Interactive Formal Verification
12: Modelling Hardware

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

Basic Principles of Modelling

Basic Principles of Modelling

- Define *mathematical abstractions* of the objects of interest (systems, hardware, protocols,...).

Basic Principles of Modelling

- Define *mathematical abstractions* of the objects of interest (systems, hardware, protocols,...).
- Whenever possible, use *definitions* — not axioms!

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.

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*!

Hardware Verification

Hardware Verification

- Pioneered by M. J. C. Gordon and his students, using successive versions of the HOL system.

Hardware Verification

- Pioneered by M. J. C. Gordon and his students, using successive versions of the HOL system.
- Used to model substantial hardware designs, including the ARM6 processor.

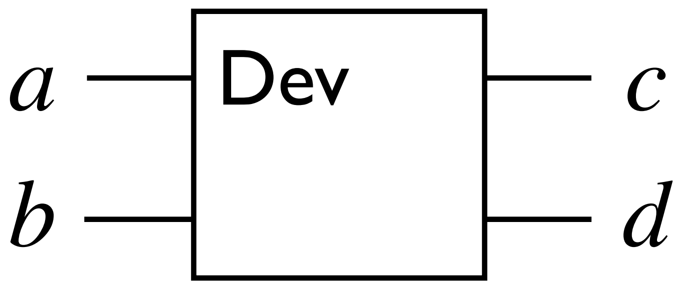
Hardware Verification

- Pioneered by M. J. C. Gordon and his students, using successive versions of the HOL system.
- Used to model substantial hardware designs, including the ARM6 processor.
- Works *hierarchically* from arithmetic units and memories right down to flip-flops and transistors.

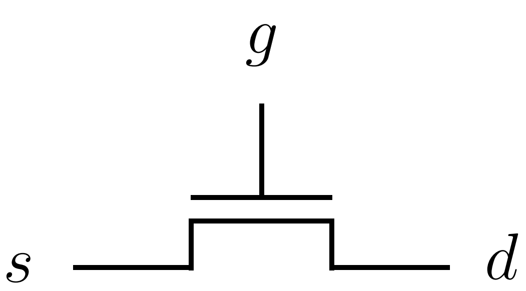
Hardware Verification

- Pioneered by M. J. C. Gordon and his students, using successive versions of the HOL system.
- Used to model substantial hardware designs, including the ARM6 processor.
- Works *hierarchically* from arithmetic units and memories right down to flip-flops and transistors.
- Crucially uses *higher-order logic*, modelling signals as boolean-valued functions over time.

Devices as Relations



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



$$g \to s = d$$

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

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

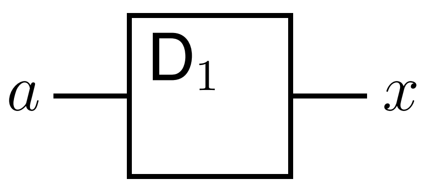
Relational Composition

Relational Composition

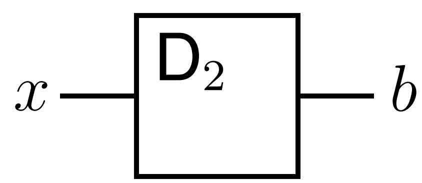
$\mathrm{S}_1[a,x]$ $\mathrm{S}_2[x,b]$

two devices modelled by two formulas

Relational Composition

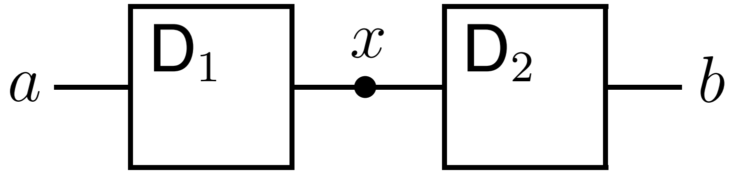


$\mathrm{S}_1[a, x]$



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

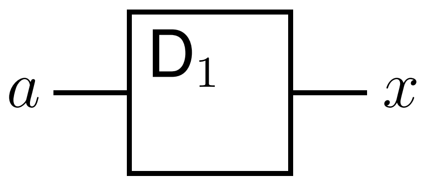
two devices modelled by two formulas



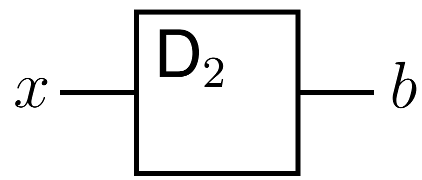
$\mathrm{S}_1[a, x] \wedge \mathrm{S}_2[x, b]$

