Interactive Formal Verification
3: Elementary Proof

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

Goals and Subgoals

Goals and Subgoals

- We start with one subgoal: the statement to be proved.

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.

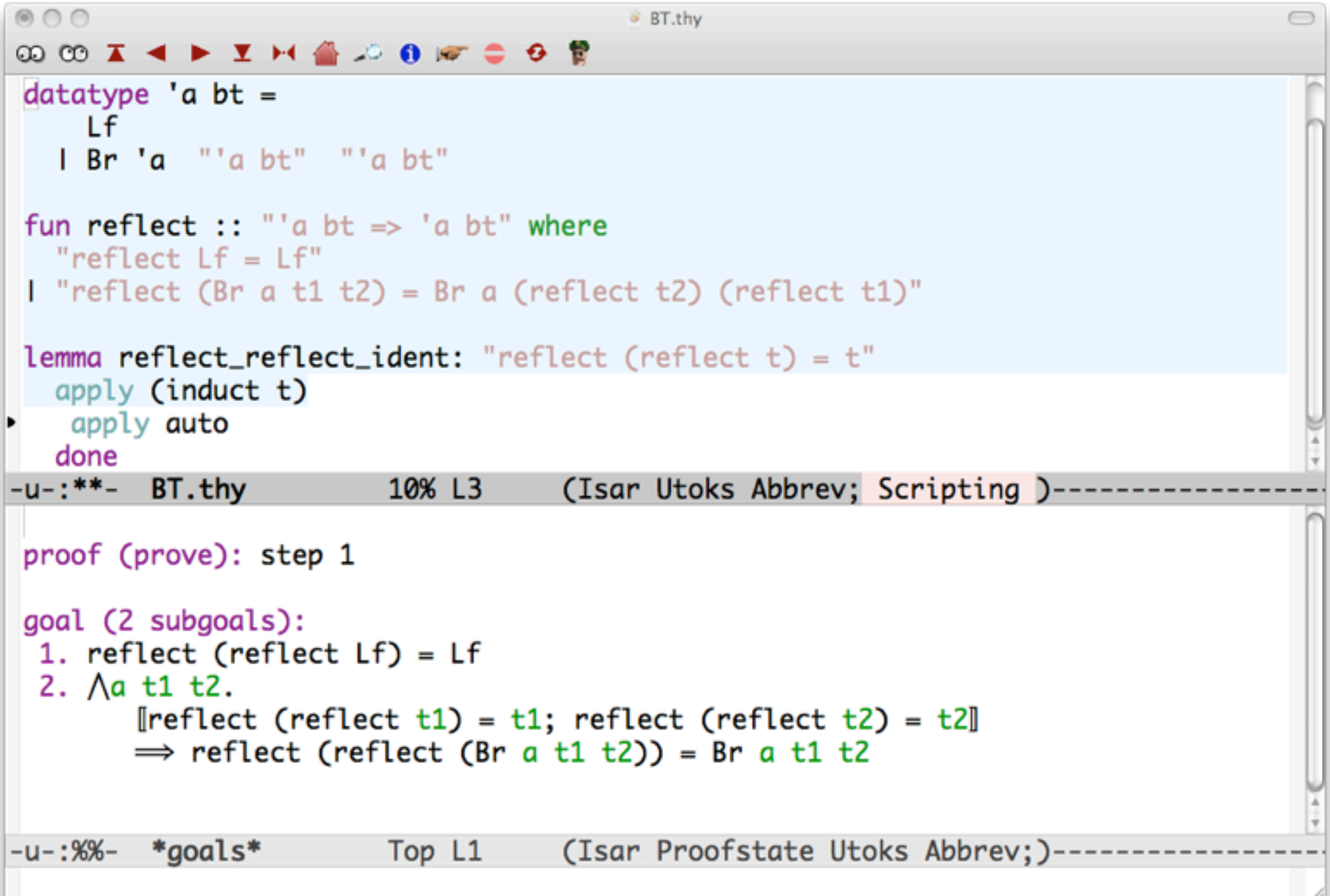
Goals and Subgoals

- We start with one subgoal: the statement to be proved.
- Proof tactics and methods typically replace a single subgoal by zero or more new subgoals.
- Certain methods, notably `auto` and `simp_all`, operate on all outstanding subgoals.

Goals and Subgoals

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

Structure of a Subgoal



Structure of a Subgoal

```
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
```

