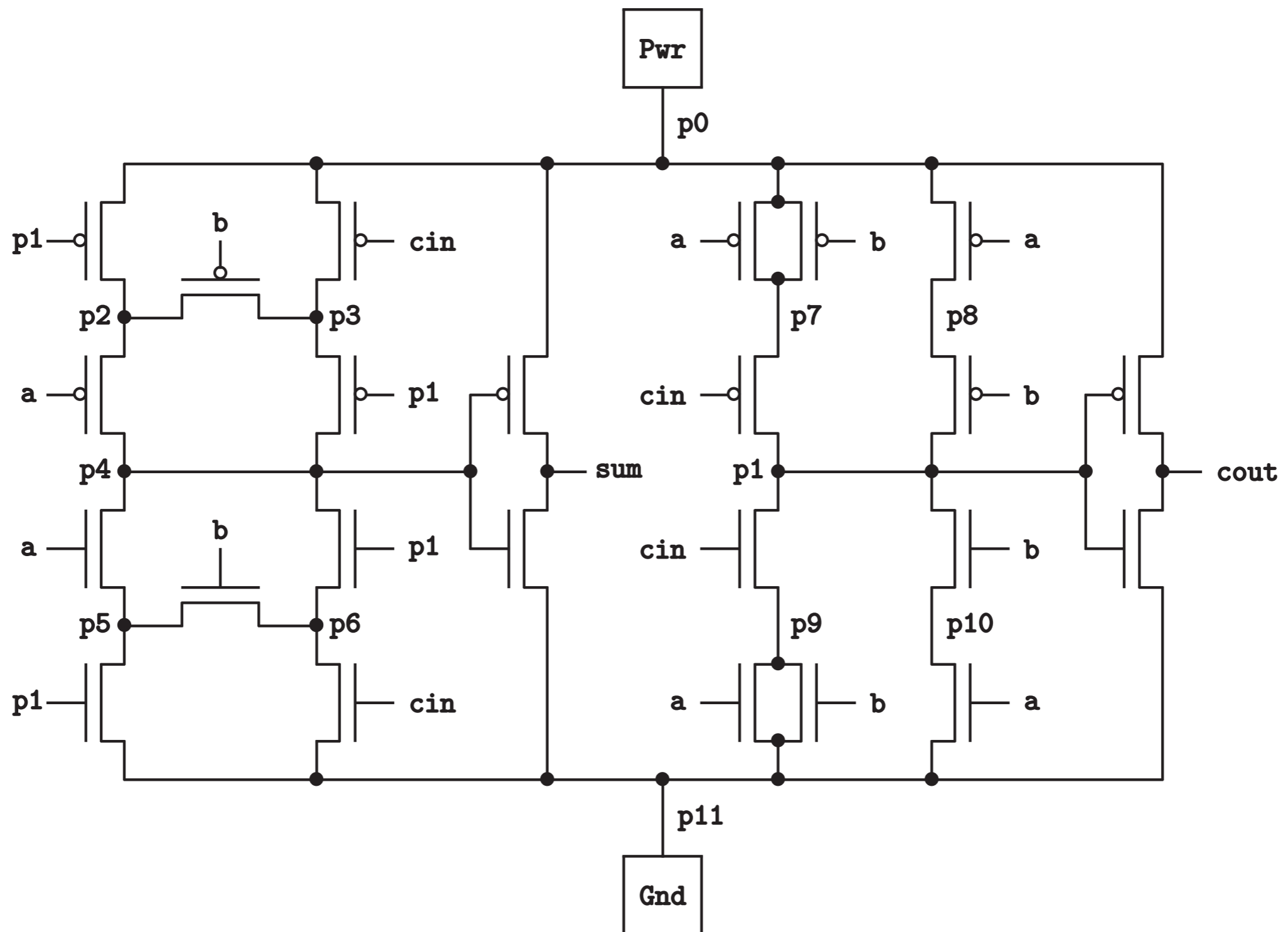
```
text{* 1-bit full adder specification *}

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

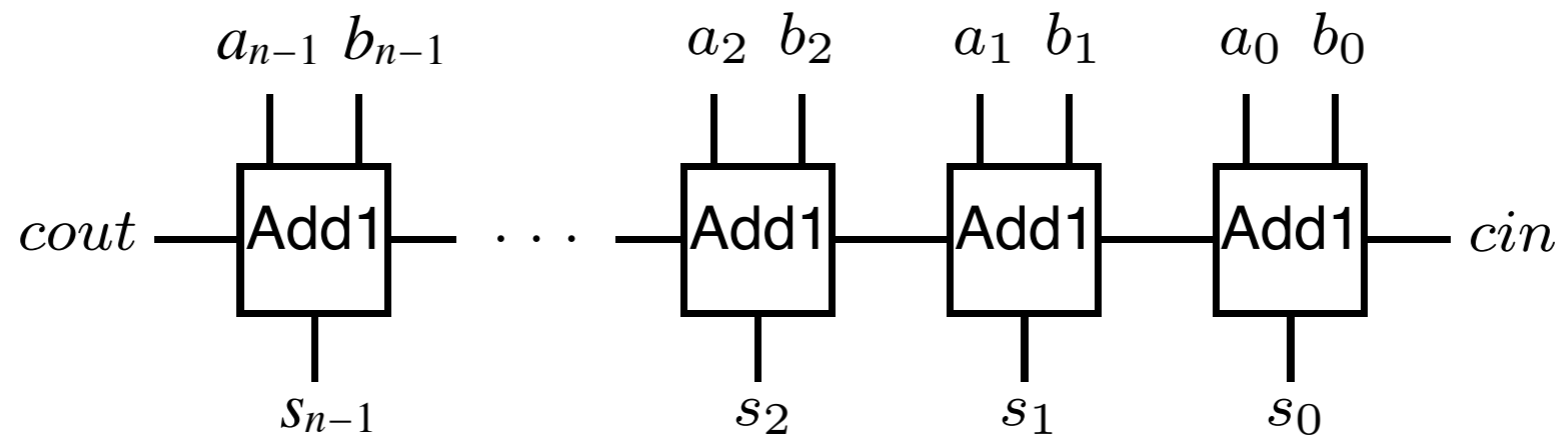
```
Adder.thy (modified)
Adder.thy (~/.Dropbox/ACS/11 - Hardware Verification/)
text{* 1-bit CMOS full adder implementation *}
definition "Add1Imp = (λ(a,b,cin,s,cout).
  ∃p0 p1 p2 p3 p4 p5 p6 p7 p8 p9 p10 p11.
    Ptran(p1,p0,p2) ∧ Ptran(cin,p0,p3) ∧
    Ptran(b,p2,p3) ∧ Ptran(a,p2,p4) ∧
    Ptran(p1,p3,p4) ∧ 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,s) ∧