the connected ports have the *same* value

Relational Composition

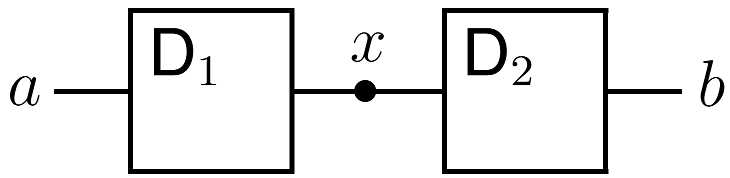


$S_1[a, x]$



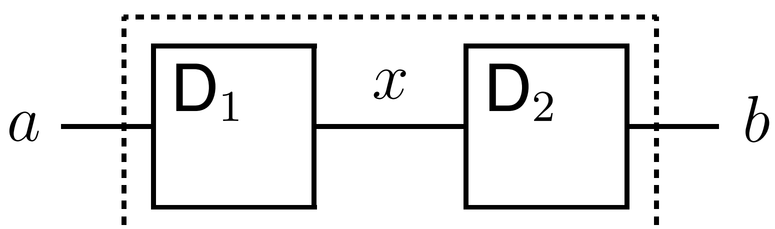
$S_2[x, b]$

two devices modelled by two formulas



$S_1[a, x] \wedge S_2[x, b]$

the connected ports have the *same* value



$\exists x.\, S_1[a, x] \wedge S_2[x, b]$

the connected ports have *some* value

Specifications and Correctness

Specifications and Correctness

- The *implementation* of a device in terms of other devices can be expressed by composition.

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.

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*: $Imp \Leftrightarrow Spec$.

Specifications and Correctness

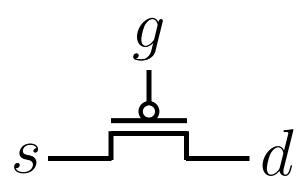
- The *implementation* of a device in terms of other devices can be expressed by composition.
- The *specification* of the device's intended behaviour can be given by an abstract formula.
- Sometimes the implementation and specification can be proved *equivalent*: $Imp \Leftrightarrow Spec$.
- The property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

Specifications and Correctness

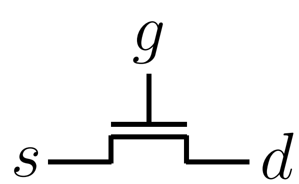
- The *implementation* of a device in terms of other devices can be expressed by composition.
- The *specification* of the device's intended behaviour can be given by an abstract formula.
- Sometimes the implementation and specification can be proved *equivalent*: $Imp \Leftrightarrow Spec$.
- The property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

  Impossible implementations satisfy *all* specifications!

The *Switch Model* of CMOS



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

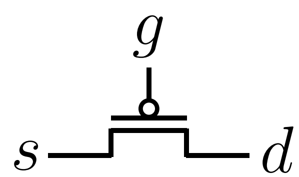


Ntran$(g, s, d) = (g \Rightarrow (d = s))$

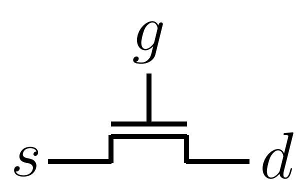
Gnd $g = (g = \mathsf{F})$

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

The *Switch Model* of CMOS



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



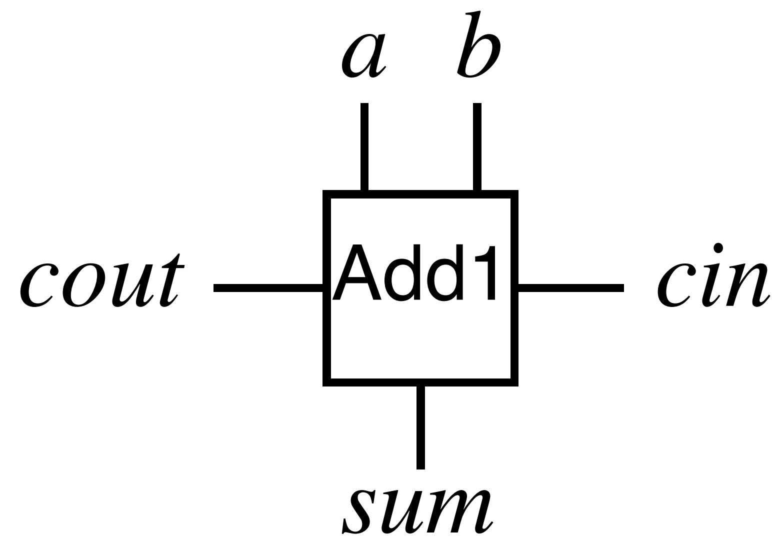
$\text{Ntran}(g, s, d) = (g \Rightarrow (d = s))$