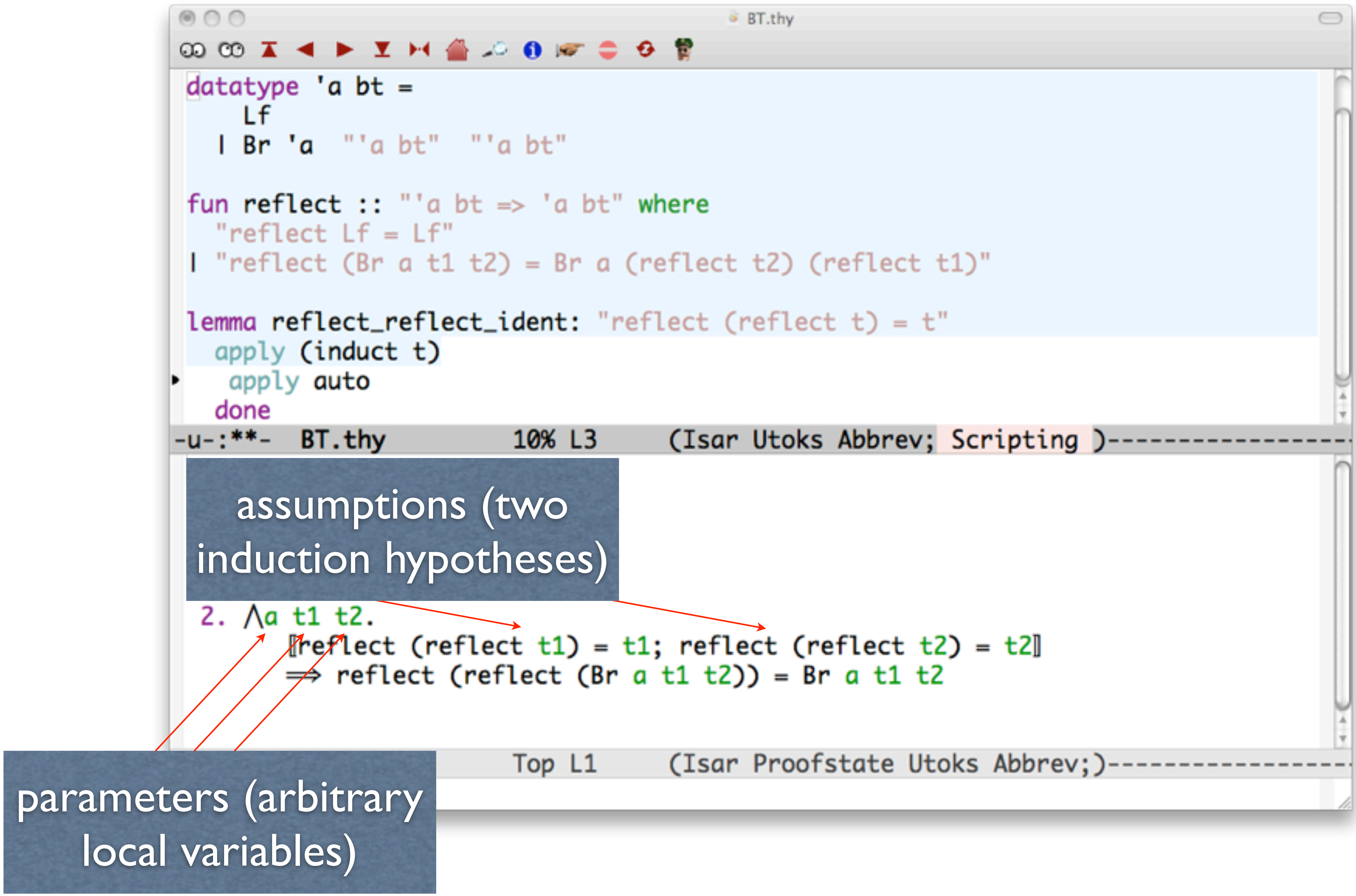
```
-u-:**-  BT.thy        10% L3      (Isar Utoks Abbrev; Scripting )----------------
```

assumptions (two induction hypotheses)