    Ntran(p4,s,p11) ∧ Gnd(p11) ∧
    Ptran(p1,p0,cout) ∧ Ntran(p1,cout,p11))"
text{* Verification of CMOS full adder *}
lemma Add1Correct: "Add1Imp(a,b,cin,s,cout) = Add1Spec(a,b,cin,s,cout)"
by (simp add: Pwr_def Gnd_def Ntran_def Ptran_def Add1Spec_def
    Add1Imp_def bit_val_def ex_bool_eq) □
```

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

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

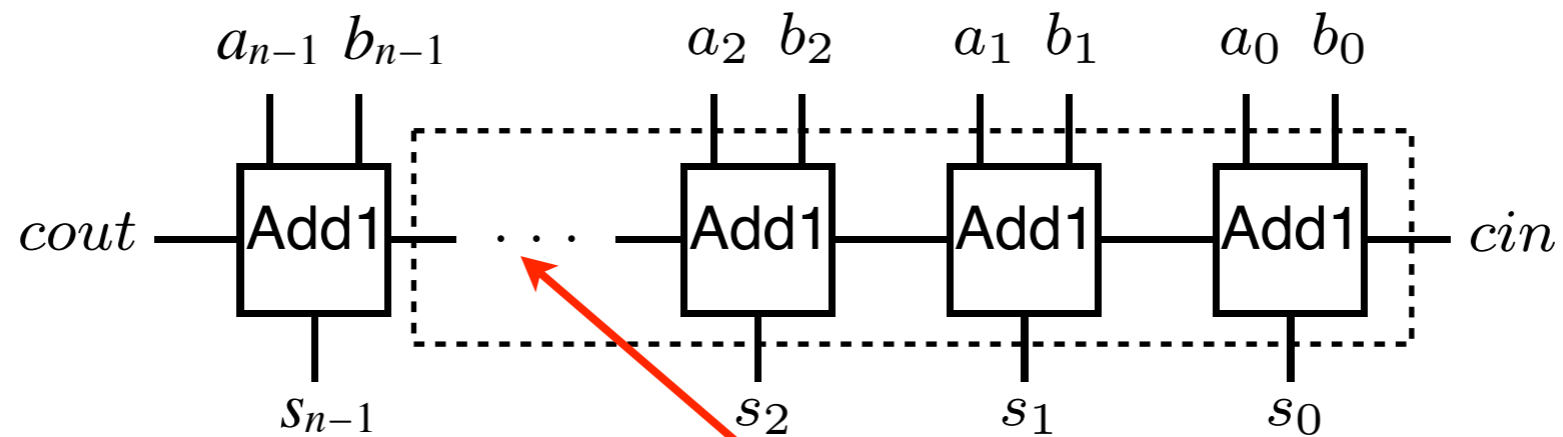
$$(2^n \times cout) + s = a + b + cin$$

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



