Interactive Formal Verification
8: Operational Semantics

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

Overview

Overview

- The operational semantics of programming languages can be given *inductively*.
  - Type checking
  - Expression evaluation
  - Command execution, including concurrency

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.

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          ("_ :== _ " 60)
      | Semi   com com           ("_; _"  [60, 60] 10)
      | Cond   bexp com com      ("IF _ THEN _ ELSE _"  60)
      | While  bexp com          ("WHILE _ DO _"  60)
```

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          ("_ :== _ " 60)
      | Semi   com com           ("_; _"  [60, 60] 10)
      | Cond   bexp com com      ("IF _ THEN _ ELSE _"  60)
      | While  bexp com          ("WHILE _ DO _"  60)
```

Arithmetic & boolean expressions are *functions* over the state

A “Big-Step” Semantics

A "Big-Step" Semantics

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

A "Big-Step" Semantics

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

A "Big-Step" Semantics

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

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

A "Big-Step" Semantics

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

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

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

A "Big-Step" Semantics

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

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

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

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

Formalised Language Semantics

Com.thy

```
text {* The big-step execution relation @{text evalc} is defined inductively *}

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"
```

-u-:--- Com.thy 17% L24 (Isar Utoks Abbrev; Scripting)----------

Wrote /Users/lp15/Dropbox/ACS/8 - Operational Semantics/Com.thy

Formalised Language Semantics

an inductive *predicate*
with special syntax

```
text {* The big-step execution relation @{text evalc} is defined inductively *}

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"
```

```
-u-:---  Com.thy        17% L24    (Isar Utoks Abbrev; Scripting )----------------
Wrote /Users/lp15/Dropbox/ACS/8 - Operational Semantics/Com.thy
```

Formalised Language Semantics

an inductive *predicate*
with special syntax

```
text {* The big-step execution relation @{text evalc} is defined inductively *}

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`

```
-u-:---  Co                                ev; Scripting )----------------
Wrote /Users/tp15/Dropbox/ACS/8 - Operational Semantics/Com.thy
```

Rule Inversion

Rule Inversion

- When $\langle \mathbf{skip}, s \rangle \rightarrow s'$ we know $s = s'$

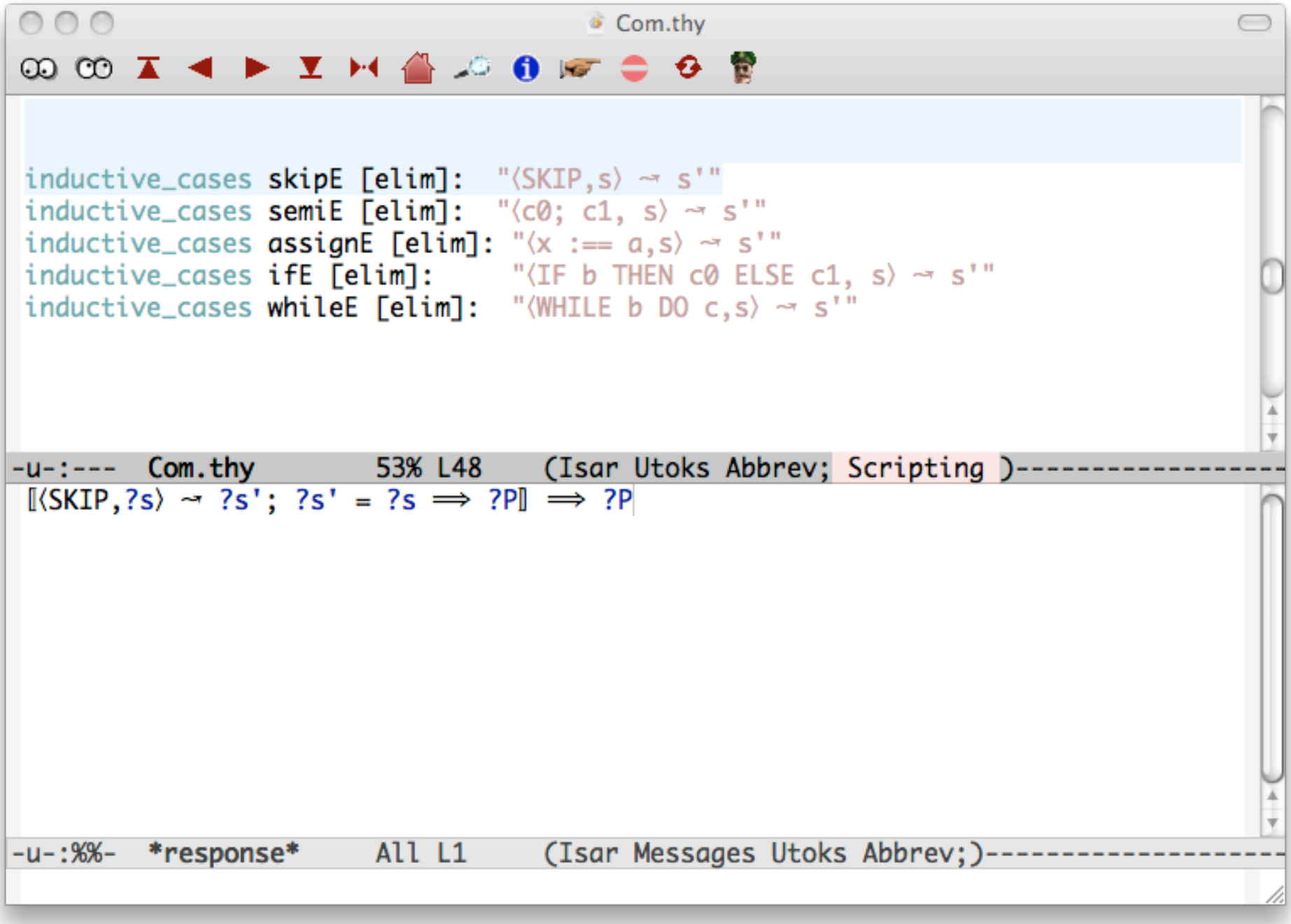
Rule Inversion

- When $\langle \textbf{skip}, s \rangle \rightarrow s'$ we know $s = s'$
- When $\langle \textbf{if } b \textbf{ then } c_0 \textbf{ else } c_1, s \rangle \rightarrow s'$ we know
  - b and $\langle c_0, s \rangle \rightarrow s'$, or...
  - $\neg b$ and $\langle c_1, s \rangle \rightarrow s'$

Rule Inversion

- When $\langle \textbf{skip}, s \rangle \rightarrow s'$ we know $s = s'$
- When $\langle \textbf{if } b \textbf{ then } c_0 \textbf{ else } c_1, s \rangle \rightarrow s'$ 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 in Isabelle



Rule Inversion in Isabelle

name of the new lemma

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% L48    (Isar Utoks Abbrev; Scripting )----------------
```

```
⟦⟨SKIP,?s⟩ ⇝ ?s'; ?s' = ?s ⟹ ?P⟧ ⟹ ?P
```

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

Rule Inversion in Isabelle

name of the new lemma

declared as an *elimination rule* to `auto` and `blast`

```
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% L48    (Isar Utoks Abbrev; Scripting )----------------
```

```
⟦⟨SKIP,?s⟩ ⇝ ?s'; ?s' = ?s ⟹ ?P⟧ ⟹ ?P
```

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

Rule Inversion in Isabelle

name of the new lemma

declared as an *elimination rule* to `auto` and `blast`

```
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'"
```

$\langle \mathbf{skip}, s \rangle \rightarrow s'$ implies $s = s'$

```
-u-:---  Com.thy          53% L48    (Isar Utoks Abbrev; Scripting )----------------
⟦⟨SKIP,?s⟩ ⇒ ?s'; ?s' = ?s ⟹ ?P⟧ ⟹ ?P

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

Rule Inversion in Isabelle

name of the new lemma

declared as an *elimination rule* to `auto` and `blast`