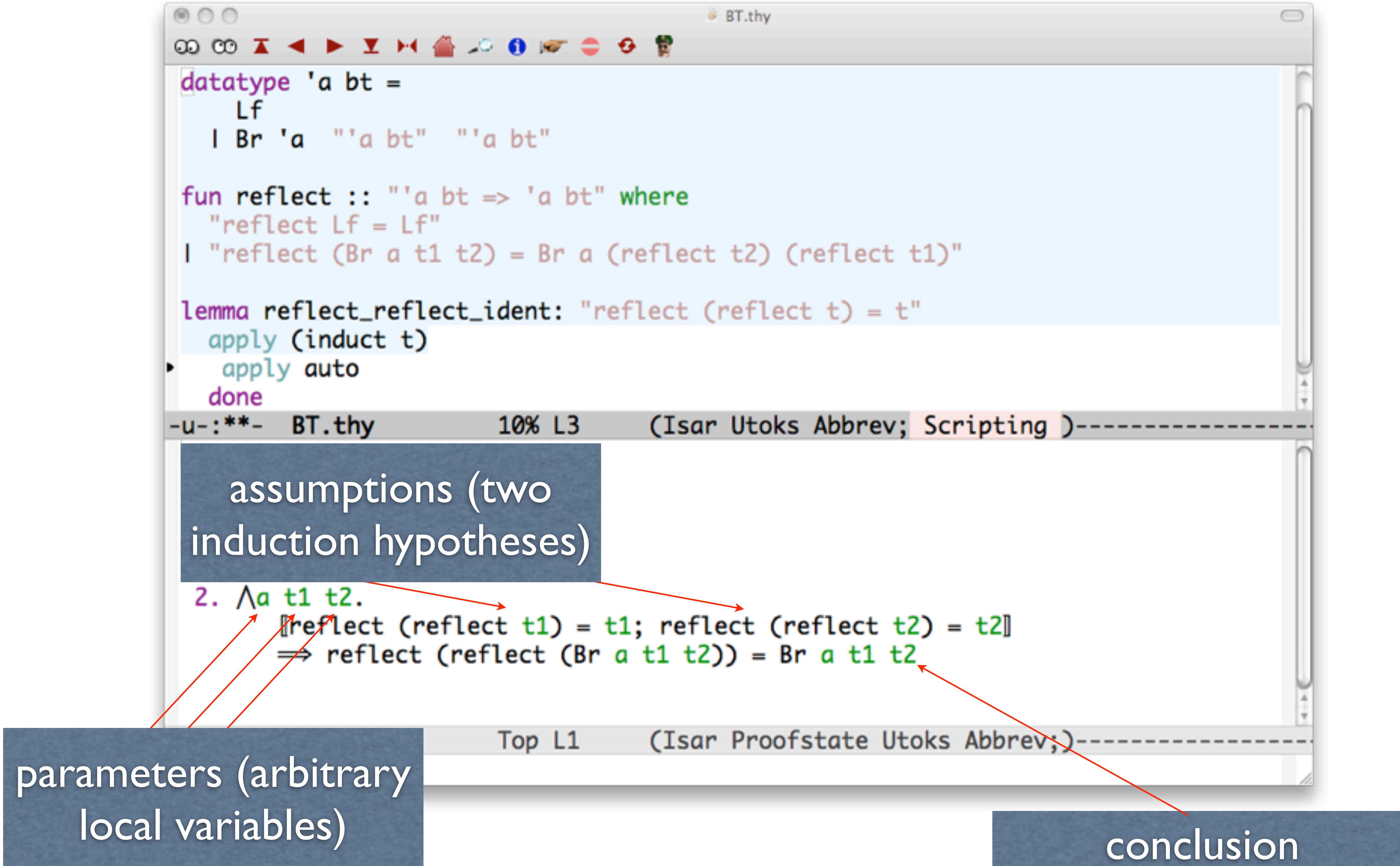
```
 2. ⋀a t1 t2.
       ⟦reflect (reflect t1) = t1; reflect (reflect t2) = t2⟧
       ⟹ reflect (reflect (Br a t1 t2)) = Br a t1 t2
```

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

Structure of a Subgoal



Structure of a Subgoal



Proof by Rewriting

```
app (Cons x xs) ys = Cons x (app xs ys)
   rev (Cons x xs) = app (rev xs) (Cons x Nil)
   rev (app xs ys) = app (rev ys) (rev xs)
app (app xs ys) zs = app xs (app ys zs)
```

Proof by Rewriting

```
app (Cons x xs) ys = Cons x (app xs ys)
   rev (Cons x xs) = app (rev xs) (Cons x Nil)
   rev (app xs ys) = app (rev ys) (rev xs)
app (app xs ys) zs = app xs (app ys zs)
```

recursive defns

Proof by Rewriting

```
app (Cons x xs) ys = Cons x (app xs ys)
   rev (Cons x xs) = app (rev xs) (Cons x Nil)
   rev (app xs ys) = app (rev ys) (rev xs)
app (app xs ys) zs = app xs (app ys zs)
```

recursive defns

induction hyp

Proof by Rewriting

```
app (Cons x xs) ys = Cons x (app xs ys)
   rev (Cons x xs) = app (rev xs) (Cons x Nil)
   rev (app xs ys) = app (rev ys) (rev xs)
app (app xs ys) zs = app xs (app ys zs)
```

recursive defns

induction hyp

lemma

Proof by Rewriting

```
app (Cons x xs) ys ⇒ Cons x (app xs ys)
   rev (Cons x xs) ⇒ app (rev xs) (Cons x Nil)
   rev (app xs ys) ⇒ app (rev ys) (rev xs)
app (app xs ys) zs ⇒ app xs (app ys zs)
```

recursive defns

induction hyp

lemma

Proof by Rewriting

```
app (Cons x xs) ys  ⇒ Cons x (app xs ys)
    rev (Cons x xs) ⇒ app (rev xs) (Cons x Nil)
    rev (app xs ys) ⇒ app (rev ys) (rev xs)
app (app xs ys) zs  ⇒ app xs (app ys zs)
```

recursive defns

induction hyp

lemma

```
rev (app (Cons a xs) ys) = app (rev ys) (rev (Cons a xs))
```

Proof by Rewriting

```
app (Cons x xs) ys → Cons x (app xs ys)
   rev (Cons x xs) → app (rev xs) (Cons x Nil)
   rev (app xs ys) → app (rev ys) (rev xs)
app (app xs ys) zs → app xs (app ys zs)
```

recursive defns

induction hyp

lemma

```
rev (app (Cons a xs) ys) = app (rev ys) (rev (Cons a xs))
```

```
rev (app (Cons a xs) ys) =
rev (Cons a (app xs ys)) =
app (rev (app xs ys)) (Cons a Nil) =
app (app (rev ys) (rev xs)) (Cons a Nil) =
app (rev ys) (app (rev xs) (Cons a Nil))
```