```
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

```
Lemma AdderCorrect:
  "AdderImp n (a, b, cin, s, cout)  $\implies$  AdderSpec n (a, b, cin, s, cout)"
proof (induction 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, s, c)"
  and AddI: "AddIImp (a n, b n, c, s n, cout)"
  by (auto intro: Suc)
```

Auto update Update Detach 100%

- AdderImp n (a, b, cin, s, ?cout) \implies AdderSpec n (a, b, cin, s, ?cout)
- AdderImp (Suc n) (a, b, cin, s, cout)

goal (1 subgoal):

1. $\bigwedge n$ cout.
[[\bigwedge cout. AdderImp n (a, b, cin, s, cout) \implies AdderSpec n (a, b, cin, s, cout);
AdderImp (Suc n) (a, b, cin, s, cout)]]
 \implies AdderSpec (Suc n) (a, b, cin, s, cout)

Find Output Sledgehammer Symbols

84,15 (2948/5714) (isabelle,sidekick,UTF-8-Isabelle)Nmr o UG 263/526MB 16:27

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, s, cout)  $\implies$  AdderSpec n (a, b, cin, s, cout)"
proof (induction 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, s, c)"
  and AddI: "AddIImp (a n, b n, c, s n, cout)"
  by (auto intro: Suc)
```

internal wire

holds by the induction hyp

name of the induction hyp

```
▪ AdderImp n (a, b, cin, s, ?cout)  $\implies$  AdderSpec n (a, b, cin, s, ?cout)
▪ AdderImp (Suc n) (a, b, cin, s, cout)
```

```
goal (1 subgoal):
  1.  $\bigwedge n$  cout.
    [[ $\bigwedge$ cout. AdderImp n (a, b, cin, s, cout)  $\implies$  AdderSpec n (a, b, cin, s, cout);
      AdderImp (Suc n) (a, b, cin, s, cout)]
     $\implies$  AdderSpec (Suc n) (a, b, cin, s, cout)
```

By assumption, we have `AdderImp (Suc n)` and therefore both `AdderImp n` and `AddIImp`. 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

```
then obtain c
  where AddS: "AdderSpec n (a, b, cin, s, c)"
  and AddI: "AddIImp (a n, b n, c, s n, cout)"
  by (auto intro: Suc)
have "bit_val (s n) * (2 ^ n) + bit_val cout * (2 * 2 ^ n) =
      (bit_val (s 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 AddI by (simp add: AddICorrect AddISpec_def)
finally show "AdderSpec (Suc n) (a, b, cin, s, cout)" using AddS
```

rearranging the terms

replacing outputs by inputs

calculation:

$$\text{bit_val } (s \ n) * 2 \wedge n + \text{bit_val } \text{cout} * (2 * 2 \wedge n) =$$
$$(\text{bit_val } c + (\text{bit_val } (a \ n) + \text{bit_val } (b \ n))) * 2 \wedge n$$

proof (chain): step 16

picking this:

$$\text{bit_val } (s \ n) * 2 \wedge n + \text{bit_val } \text{cout} * (2 * 2 \wedge n) =$$
$$(\text{bit_val } c + (\text{bit_val } (a \ n) + \text{bit_val } (b \ n))) * 2 \wedge n$$

95,10 (3404/5714) (isabelle,sidekick,UTF-8-Isabelle)Nm r o UG 314/526MB 16:29

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 `AddIImp` by `AddISpec`, 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.

Partial Correctness is Proved!