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

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

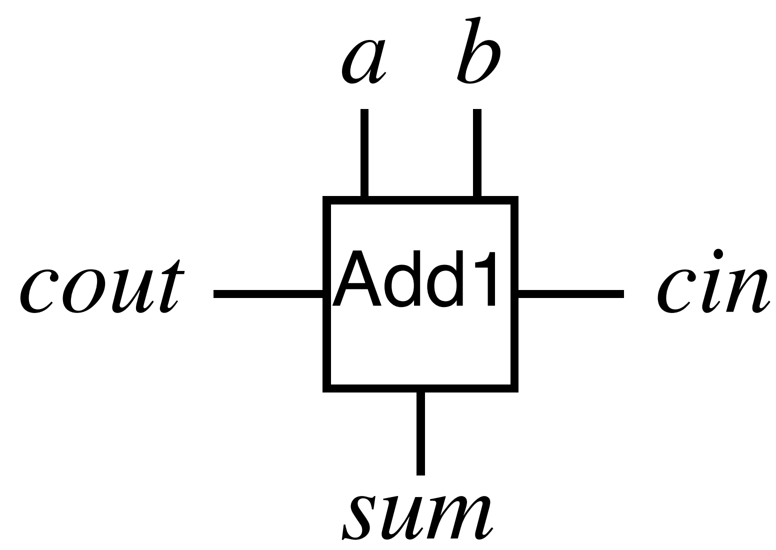
```
subsection{* Specification of CMOS primitives *}

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

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

Full Adder: Specification



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

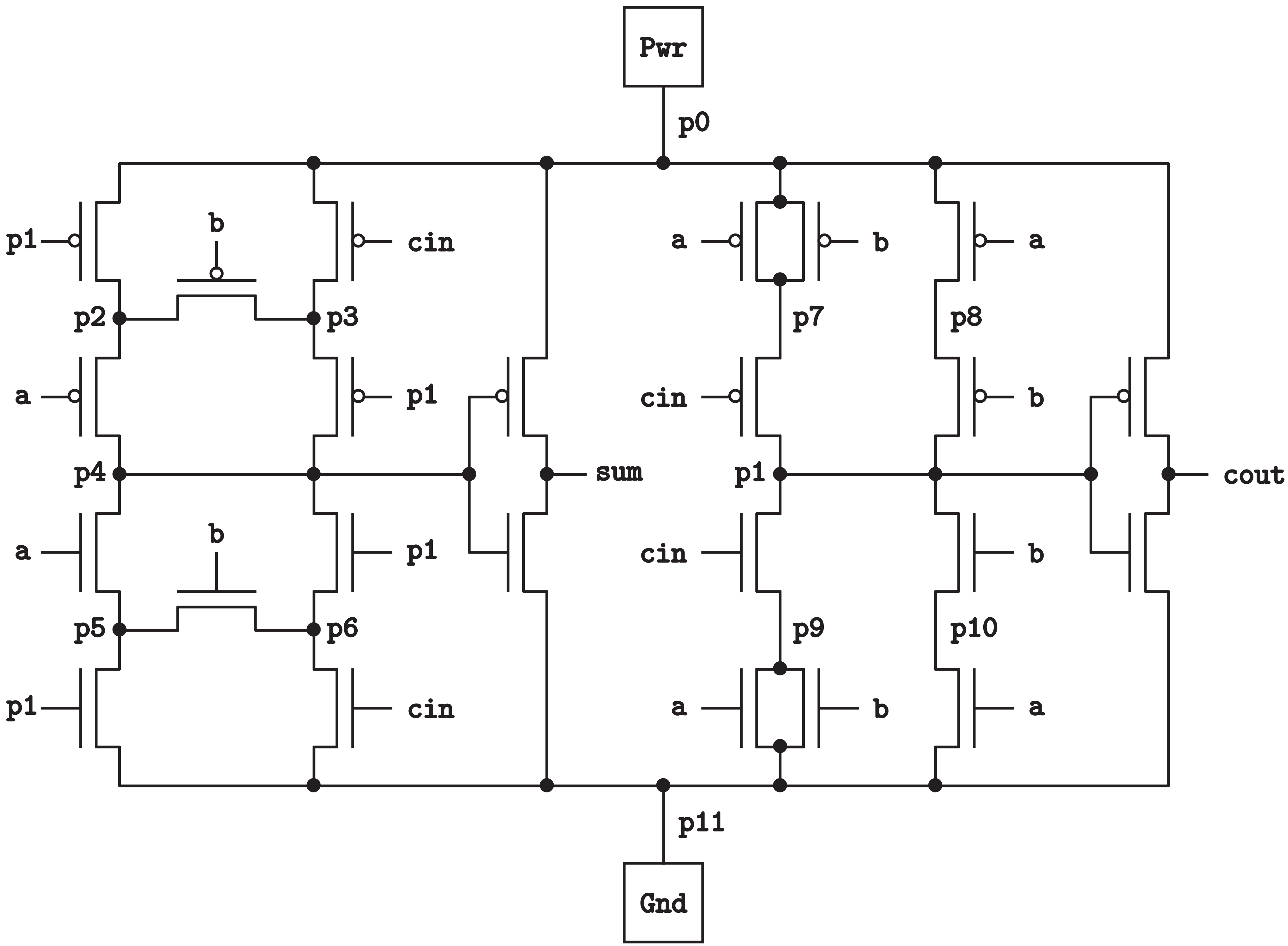
Full Adder: Specification



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