Proof by Rewriting

```
app (Cons x xs) ys ⇒ Cons x (app xs ys)
   rev (Cons x xs) ⇒ app (rev xs) (Cons x Nil)
   rev (app xs ys) ⇒ app (rev ys) (rev xs)
app (app xs ys) zs ⇒ app xs (app ys zs)
```

recursive defns

induction hyp

lemma

```
rev (app (Cons a xs) ys) = app (rev ys) (rev (Cons a xs))
```

```
rev (app (Cons a xs) ys) =
rev (Cons a (app xs ys)) =
app (rev (app xs ys)) (Cons a Nil) =
app (app (rev ys) (rev xs)) (Cons a Nil) =
app (rev ys) (app (rev xs) (Cons a Nil))
```

Proof by Rewriting

```
app (Cons x xs) ys → Cons x (app xs ys)
    rev (Cons x xs) → app (rev xs) (Cons x Nil)
    rev (app xs ys) → app (rev ys) (rev xs)
app (app xs ys) zs → app xs (app ys zs)
```

recursive defns

induction hyp

lemma

```
rev (app (Cons a xs) ys) = app (rev ys) (rev (Cons a xs))
```

```
rev (app (Cons a xs) ys) =
rev (Cons a (app xs ys)) =
app (rev (app xs ys)) (Cons a Nil) =
app (app (rev ys) (rev xs)) (Cons a Nil) =
app (rev ys) (app (rev xs) (Cons a Nil))
```

Proof by Rewriting

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

```
rev (app (Cons a xs) ys) = app (rev ys) (rev (Cons a xs))
```

```
rev (app (Cons a xs) ys) =
rev (Cons a (app xs ys)) =
app (rev (app xs ys)) (Cons a Nil) =
app (app (rev ys) (rev xs)) (Cons a Nil) =
app (rev ys) (app (rev xs) (Cons a Nil))
```

Proof by Rewriting

```
app (Cons x xs) ys → Cons x (app xs ys)
   rev (Cons x xs) → app (rev xs) (Cons x Nil)
   rev (app xs ys) → app (rev ys) (rev xs)
app (app xs ys) zs → app xs (app ys zs)
```

recursive defns

induction hyp

lemma

```
rev (app (Cons a xs) ys) = app (rev ys) (rev (Cons a xs))
```

```
rev (app (Cons a xs) ys) =
rev (Cons a (app xs ys)) =
app (rev (app xs ys)) (Cons a Nil) =
app (app (rev ys) (rev xs)) (Cons a Nil) =
app (rev ys) (app (rev xs) (Cons a Nil))
```

Proof by Rewriting

```
app (Cons x xs) ys ⇒ Cons x (app xs ys)
   rev (Cons x xs) ⇒ app (rev xs) (Cons x Nil)
   rev (app xs ys) ⇒ app (rev ys) (rev xs)
app (app xs ys) zs ⇒ app xs (app ys zs)
```

recursive defns

induction hyp

lemma

```
rev (app (Cons a xs) ys) = app (rev ys) (rev (Cons a xs))
```

```
rev (app (Cons a xs) ys) =
rev (Cons a (app xs ys)) =
app (rev (app xs ys)) (Cons a Nil) =
app (app (rev ys) (rev xs)) (Cons a Nil) =
app (rev ys) (app (rev xs) (Cons a Nil))
```

```
app (rev ys) (rev (Cons a xs)) =
app (rev ys) (app (rev xs) (Cons a Nil))
```

Proof by Rewriting

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

```
rev (app (Cons a xs) ys) = app (rev ys) (rev (Cons a xs))
```

```
rev (app (Cons a xs) ys) =
rev (Cons a (app xs ys)) =
app (rev (app xs ys)) (Cons a Nil) =
app (app (rev ys) (rev xs)) (Cons a Nil) =
app (rev ys) (app (rev xs) (Cons a Nil))
```

```
                    app (rev ys) (rev (Cons a xs)) =
                    app (rev ys) (app (rev xs) (Cons a Nil))
```

Rewriting with Equivalences

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

Rewriting with Equivalences

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

introduces a case split on the sign of c

Rewriting with Equivalences

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

introduces a case split on the sign of c

- Logical equivalencies are just boolean equations.

Rewriting with Equivalences

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

introduces a case split on the sign of c

- Logical equivalencies are just boolean equations.
- They lead to a clear and simple proof style.

Rewriting with Equivalences

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

introduces a case split on the sign of c

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

Automatic Case Splitting

Automatic Case Splitting

Simplification will replace

Automatic Case Splitting

Simplification will replace

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

Automatic Case Splitting

Simplification will replace

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

by

Automatic Case Splitting

Simplification will replace

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

by

$$(b \to P(x)) \land (\neg b \to P(y))$$

Automatic Case Splitting

Simplification will replace

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

by

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

- By default, this only happens when simplifying the conclusion.

Automatic Case Splitting

Simplification will replace

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

by

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

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

Conditional Rewrite Rules

```
xs ≠ [] ⇒ hd (xs @ ys) = hd xs

n ≤ m ⇒ (Suc m) - n = Suc (m - n)

[|a ≠ 0; b ≠ 0|] ⇒ b / (a*b) = 1 / a
```

Conditional Rewrite Rules

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

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

$$[\![\texttt{a} \neq \texttt{0; b} \neq \texttt{0}]\!] \Rightarrow \texttt{b / (a*b) = 1 / a}$$

- *First* match the left-hand side, *then* **recursively** prove the conditions by simplification.