```
text{* Partial correctness of ripple-carry adder for all n by induction *}
lemma AdderCorrect:
  "AdderImp n (a, b, cin, s, cout) ==> AdderSpec n (a, b, cin, s, cout)"
proof (induction 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, s, c)"
      and Add1: "Add1Imp (a n, b n, c, s n, cout)"
      by (auto intro: Suc)
    have "bit_val (s n) * (2 ^ n) + bit_val cout * (2 * 2 ^ n) =
      (bit_val (s 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, s, 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

The screenshot shows a theorem prover window titled "Adder.thy (modified)". The main text area contains the following code:

```
Lemma AdderSpec_Suc:
  "AdderSpec (Suc n) (a, b, cin, s, cout) =
  (∃c. AdderSpec n (a, b, cin, s, c) & Add1Spec (a n, b n, c, s n, cout))"
apply (auto simp add: AdderSpec_def Add1Spec_def ex_bool_eq bit_val_def)
```

Below the code, a goal is displayed with 16 subgoals. The first six subgoals are:

1. $[a\ n; b\ n; s\ n; \neg\ cout; cin; bits_val\ s\ n = Suc\ (2\ ^\ n + (bits_val\ a\ n + bits_val\ b\ n))]$
 $\Rightarrow False$
2. $[a\ n; b\ n; s\ n; \neg\ cout; \neg\ cin; bits_val\ s\ n = 2\ ^\ n + (bits_val\ a\ n + bits_val\ b\ n)]$
 $\Rightarrow False$
3. $[a\ n; b\ n; \neg\ s\ n; \neg\ cout; cin;$
 $bits_val\ s\ n = Suc\ (2\ ^\ n + bits_val\ a\ n + (2\ ^\ n + bits_val\ b\ n))]$
 $\Rightarrow False$
4. $[a\ n; b\ n; \neg\ s\ n; \neg\ cout; \neg\ cin;$
 $bits_val\ s\ n = 2\ ^\ n + bits_val\ a\ n + (2\ ^\ n + bits_val\ b\ n)]$
 $\Rightarrow False$
5. $[a\ n; \neg\ b\ n; s\ n; cout; cin;$
 $2 * 2\ ^\ n + bits_val\ s\ n = Suc\ (bits_val\ a\ n + bits_val\ b\ n)]$
 $\Rightarrow False$
6. $[a\ n; \neg\ b\ n; s\ n; cout; \neg\ cin; 2 * 2\ ^\ n + bits_val\ s\ n = bits_val\ a\ n + bits_val\ b\ n]$

Annotations include a blue box "just need to prove this..." with an arrow pointing to the lemma statement, and another blue box "HELP!!" with four arrows pointing to the first four subgoals. The interface also shows "Auto update" checked, "Update" and "Detach" buttons, a "100%" zoom level, and a sidebar with "Documentation", "Sidekick", and "Theories". The status bar at the bottom shows "144,13 (5042/5799)" and "(isabelle,sidekick,UTF-8-Isabelle)Nm r o UG 305/525MB 16:35".