```
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'"
```

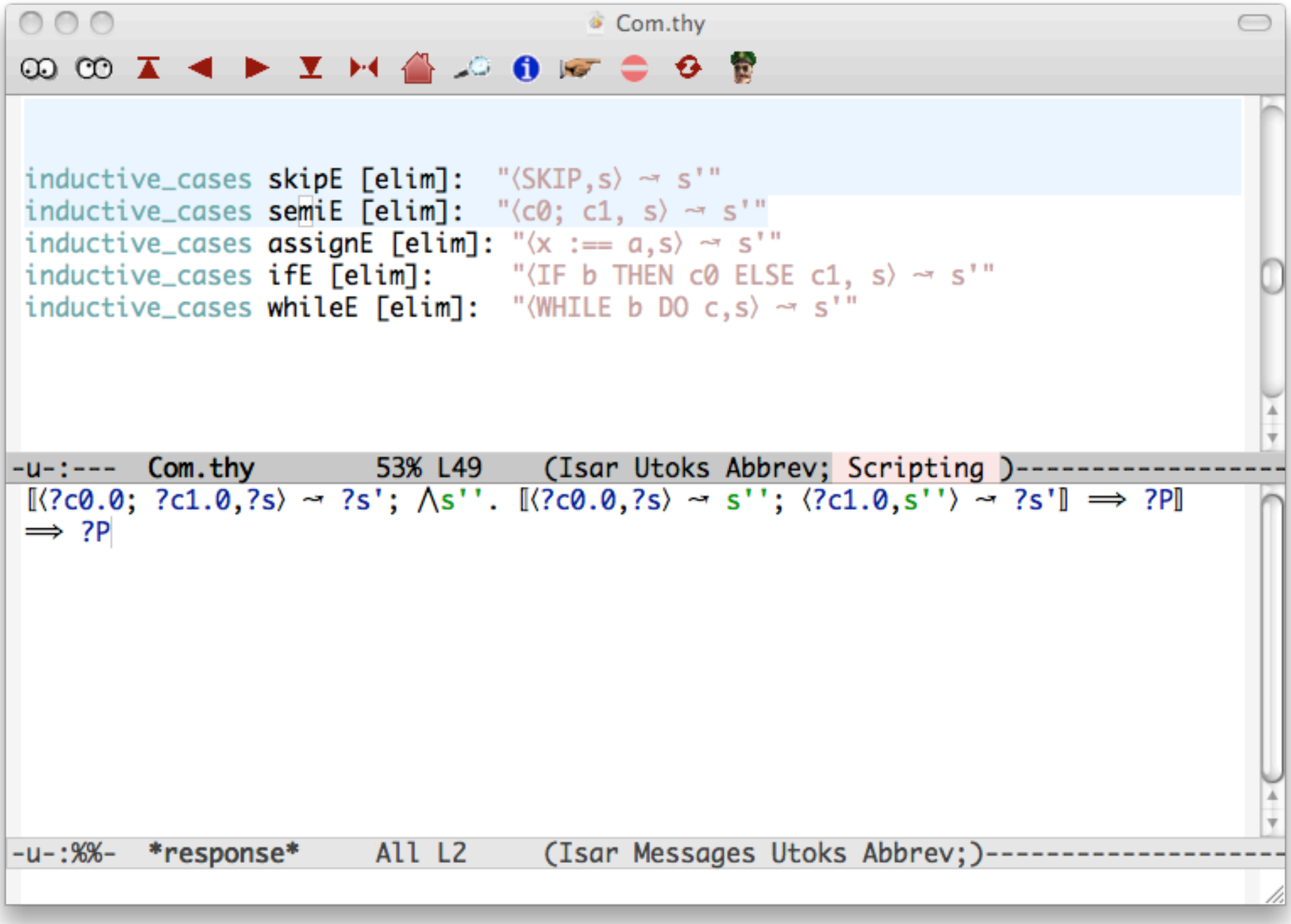
$\langle \mathbf{skip}, s \rangle \rightarrow s'$ implies $s = s'$

```
-u-:---  Com.thy        53% L48    (Isar Utoks Abbrev; Scripting )----------------
⟦⟨SKIP,?s⟩ ⇒ ?s'; ?s' = ?s ⟹ ?P⟧ ⟹ ?P
```

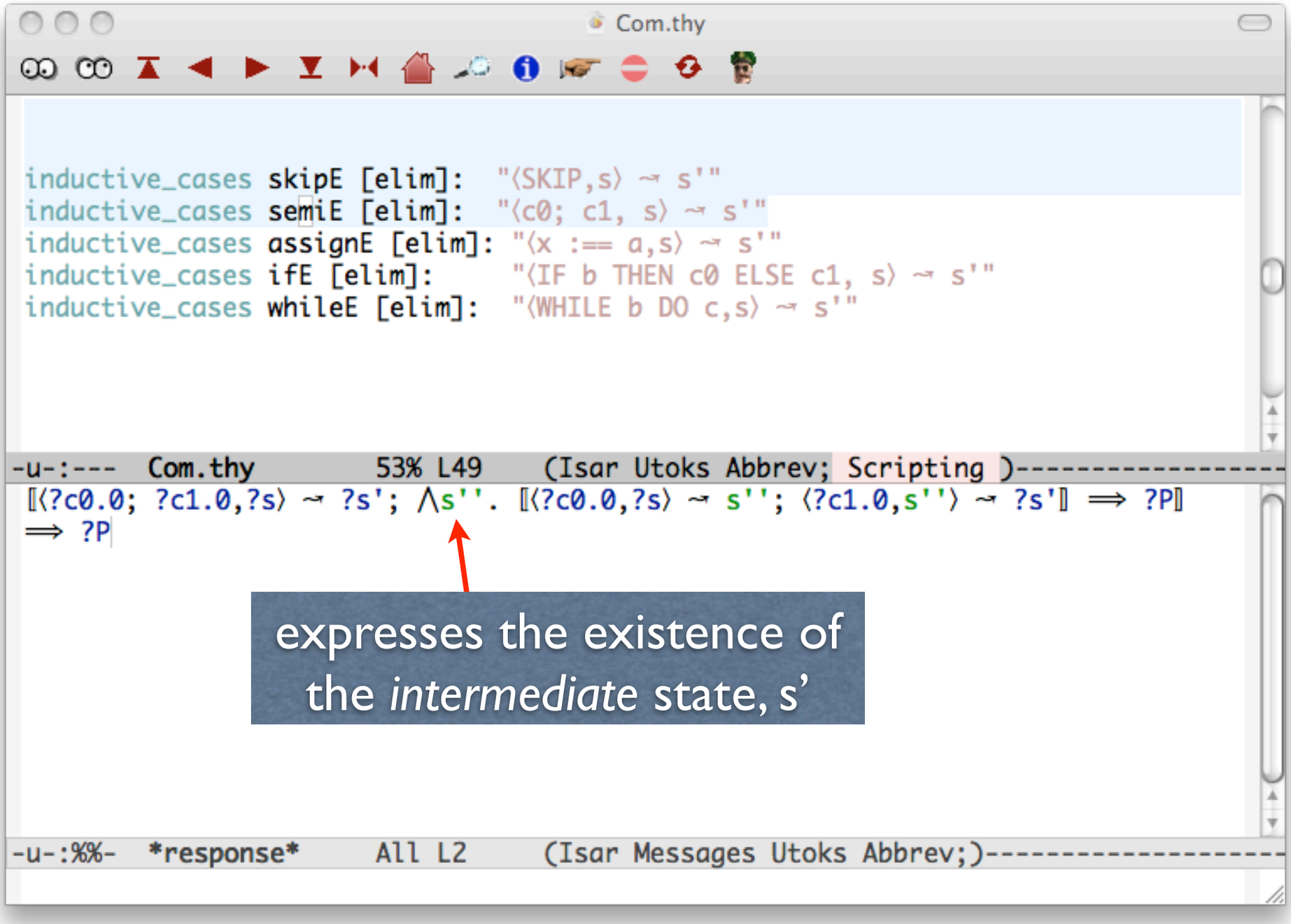
the typical format of an elimination rule

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

Rule Inversion Again



Rule Inversion Again



A Non-Termination Proof

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

The inductive version considers
all possible commands

A Non-Termination Proof

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

This formula is not provable by induction!

The inductive version considers
all possible commands

A Non-Termination Proof

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

This formula is not provable by induction!

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

The inductive version considers
all possible commands

A Non-Termination Proof

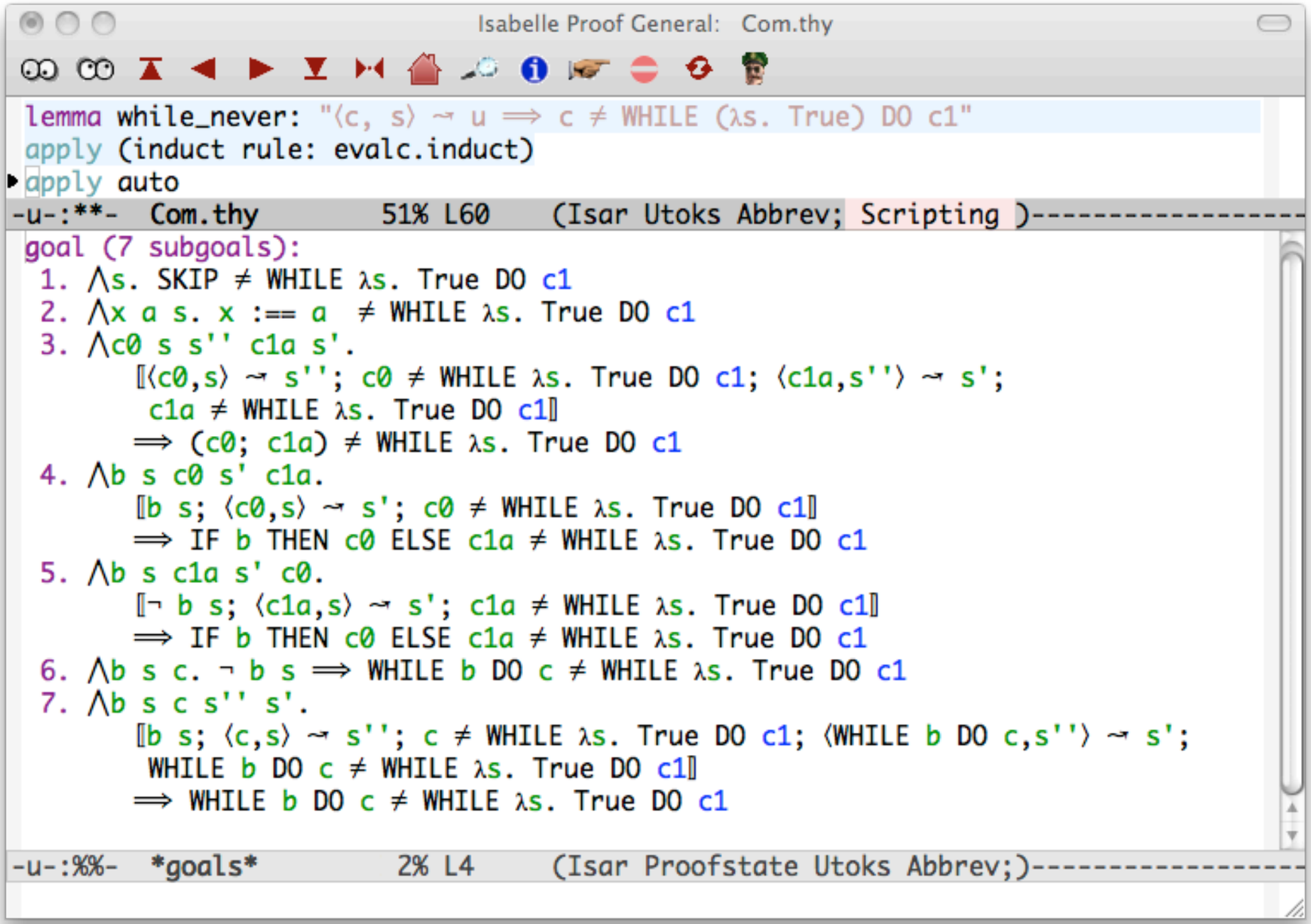
$$\langle \mathbf{while\ true\ do}\ c, s\rangle \not\to s'$$

This formula is not provable by induction!

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

The inductive version considers
all possible commands

Non-Termination in Isabelle



Non-Termination in Isabelle

7 subgoals, one for each rule of the definition

Isabelle Proof General: Com.thy