Conditional Rewrite Rules

```
xs ≠ [] ⇒ hd (xs @ ys) = hd xs

n ≤ m ⇒ (Suc m) - n = Suc (m - n)

[|a ≠ 0; b ≠ 0|] ⇒ b / (a*b) = 1 / a
```

- *First* match the left-hand side, *then* **recursively** prove the conditions by simplification.
- If successful, applying the resulting rewrite rule.

Conditional Rewrite Rules

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

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

$$[\![\texttt{a} \neq \texttt{0; b} \neq \texttt{0}]\!] \Rightarrow \texttt{b / (a*b)} = \texttt{1 / a}$$

- *First* match the left-hand side, *then* **recursively** prove the conditions by simplification.
- If successful, applying the resulting rewrite rule.

Termination Issues

Termination Issues

- *Looping*: `f(x) = h(g(x))`, `g(x) = f(x+2)`

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!

Termination Issues

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

The Methods `simp` and `auto`

The Methods `simp` and `auto`

- `simp` performs *rewriting* (along with simple arithmetic simplification) on the *first* subgoal

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.

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`

Variations on `simp` and `auto`

```
simp add: add_assoc

simp del: rev_rev (no_asm_simp)

simp (no_asm)

simp_all (no_asm_simp) add: … del: …

auto simp add: … del: …
```

Variations on `simp` and `auto`

using another rewrite rule

```
simp add: add_assoc

simp del: rev_rev (no_asm_simp)

simp (no_asm)

simp_all (no_asm_simp) add: … del: …

auto simp add: … del: …
```

Variations on `simp` and `auto`

using another rewrite rule

omitting a certain rule

```
simp add: add_assoc

simp del: rev_rev (no_asm_simp)

simp (no_asm)

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

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

Variations on `simp` and `auto`

using another rewrite rule

omitting a certain rule

```
simp add: add_assoc

simp del: rev_rev (no_asm_simp)

simp (no_asm)

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

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

not simplifying the assumptions

Variations on `simp` and `auto`

using another rewrite rule

omitting a certain rule

```
simp add: add_assoc

simp del: rev_rev (no_asm_simp)

simp (no_asm)

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

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

not simplifying the assumptions

ignoring all assumptions

Variations on `simp` and `auto`

using another rewrite rule

omitting a certain rule

```
simp add: add_assoc

simp del: rev_rev (no_asm_simp)

simp (no_asm)

simp_all (no_asm_simp) add: … del: …

auto simp add: … del: …
```

not simplifying the assumptions

ignoring all assumptions

do `simp` for all subgoals

Variations on `simp` and `auto`

```
simp add: add_assoc

simp del: rev_rev (no_asm_simp)

simp (no_asm)

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

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

using another rewrite rule

omitting a certain rule

not simplifying the assumptions

ignoring all assumptions

do `simp` for all subgoals

`auto` with options

Rules for Arithmetic

Rules for Arithmetic

- An identifier can denote a *list* of lemmas.

Rules for Arithmetic

- An identifier can denote a *list* of lemmas.
- `add_ac` and `mult_ac`: associative/commutative properties of addition and multiplication

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

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

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`

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`

- `thm name` prints theorems.

Simple Proof by Induction

Simple Proof by Induction

- State the desired theorem using "`lemma`", with its name and optionally `[simp]`

Simple Proof by Induction

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

Simple Proof by Induction

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

Completing the Proof

Completing the Proof

- Apply "`induct`" with the chosen variable.

Completing the Proof

- Apply “`induct`” with the chosen variable.
- The first subgoal will be the base case, and it should be trivial using “`simp`”.

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.

Completing the Proof

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

Basics of Proof General

Basics of Proof General

- You create or visit an Isabelle theory file within the text editor, Emacs.

Basics of Proof General

- You create or visit an Isabelle theory file within the text editor, Emacs.
- Moving *forward* executes Isabelle commands; the processed text turns blue.

Basics of Proof General

- You create or visit an Isabelle theory file within the text editor, Emacs.
- Moving *forward* executes Isabelle commands; the processed text turns blue.
- Moving *backward* undoes those commands.

Basics of Proof General

- You create or visit an Isabelle theory file within the text editor, Emacs.
- Moving *forward* executes Isabelle commands; the processed text turns blue.
- Moving *backward* undoes those commands.
- *Go to end* processes the entire theory; you can also *go to start*, or go to an arbitrary point in the file.