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 (induction n, auto simp add: bit_val_def)

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

lemma AdderCorrect2:
  ...

using this:
  ▪ bits_val a n < 2 ^ n
  ▪ bits_val b n < 2 ^ n
  ▪ bits_val s n < 2 ^ n

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

a trivial upper bound on the value of a bit string

inserting three instances of that fact

now the proof is trivial

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 theorem prover interface with a file named 'Adder.thy (modified)'. The main editor displays a lemma proof:

```
Lemma AdderCorrect2:
  "AdderSpec n (a, b, cin, s, cout)  $\implies$  AdderImp n (a, b, cin, s, cout)"
proof (induction n arbitrary: cout)
  case 0 thus ?case
    by (simp add: AdderSpec_def)
next
  case (Suc n)
  thus "AdderImp (Suc n) (a, b, cin, s, cout)" using Suc
    by (auto simp add: AdderSpec_Suc Add1Correct)
qed
```

A blue callout box on the right side of the editor contains the text: "The implementation and specification are equivalent!".

Below the main editor, there is a section titled 'using this:' containing a list of logical equivalences:

- $\text{AdderSpec } n \text{ (a, b, cin, s, ?cout)} \implies \text{AdderImp } n \text{ (a, b, cin, s, ?cout)}$
- $\text{AdderSpec (Suc } n \text{) (a, b, cin, s, cout)}$
- $\text{AdderSpec } n \text{ (a, b, cin, s, ?cout)} \implies \text{AdderImp } n \text{ (a, b, cin, s, ?cout)}$
- $\text{AdderSpec (Suc } n \text{) (a, b, cin, s, cout)}$

Below the list, a goal is shown:

```
goal (1 subgoal):
  1. AdderImp (Suc n) (a, b, cin, s, cout)
```

The interface also shows a status bar at the bottom with the text: '155,57 (5433/5493) (isabelle,sidekick,UTF-8-Isabelle)Nm r o UG 396/502MB 16:54'.

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 *thm₁ thm₂ thm₃*]
discharges premises from left to right
- *thm* [simplified]
applies the simplifier to *thm*
- *thm* [*attr₁, attr₂, attr₃*]
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

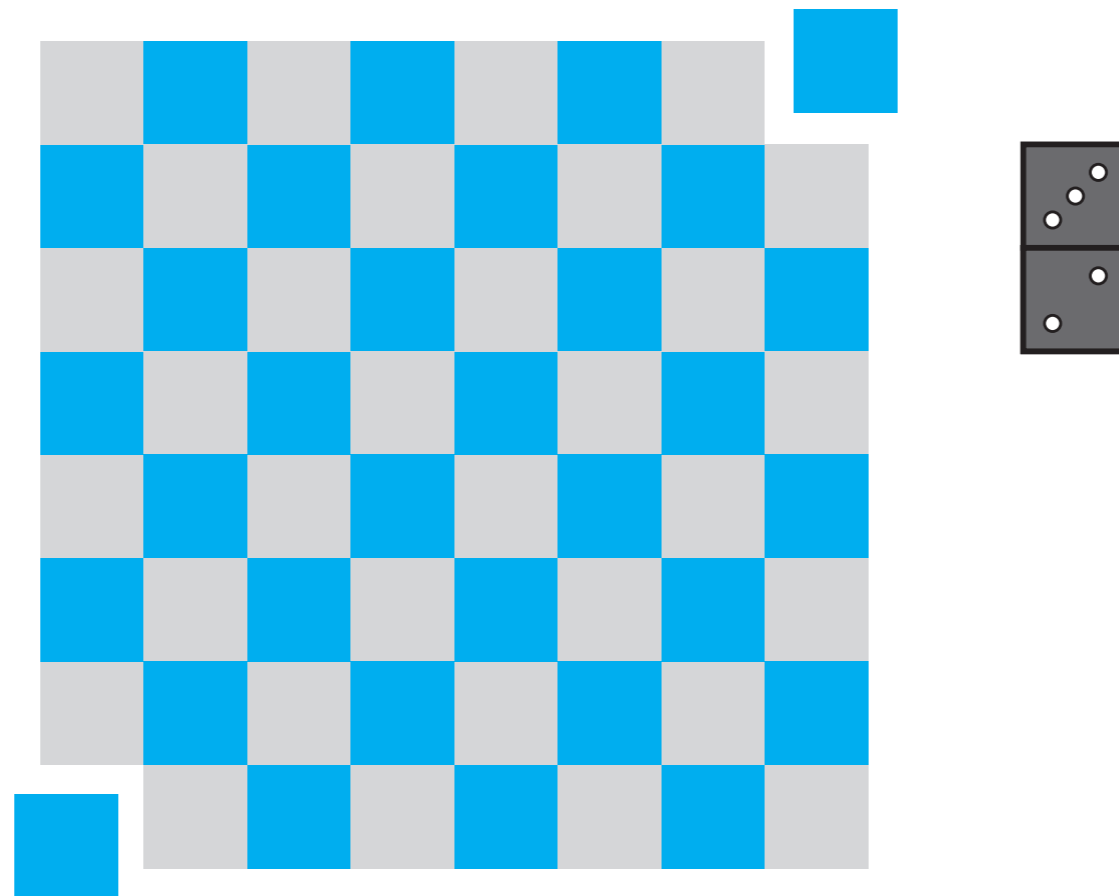
Lawrence C Paulson
Computer Laboratory
University of Cambridge