```
lemma while_never: "⟨c, s⟩ ⇒ u ⟹ c ≠ WHILE (λs. True) DO c1"
apply (induct rule: evalc.induct)
apply auto
-u-:**-  Com.thy        51% L60    (Isar Utoks Abbrev; Scripting )----------------
goal (7 subgoals):
 1. ⋀s. SKIP ≠ WHILE λs. True DO c1
 2. ⋀x a s. x :== a  ≠ WHILE λs. True DO c1
 3. ⋀c0 s s'' c1a s'.
       ⟦⟨c0,s⟩ ⇒ s''; c0 ≠ WHILE λs. True DO c1; ⟨c1a,s''⟩ ⇒ s';
        c1a ≠ WHILE λs. True DO c1⟧
       ⟹ (c0; c1a) ≠ WHILE λs. True DO c1
 4. ⋀b s c0 s' c1a.
       ⟦b s; ⟨c0,s⟩ ⇒ s'; c0 ≠ WHILE λs. True DO c1⟧
       ⟹ IF b THEN c0 ELSE c1a ≠ WHILE λs. True DO c1
 5. ⋀b s c1a s' c0.
       ⟦¬ b s; ⟨c1a,s⟩ ⇒ s'; c1a ≠ WHILE λs. True DO c1⟧
       ⟹ IF b THEN c0 ELSE c1a ≠ WHILE λs. True DO c1
 6. ⋀b s c. ¬ b s ⟹ WHILE b DO c ≠ WHILE λs. True DO c1
 7. ⋀b s c s'' s'.
       ⟦b s; ⟨c,s⟩ ⇒ s''; c ≠ WHILE λs. True DO c1; ⟨WHILE b DO c,s''⟩ ⇒ s';
        WHILE b DO c ≠ WHILE λs. True DO c1⟧
       ⟹ WHILE b DO c ≠ WHILE λs. True DO c1