```
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)"
```

Full Adder: Implementation



Full Adder in Isabelle

Adder.thy

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

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

text{* Verification of CMOS full adder *}
lemma Add1Correct:
   "Add1Imp(a,b,cin,sum,cout) = Add1Spec(a,b,cin,sum,cout)"
by (simp add: Pwr_def Gnd_def Ntran_def Ptran_def Add1Spec_def
              Add1Imp_def bit_val_def ex_bool_eq)
-u-:**-  Adder.thy      27% L53     (Isar Utoks Abbrev; Scripting )----------------
```

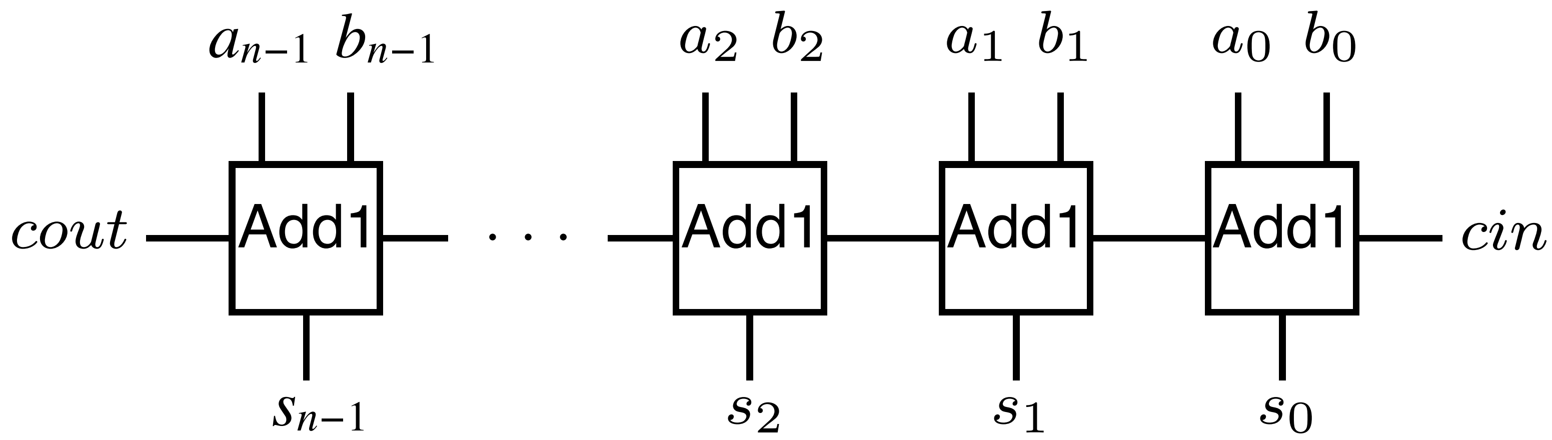
Full Adder in Isabelle

Adder.thy

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

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