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

The screenshot shows a theorem prover interface with a file named `Tilings.thy`. The code defines an inductive set `tiling` for a set `A`. The code is as follows:

```
text {* Originated by Max Black. Popularized by J McCarthy. *}

section{* Inductive Tiling *}

inductive_set tiling :: "'a set set => 'a set set" for A
where
  empty : "{} ∈ tiling A"
| Un : "[ a ∈ A; t ∈ tiling A; a ∩ t = {} ] => a ∪ t ∈ tiling A"

declare tiling.intros [intro]
```

Four callouts with red arrows explain parts of the code:

- given a set of tiles...** points to the `for A` part of the `inductive_set` declaration.
- the empty set and aut can be tiled** points to the `empty` and `Un` clauses of the `inductive_set`.
- we give the introduction rules to auto and blast** points to the `declare tiling.intros [intro]` line.

The interface also shows a sidebar with `Documentation`, `Sidekick`, and `Theories`. At the bottom, there are tabs for `Find`, `Output`, `Sledgehammer`, and `Symbols`. The status bar at the bottom indicates `5,61 (163/4287)` and `(isabelle,sidekick,UTF-8-Isabelle)Nm r o UC 335/50 MB 17:05`.

Simple Proofs about Tilings

The screenshot shows a theorem prover interface with two lemmas. The first lemma, `tiling_UnI`, has its `by` clause annotated with a callout: "giving the theorem to auto and blast...". The second lemma, `tiling_finite`, has its `assumes` clause annotated with "referring to the theorem's assumptions" and its `by` clause annotated with "set: another way to specify the induction". A yellow highlight under the `by` clause of the second lemma is annotated with "a comma can join two methods". The interface includes a status bar at the bottom with "24,1 (684/3956)" and "(isabelle,sidekick,UTF-8-Isabelle)Nm r o UG 273/521MB 17:06".

```
lemma tiling_UnI [intro]:  
  "[ t1 ∈ tiling A; t2 ∈ tiling A; t1 ∩ t2 = {} ] ⇒ t1 ∪ t2 ∈ tiling A"  
by (induction rule: tiling.induct, auto simp add: Un_assoc)  
  
lemma tiling_finite:  
  assumes "∧a. a ∈ A ⇒ finite a"  
  shows "t ∈ tiling A ⇒ finite t"  
by (induction set: tiling, auto simp add: assms)
```

set: another way to specify the induction

a comma can join two methods

giving the theorem to auto and blast...

referring to the theorem's assumptions

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

```
Tilings.thy (modified)
Tilings.thy (~/.Dropbox/ACS/12 - Mutilated chessboard/)
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

```
Tilings.thy (modified)
Tilings.thy (~/.Dropbox/ACS/12 - Mutilated chessboard/)

text {*Sets of white or black squares *}

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

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

abbreviation
  blacks :: "(nat x 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)
```

colours defined using modular arithmetic

abbreviations provide notation

case analysis on the named set

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