-u-:%%-  *goals*         2% L4     (Isar Proofstate Utoks Abbrev;)----------------
```

Non-Termination in Isabelle

7 subgoals, one for each rule of the definition

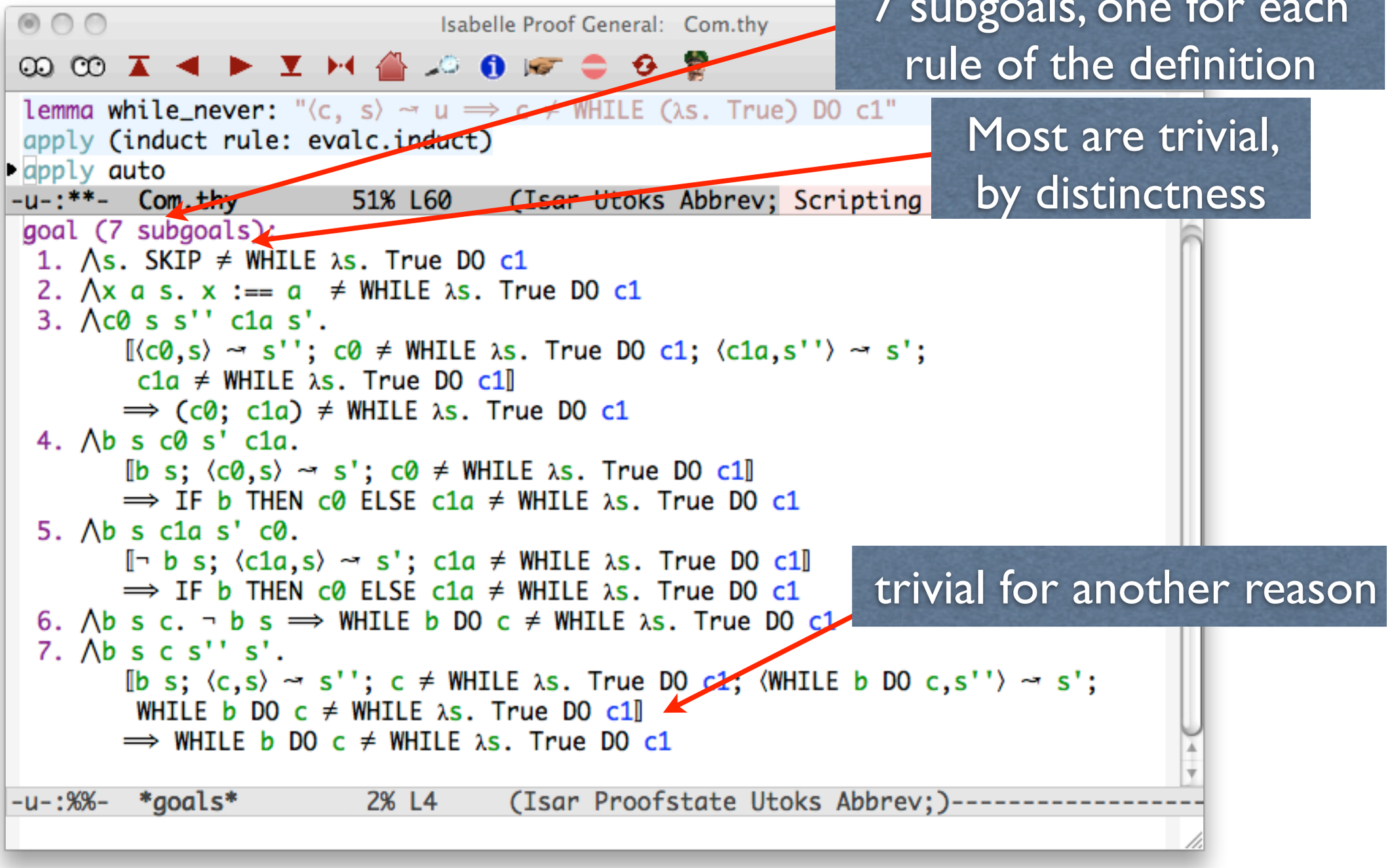
Most are trivial, by distinctness

Isabelle Proof General: Com.thy

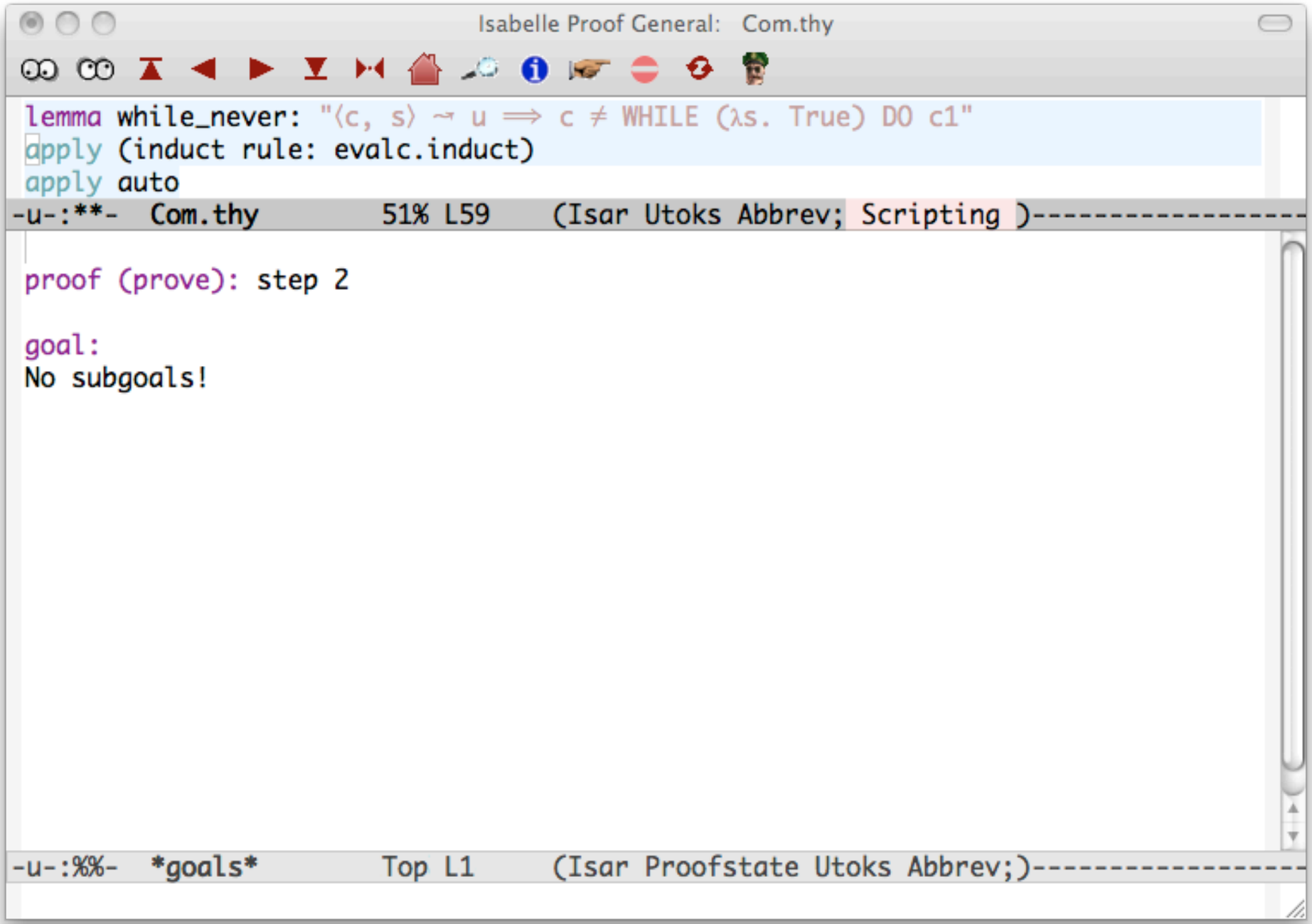
```
lemma while_never: "⟨c, s⟩ ⇒ u ⟹ c ≠ WHILE (λs. True) DO c1"
apply (induct rule: evalc.induct)
apply auto
-u-:**-  Com.thy        51% L60    (Isar Utoks Abbrev; Scripting
goal (7 subgoals):
 1. ⋀s. SKIP ≠ WHILE λs. True DO c1
 2. ⋀x a s. x :== a  ≠ WHILE λs. True DO c1
 3. ⋀c0 s s'' c1a s'.
       ⟦⟨c0,s⟩ ⇒ s''; c0 ≠ WHILE λs. True DO c1; ⟨c1a,s''⟩ ⇒ s';
        c1a ≠ WHILE λs. True DO c1⟧
       ⟹ (c0; c1a) ≠ WHILE λs. True DO c1
 4. ⋀b s c0 s' c1a.
       ⟦b s; ⟨c0,s⟩ ⇒ s'; c0 ≠ WHILE λs. True DO c1⟧
       ⟹ IF b THEN c0 ELSE c1a ≠ WHILE λs. True DO c1
 5. ⋀b s c1a s' c0.
       ⟦¬ b s; ⟨c1a,s⟩ ⇒ s'; c1a ≠ WHILE λs. True DO c1⟧
       ⟹ IF b THEN c0 ELSE c1a ≠ WHILE λs. True DO c1
 6. ⋀b s c. ¬ b s ⟹ WHILE b DO c ≠ WHILE λs. True DO c1
 7. ⋀b s c s'' s'.
       ⟦b s; ⟨c,s⟩ ⇒ s''; c ≠ WHILE λs. True DO c1; ⟨WHILE b DO c,s''⟩ ⇒ s';
        WHILE b DO c ≠ WHILE λs. True DO c1⟧
       ⟹ WHILE b DO c ≠ WHILE λs. True DO c1

-u-:%%-  *goals*         2% L4     (Isar Proofstate Utoks Abbrev;)----------
```

Non-Termination in Isabelle



Done!

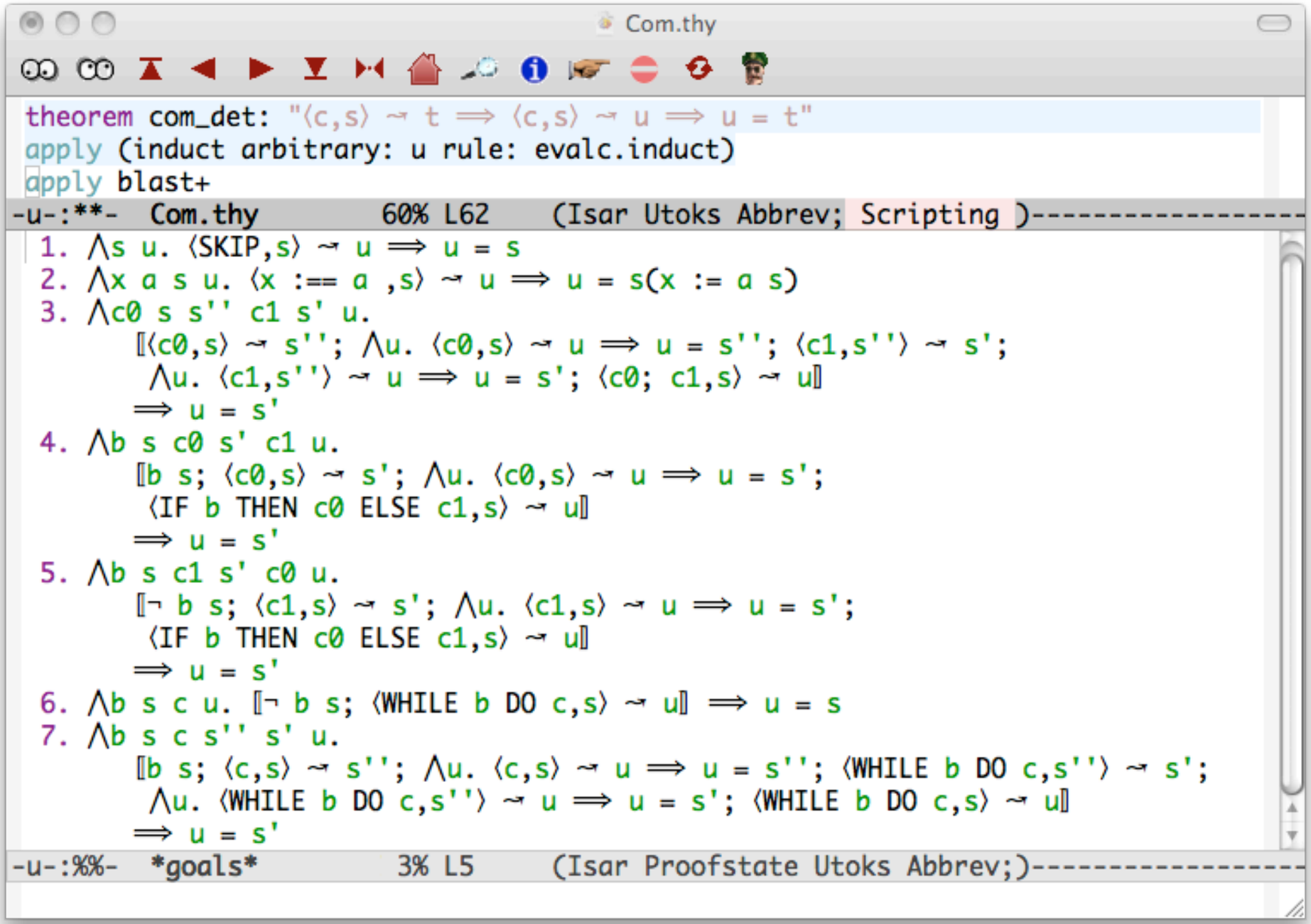

Isabelle Proof General: Com.thy
lemma while_never: "⟨c, s⟩ ⇝ u ⟹ c ≠ WHILE (λs. True) DO c1"
apply (induct rule: evalc.induct)
apply auto
-u-:**- Com.thy 51% L59 (Isar Utoks Abbrev; Scripting)
proof (prove): step 2
goal:
No subgoals!
-u-:%%- *goals* Top L1 (Isar Proofstate Utoks Abbrev;)

Determinacy

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

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

Determinacy in Isabelle



Determinacy in Isabelle

allow the other state to *vary*

Com.thy