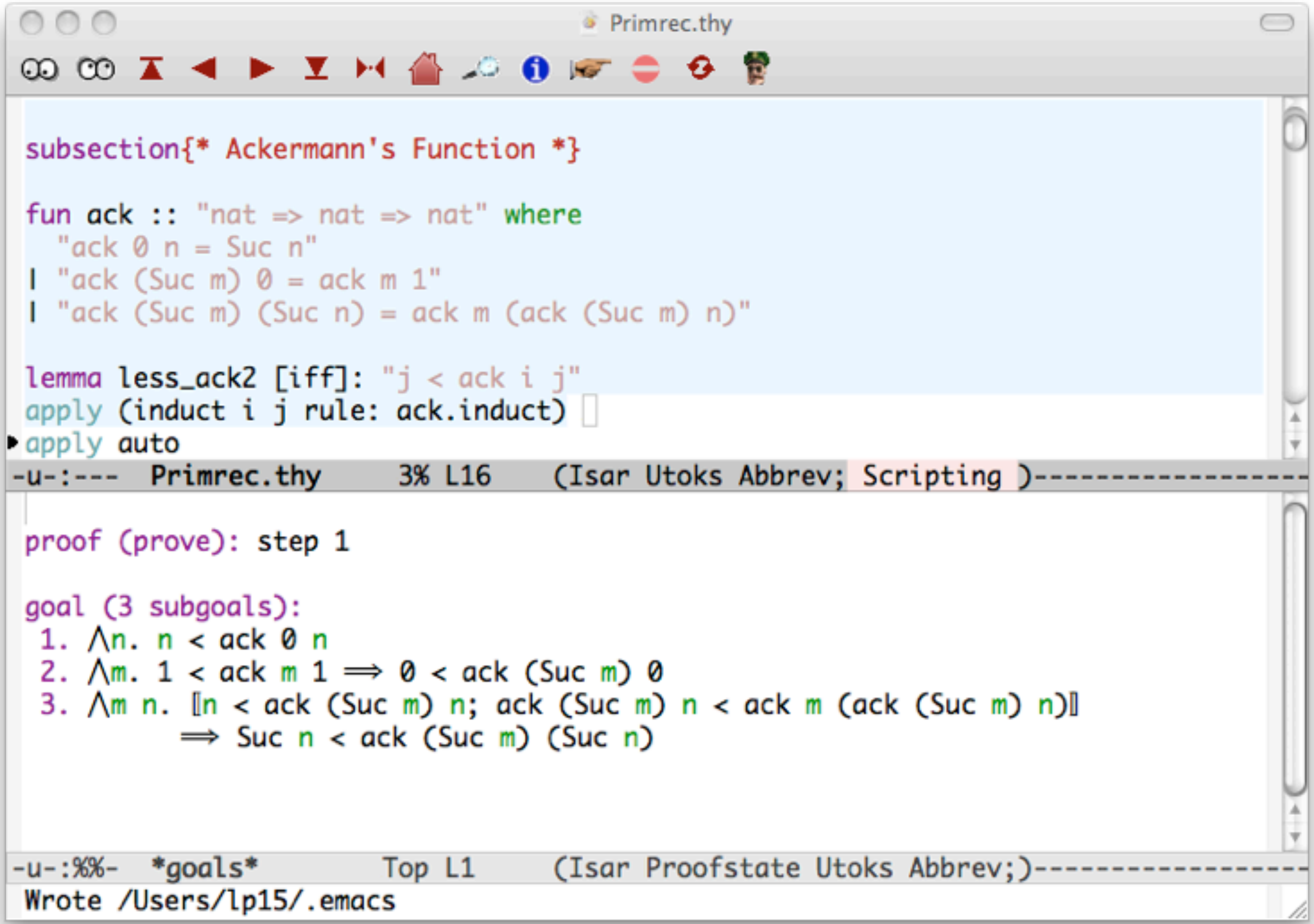
Basics of Proof General

- You create or visit an Isabelle theory file within the text editor, Emacs.
- Moving *forward* executes Isabelle commands; the processed text turns blue.
- Moving *backward* undoes those commands.
- *Go to end* processes the entire theory; you can also *go to start*, or go to an arbitrary point in the file.
- *Go to home* takes you to the end of the blue (processed) region.