```
Tilings.thy (modified)
Tilings.thy (~/.Dropbox/ACS/12 - Mutilated chessboard/)

section{* Chess Boards *}

lemma dominoes_tile_row: "{i} × {0..< 2*n} ∈ tiling domino"
proof (induction n)
  case 0 show ?case by auto
next
  case (Suc n)
  have "{i} × {0..< 2 * Suc n} = {(i, 2*n), (i, Suc(2*n))} ∪ ({i} × {0..< 2*n})"
  by auto
  also have "... ∈ tiling domino"
  by (rule tiling.intros, auto intro: domino.intros Suc)
  finally show ?case .
qed

lemma Suc_by_board:
  "{0..< Suc n} × B = ({0..< n} × B) ∪ ({n} × B)"
by auto

lemma dominoes_tile_matrix: "{0..< m} × {0..< 2*n} ∈ tiling domino"
by (induction m, auto simp add: Suc_by_board dominoes_tile_row)
```

$\{i..<j\} = \{i, \dots, j-1\}$

even-length rows can be tiled

even-length blocks can be tiled

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 (induction 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)
  then show ?case using Un
    by (auto simp add: Int_Un_distrib card_insert_if)
```

using this:

- whites ∩ d = {(i, j)}
- blacks ∩ d = {(m, n)}
- d ∈ domino
- t ∈ tiling domino
- d ∩ t = {}
- card (whites ∩ t) = card (blacks ∩ t)

use the result of "obtain"

... as well as the induction hyps

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

```
Tilings.thy (modified)
Tilings.thy (~/.Dropbox/ACS/12 - Mutilated chessboard/)

theorem gen_mutil_not_tiling:
  assumes "t ∈ tiling domino" "sqᵢ ⊆ whites ∩ t" "sqᵢ ≠ {}"
  shows "(t - sqᵢ) ∉ tiling domino"
proof
  assume tm: "t - sqᵢ ∈ tiling domino"
  have fsqs: "finite sqᵢ" using assms
    by (metis Int_subset_iff finite_subset tiling_finite [OF domino_finite])
  hence c: "0 < card sqᵢ" "0 < card (whites ∩ t)" using assms
    by (auto simp add: card_gt_0_iff)
  have "card (whites ∩ (t - sqᵢ)) = card ((whites ∩ t) - sqᵢ)"
    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 - sqᵢ))"
  proof -
    have "blacks ∩ (t - sqᵢ) = blacks ∩ t" using assms
      by (force simp add: coloured_def)
    then show ?thesis by simp
  qed
  finally show False using tiling_domino_0_1 [OF tm] by auto
```

default proof of a negation

accumulating some facts

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

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..

```
Tilings.thy (modified)
Tilings.thy (~/.Dropbox/ACS/12 - Mutilated chessboard/)

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

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))} ≠ {}

133,34 (3860/3957) (isabelle,sidekick,UTF-8-Isabelle)Nm r o UC 333/471 MB 17:37
```