```
theorem com_det: "⟨c,s⟩ ~ t ⟹ ⟨c,s⟩ ~ u ⟹ u = t"
apply (induct arbitrary: u rule: evalc.induct)
apply blast+
-u-:**-  Com.thy         60% L62    (Isar Utoks Abbrev; Scripting )----------
 1. ⋀s u. ⟨SKIP,s⟩ ~ u ⟹ u = s
 2. ⋀x a s u. ⟨x :== a ,s⟩ ~ u ⟹ u = s(x := a s)
 3. ⋀c0 s s'' c1 s' u.
       ⟦⟨c0,s⟩ ~ s''; ⋀u. ⟨c0,s⟩ ~ u ⟹ u = s''; ⟨c1,s''⟩ ~ s';
        ⋀u. ⟨c1,s''⟩ ~ u ⟹ u = s'; ⟨c0; c1,s⟩ ~ u⟧
       ⟹ u = s'
 4. ⋀b s c0 s' c1 u.
       ⟦b s; ⟨c0,s⟩ ~ s'; ⋀u. ⟨c0,s⟩ ~ u ⟹ u = s';
        ⟨IF b THEN c0 ELSE c1,s⟩ ~ u⟧
       ⟹ u = s'
 5. ⋀b s c1 s' c0 u.
       ⟦¬ b s; ⟨c1,s⟩ ~ s'; ⋀u. ⟨c1,s⟩ ~ u ⟹ u = s';
        ⟨IF b THEN c0 ELSE c1,s⟩ ~ u⟧
       ⟹ u = s'
 6. ⋀b s c u. ⟦¬ b s; ⟨WHILE b DO c,s⟩ ~ u⟧ ⟹ u = s
 7. ⋀b s c s'' s' u.
       ⟦b s; ⟨c,s⟩ ~ s''; ⋀u. ⟨c,s⟩ ~ u ⟹ u = s''; ⟨WHILE b DO c,s''⟩ ~ s';
        ⋀u. ⟨WHILE b DO c,s''⟩ ~ u ⟹ u = s'; ⟨WHILE b DO c,s⟩ ~ u⟧
       ⟹ u = s'
-u-:%%-  *goals*         3% L5     (Isar Proofstate Utoks Abbrev;)----------
```

Determinacy in Isabelle

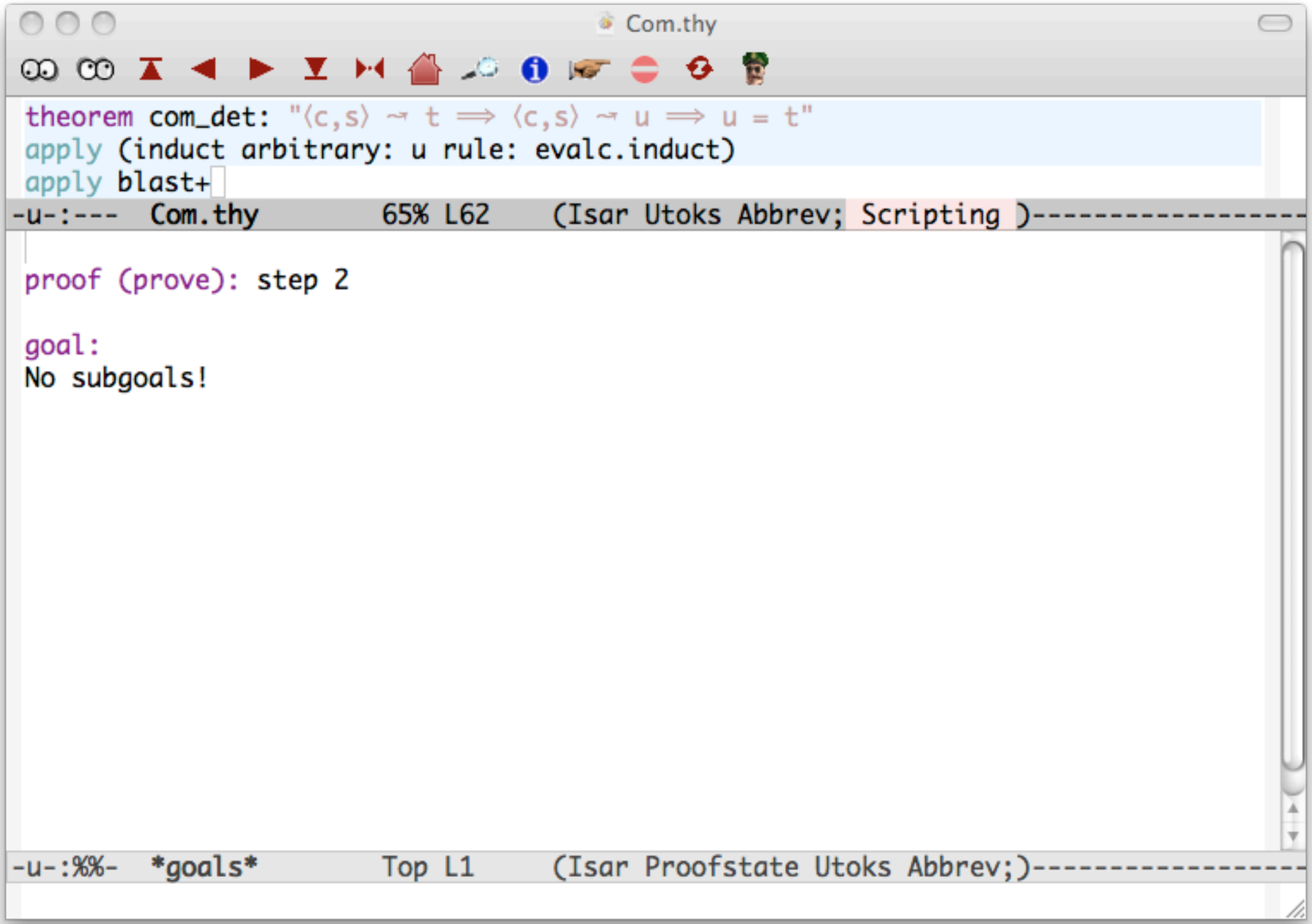
allow the other state to *vary*

trivial by rule inversion

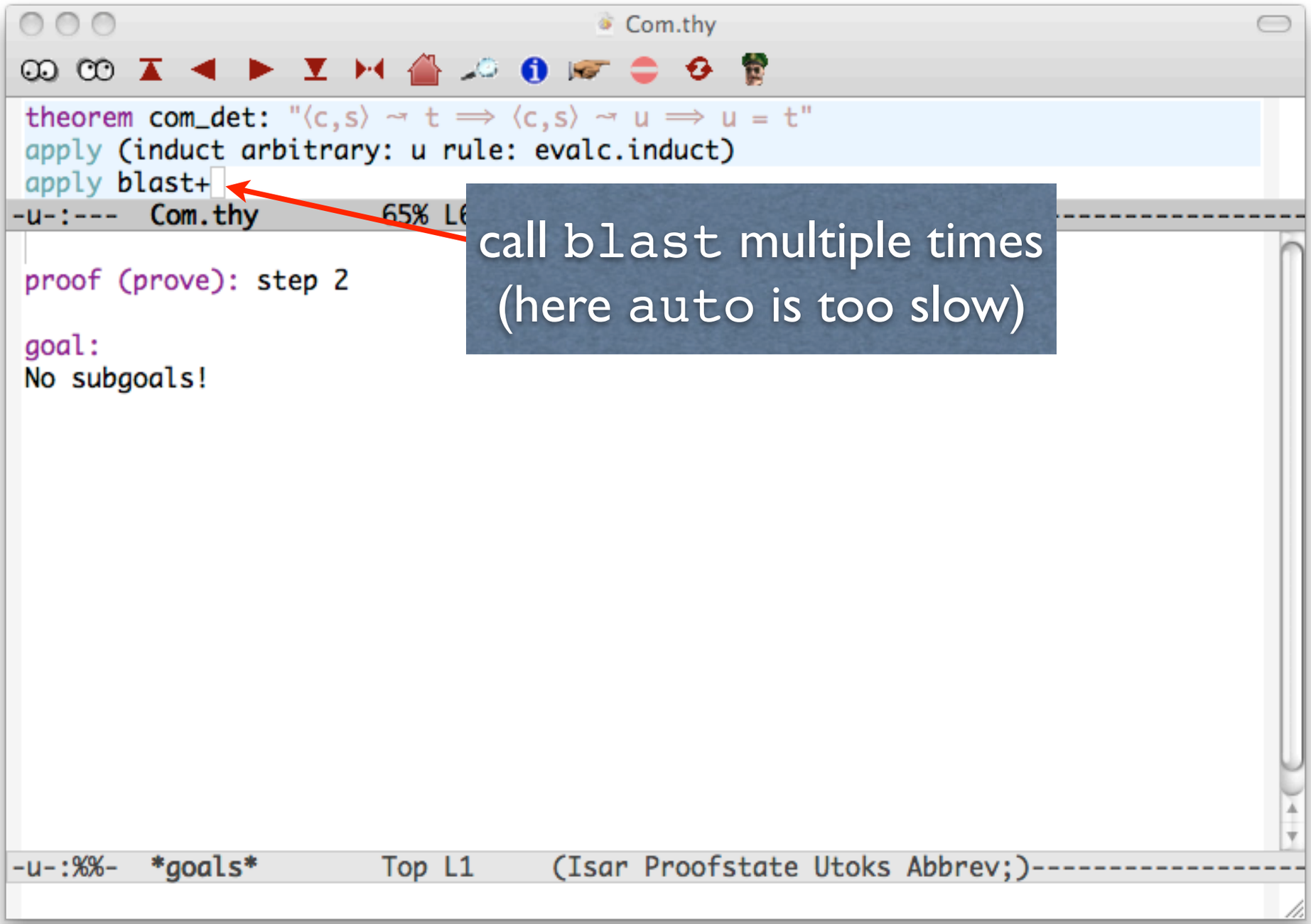
Com.thy