Basics of Proof General

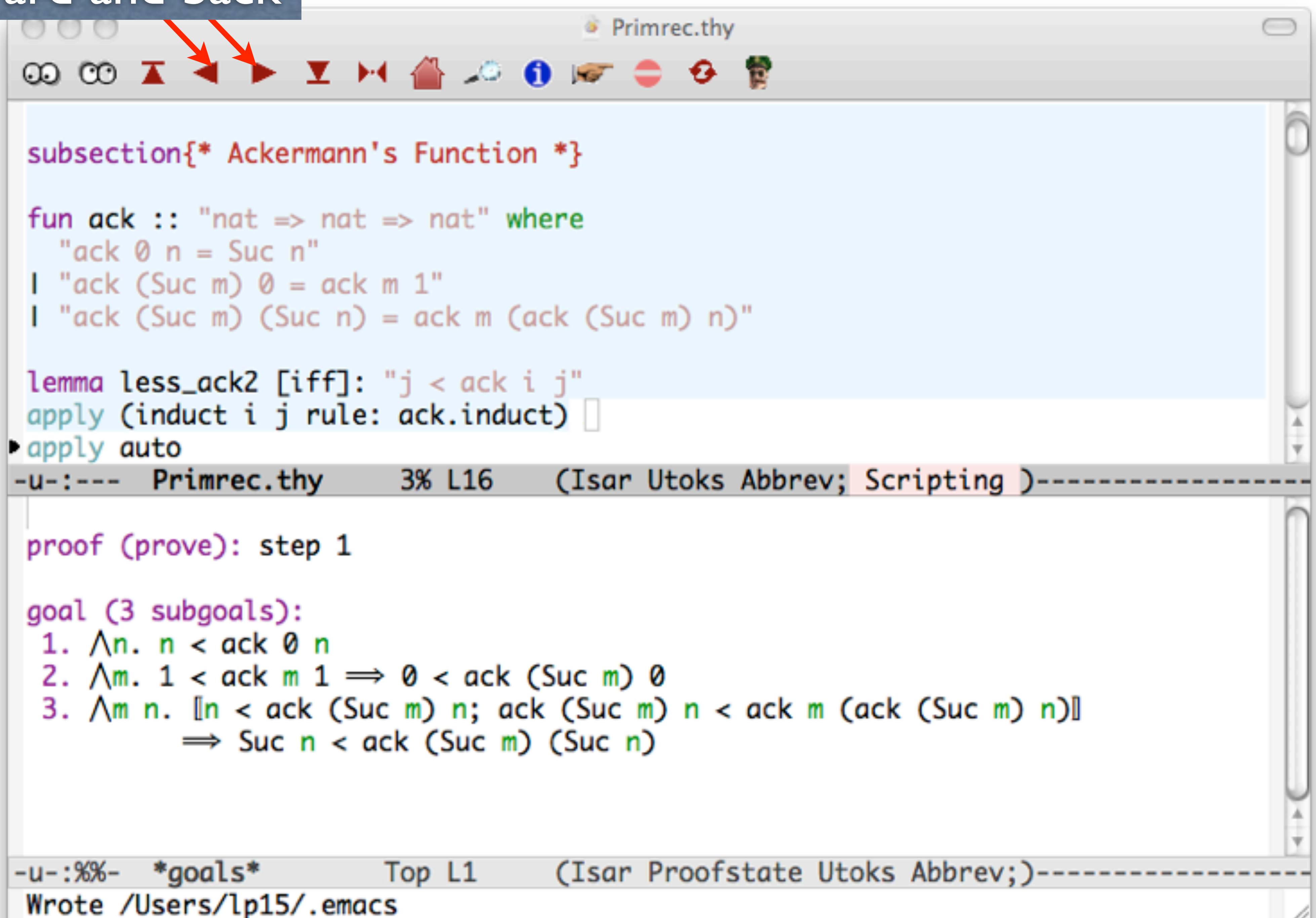
- You create or visit an Isabelle theory file within the text editor, Emacs.
- Moving *forward* executes Isabelle commands; the processed text turns blue.
- Moving *backward* undoes those commands.
- *Go to end* processes the entire theory; you can also *go to start*, or go to an arbitrary point in the file.
- *Go to home* takes you to the end of the blue (processed) region.
- (A different user interface: `isabelle jedit`)