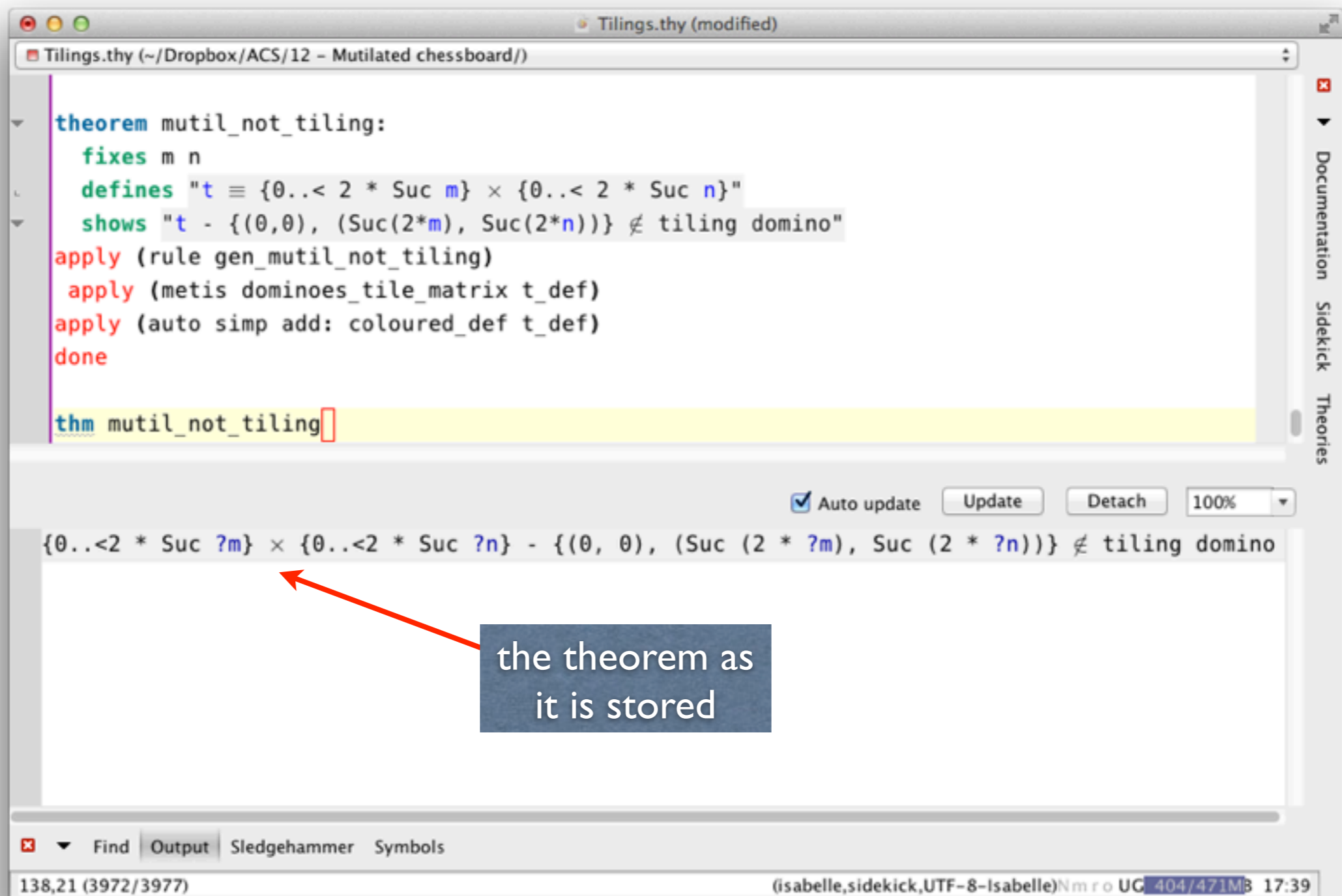
a local constant, t

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 the Isabelle/jEdit interface. The top window displays the source code for a theorem named `mutil_not_tiling`. The code defines a set `t` and shows that it is not a tiling domino. The final line of the theorem is `thm mutil_not_tiling`, which is highlighted in yellow. Below the code editor, the expanded form of the theorem is shown, where the definitions have been substituted. A red arrow points from a text box to the expanded theorem statement.

```
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

thm mutil_not_tiling
```

`{0..2 * Suc ?m} × {0..2 * Suc ?n} - {(0, 0), (Suc (2 * ?m), Suc (2 * ?n))} ∉ tiling domino`

the theorem as it is stored

Note that the final theorem does not mention `defines` or `t`. All such definitions are expanded.

With Isabelle/jEdit, it's necessary to display the theorem explicitly using `thm`.

Finding Structured Proofs

The screenshot shows the Isabelle/Isar IDE with two windows of the file `Tilings.thy`. The top window shows a proof structure:

```
shows "t ∈ tiling A ⇒ finite t"  
proof (induction set: tiling)  
  case empty  
  show ?case by simp  
next  
  case (Un d t)  
  thus ?case  
  apply (rule finite_UnI)
```

The bottom window shows an error message:

```
Failed to apply proof method:  
using this:  
  ▪ d ∈ A  
  ▪ t ∈ tiling A  
  ▪ d ∩ t = {}  
  ▪ finite t  
goal (1 subgoal):  
  1. [d ∈ A; t ∈ tiling A; d ∩ t = {}; finite t]  
     ⇒ finite (d ∪ t)
```

Two callout boxes provide context:

- A blue box with the text "you need apply -" has an arrow pointing to the `apply -` line in the proof.
- A blue box with the text "It's okay to fool around with apply, but what if this keeps happening?" has an arrow pointing to the error message.

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

Other Facets of Isabelle

- *Document preparation*: you can generate L^AT_EX 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 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