```
theorem com_det: "⟨c,s⟩ ~ t ⟹ ⟨c,s⟩ ~ u ⟹ u = t"
apply (induct arbitrary: u rule: evalc.induct)
apply blast+
-u-:**-  Com.thy         60% L62    (Isar Utoks Abbrev; Scripting )----------------
 1. ⋀s u. ⟨SKIP,s⟩ ~ u ⟹ u = s
 2. ⋀x a s u. ⟨x :== a ,s⟩ ~ u ⟹ u = s(x := a s)
 3. ⋀c0 s s'' c1 s' u.
       ⟦⟨c0,s⟩ ~ s''; ⋀u. ⟨c0,s⟩ ~ u ⟹ u = s''; ⟨c1,s''⟩ ~ s';
        ⋀u. ⟨c1,s''⟩ ~ u ⟹ u = s'; ⟨c0; c1,s⟩ ~ u⟧
       ⟹ u = s'
 4. ⋀b s c0 s' c1 u.
       ⟦b s; ⟨c0,s⟩ ~ s'; ⋀u. ⟨c0,s⟩ ~ u ⟹ u = s';
        ⟨IF b THEN c0 ELSE c1,s⟩ ~ u⟧
       ⟹ u = s'
 5. ⋀b s c1 s' c0 u.
       ⟦¬ b s; ⟨c1,s⟩ ~ s'; ⋀u. ⟨c1,s⟩ ~ u ⟹ u = s';
        ⟨IF b THEN c0 ELSE c1,s⟩ ~ u⟧
       ⟹ u = s'
 6. ⋀b s c u. ⟦¬ b s; ⟨WHILE b DO c,s⟩ ~ u⟧ ⟹ u = s
 7. ⋀b s c s'' s' u.
       ⟦b s; ⟨c,s⟩ ~ s''; ⋀u. ⟨c,s⟩ ~ u ⟹ u = s''; ⟨WHILE b DO c,s''⟩ ~ s';
        ⋀u. ⟨WHILE b DO c,s''⟩ ~ u ⟹ u = s'; ⟨WHILE b DO c,s⟩ ~ u⟧
       ⟹ u = s'
-u-:%%-  *goals*         3% L5     (Isar Proofstate Utoks Abbrev;)----------------
```

Proved by Rule Inversion



Proved by Rule Inversion



Semantic Equivalence

Isabelle Proof General: Com.thy

```
subsection {*Equivalence of commands*}

text{*Two commands are equivalent if they allow the same transitions.*}

definition
  equiv_c :: "com ⇒ com ⇒ bool" ("_ ~ _")
where
  "(c ~ c') = (∀s s'. (⟨c, s⟩ ⇒ s') = (⟨c', s⟩ ⇒ s'))"
```

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

Semantic Equivalence

Isabelle Proof General: Com.thy

```
subsection {*Equivalence of commands*}

text{*Two commands are equivalent if they allow the same transitions.*}

definition
  equiv_c :: "com ⇒ com ⇒ bool" ("_ ~ _")
where
  "(c ~ c') = (∀s s'. (⟨c, s⟩ ⇒ s') = (⟨c', s⟩ ⇒ s'))"
```

-u-:--- Com.thy 57% L77 (Isar Utoks Abbrev; Scripting)----------

Wrote /Users/lp15/Dropbox/ACS/8 - Operational Semantics/Com.thy

We can even define the infix syntax

Semantic Equivalence

Isabelle Proof General: Com.thy

```
subsection {*Equivalence of commands*}

text{*Two commands are equivalent if they allow the same transitions.*}

definition
  equiv_c :: "com ⇒ com ⇒ bool" ("_ ~ _")
where
  "(c ~ c') = (∀s s'. (⟨c, s⟩ ~> s') = (⟨c', s⟩ ~> s'))"
```

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

We can even define the infix syntax

It is trivially shown to be an *equivalence relation*

Semantic Equivalence

Isabelle Proof General: Com.thy

```
subsection {*Equivalence of commands*}

text{*Two commands are equivalent if they allow the same transitions.*}

definition
  equiv_c :: "com ⇒ com ⇒ bool" ("_ ~ _")
where
  "(c ~ c') = (∀s s'. (⟨c, s⟩ ⇒ s') = (⟨c', s⟩ ⇒ s'))"


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)
```

-u-:--- Com.thy 57% L77 (Isar Utoks Abbrev; Scripting)----------

Wrote /Users/lp15/Dropbox/ACS/8 - Operational Semantics/Com.thy

We can even define the infix syntax

It is trivially shown to be an *equivalence relation*

More Semantic Equivalence!

Isabelle Proof General: Com.thy

```
lemma equiv_semi:
  "c1 ~ c1' ⟹ c2 ~ c2' ⟹ (c1; c2) ~ (c1'; c2')"
by (force simp add: equiv_c_def)