Proof General Tools

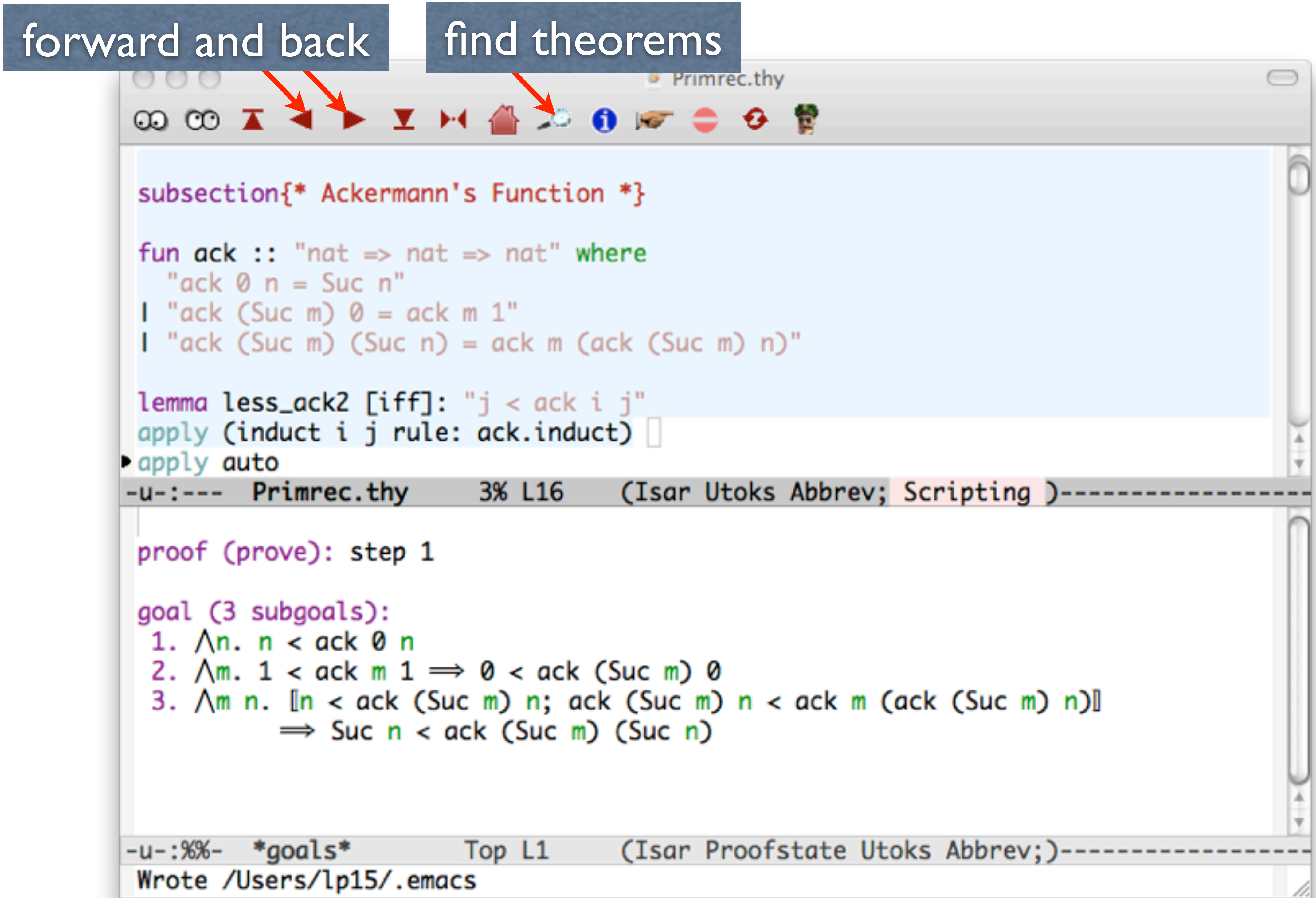


Proof General Tools

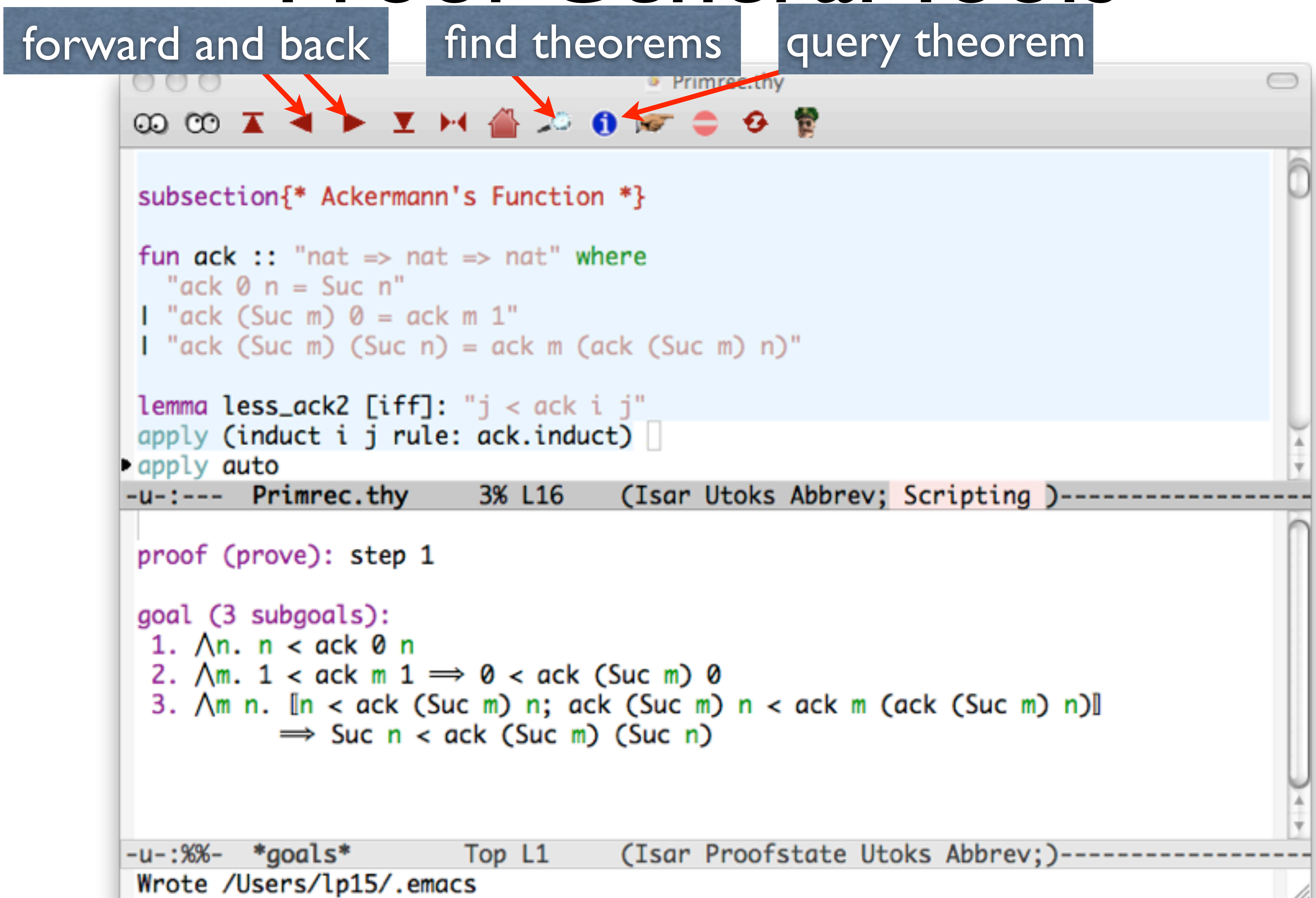
forward and back



Proof General Tools



Proof General Tools



Proof General Tools