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

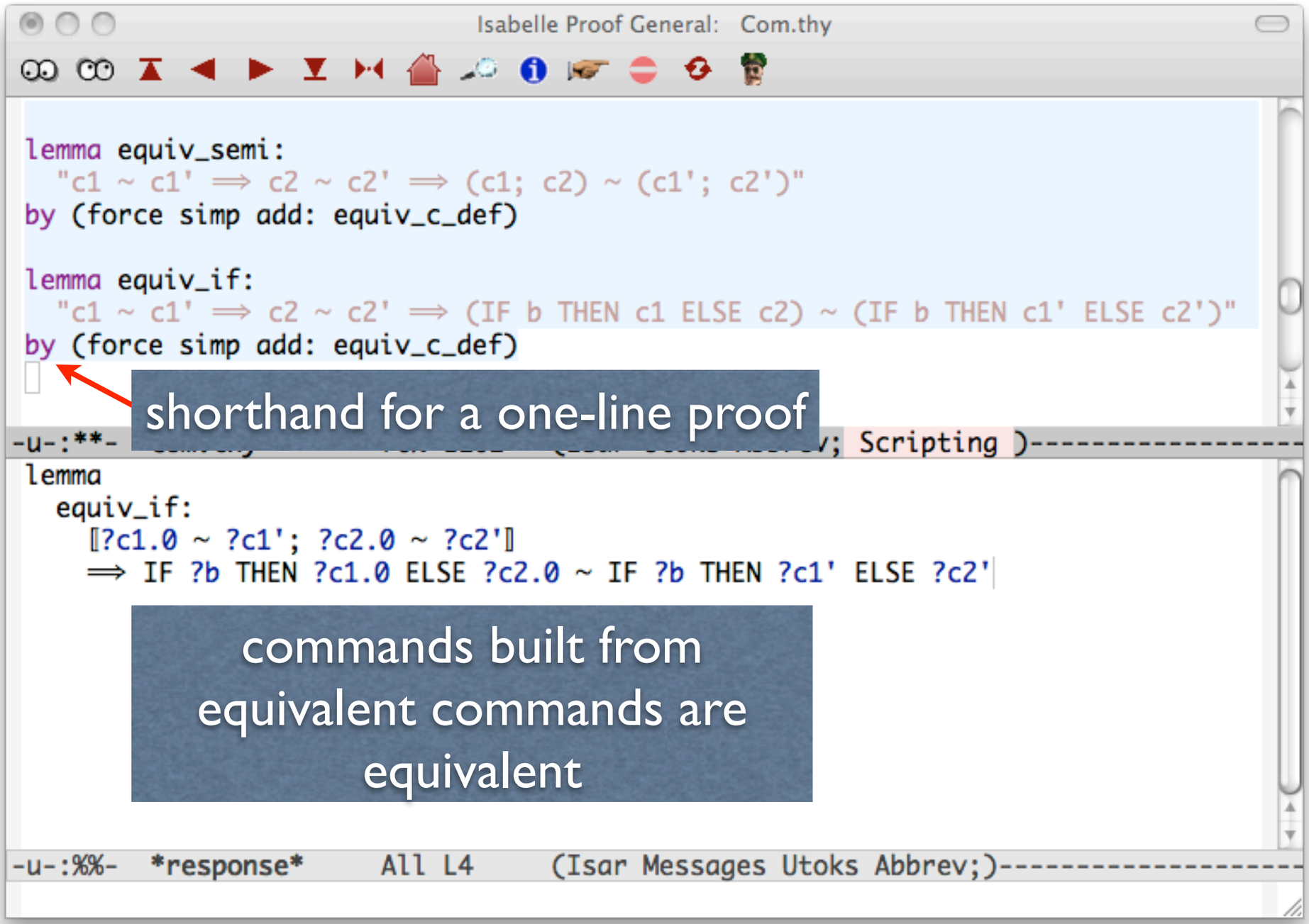
-u-:**- Com.thy 70% L102 (Isar Utoks Abbrev; Scripting)----------

```
lemma
  equiv_if:
    ⟦?c1.0 ~ ?c1'; ?c2.0 ~ ?c2'⟧
    ⟹ IF ?b THEN ?c1.0 ELSE ?c2.0 ~ IF ?b THEN ?c1' ELSE ?c2'
```

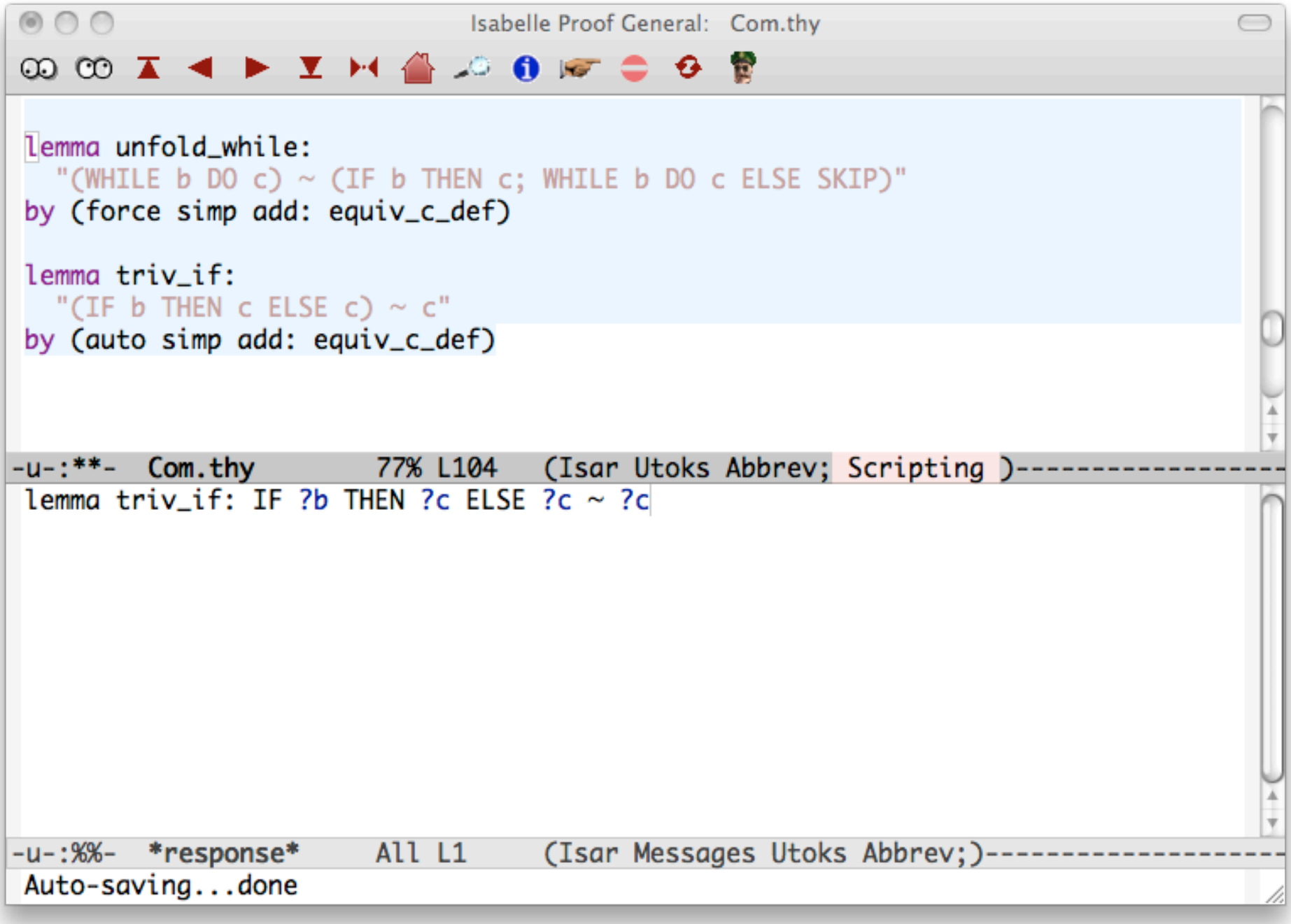
commands built from equivalent commands are equivalent

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

More Semantic Equivalence!

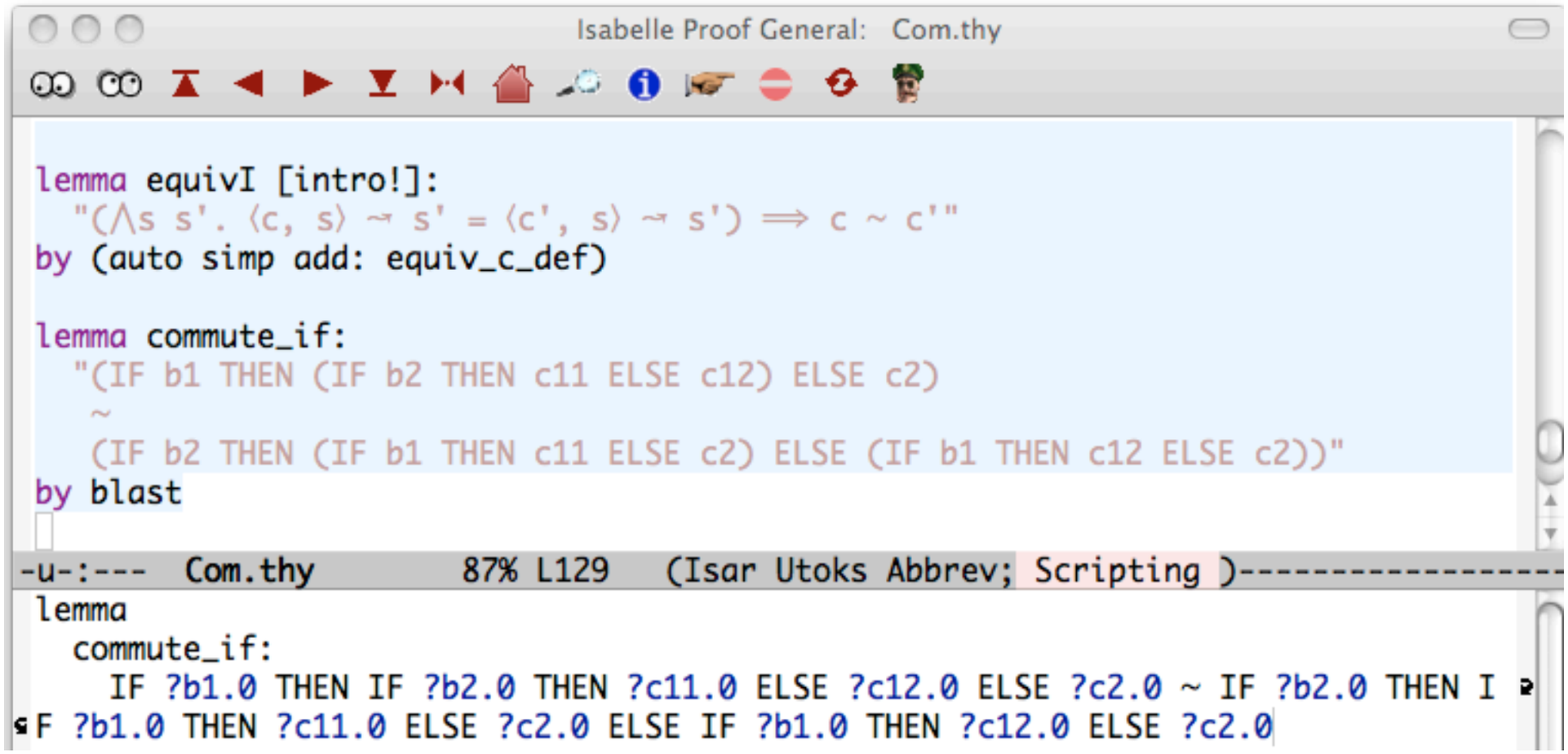


And More!!


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

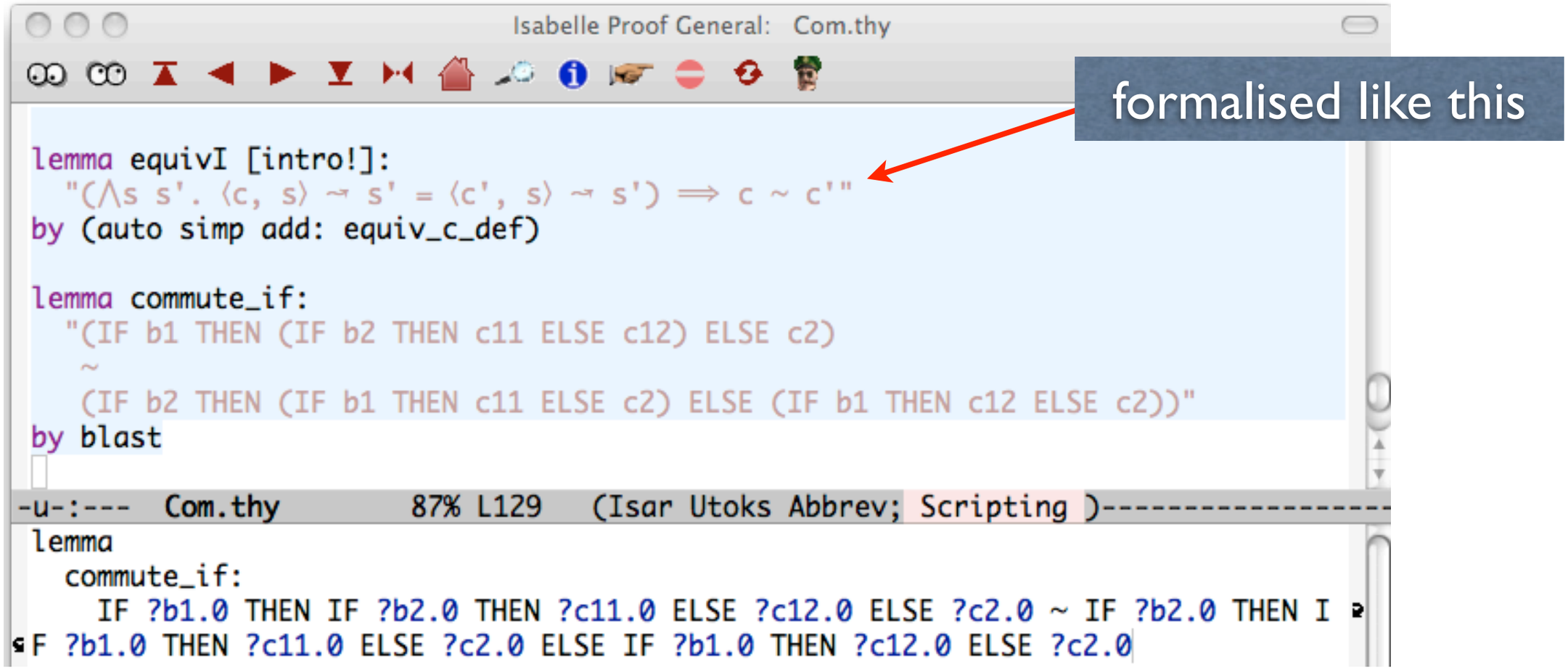
A New Introduction Rule

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



A New Introduction Rule

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



A New Introduction Rule

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

declared like this

formalised like this

Isabelle Proof General: Com.thy

```
lemma equivI [intro!]:
  "(⋀s s'. ⟨c, s⟩ ↝ s' = ⟨c', s⟩ ↝ s') ⟹ c ~ 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
```

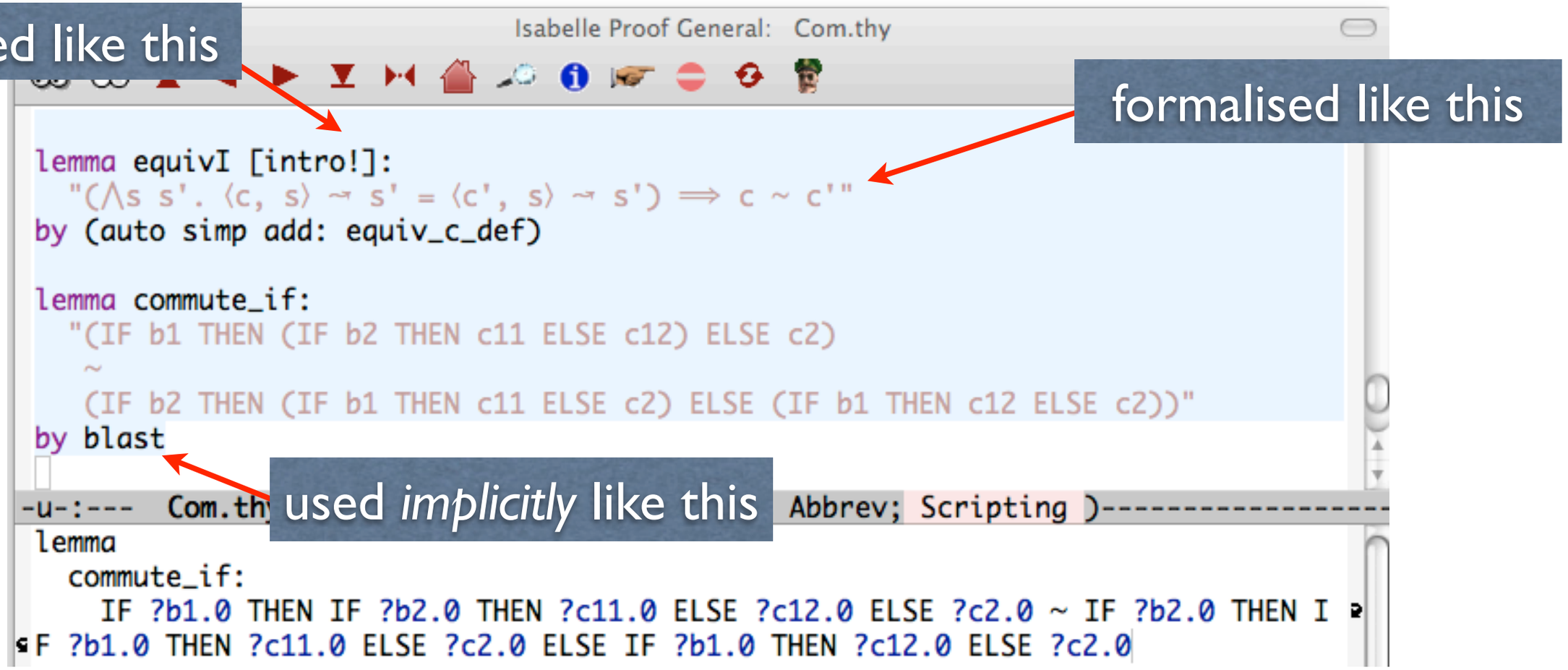
```
-u-:---  Com.thy        87% L129   (Isar Utoks Abbrev; Scripting )----------------
lemma
  commute_if:
    IF ?b1.0 THEN IF ?b2.0 THEN ?c11.0 ELSE ?c12.0 ELSE ?c2.0 ~ IF ?b2.0 THEN I ↵
F ?b1.0 THEN ?c11.0 ELSE ?c2.0 ELSE IF ?b1.0 THEN ?c12.0 ELSE ?c2.0
```

A New Introduction Rule

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

declared like this

formalised like this



Final Remarks on Semantics

Final Remarks on Semantics

- Small-step semantics is treated similarly.

Final Remarks on Semantics

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

Final Remarks on Semantics

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