text{* Verification of CMOS full adder *}
lemma Add1Correct:
   "Add1Imp(a,b,cin,sum,cout) = Add1Spec(a,b,cin,sum,cout)"
by (simp add: Pwr_def Gnd_def Ntran_def Ptran_def Add1Spec_def
              Add1Imp_def bit_val_def ex_bool_eq)
-u-:**-  Adder.thy      27% L53    (Isar Utoks Abbrev; Scripting )----------------
```

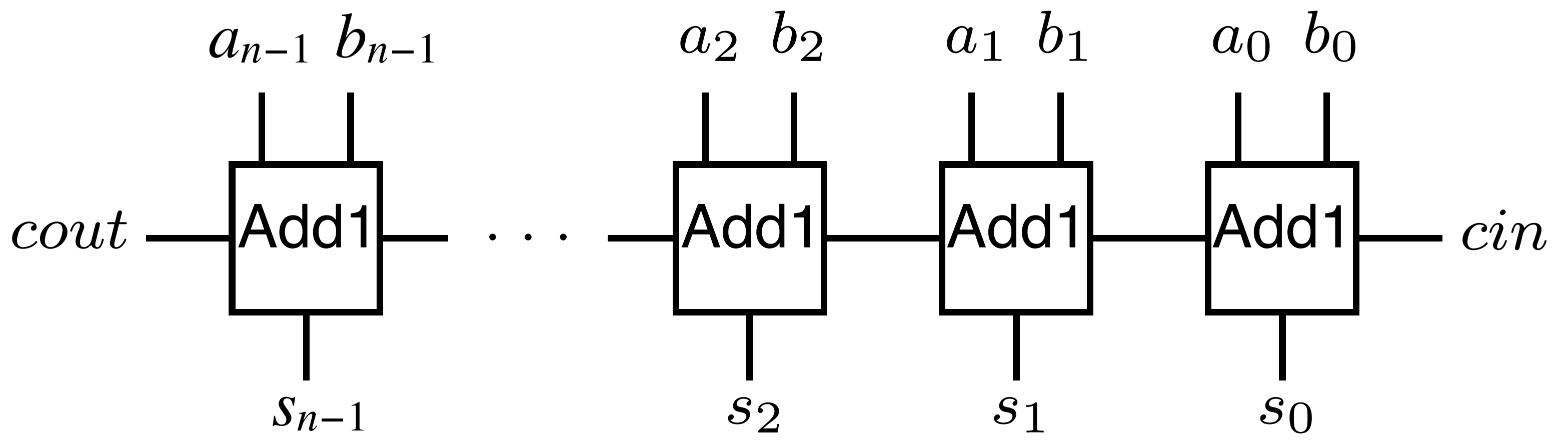
$(\exists b.\ P\ b) = (P\ \mathrm{True} \vee P\ \mathrm{False})$

An *n*-bit Ripple-Carry Adder



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

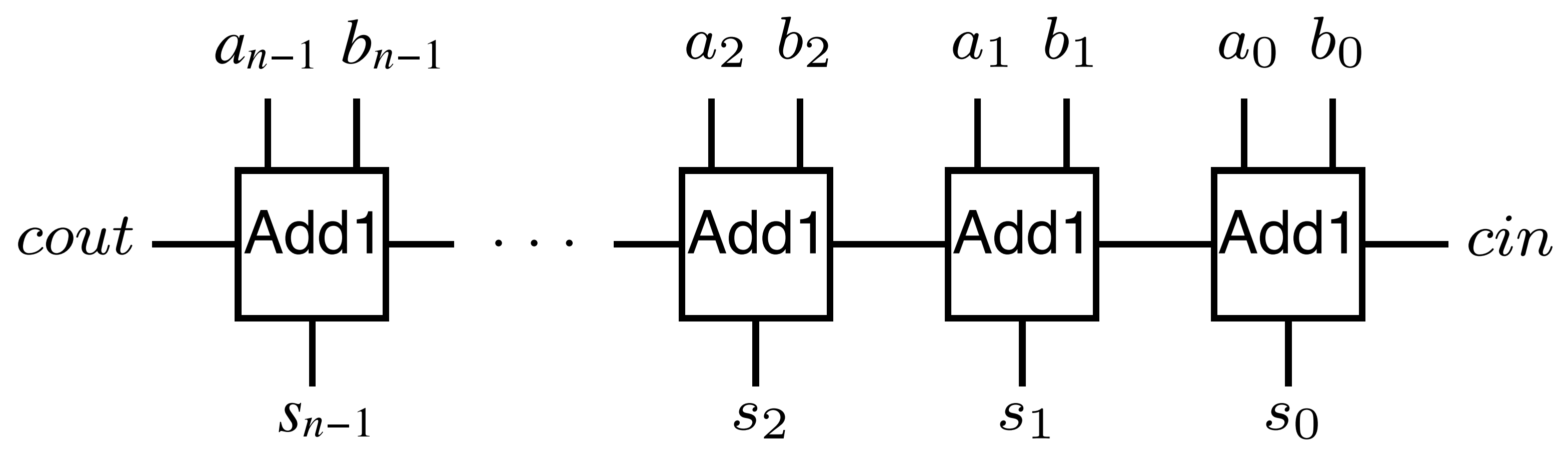
An *n*-bit Ripple-Carry Adder



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

- Cascading several full adders yields an *n*-bit adder.

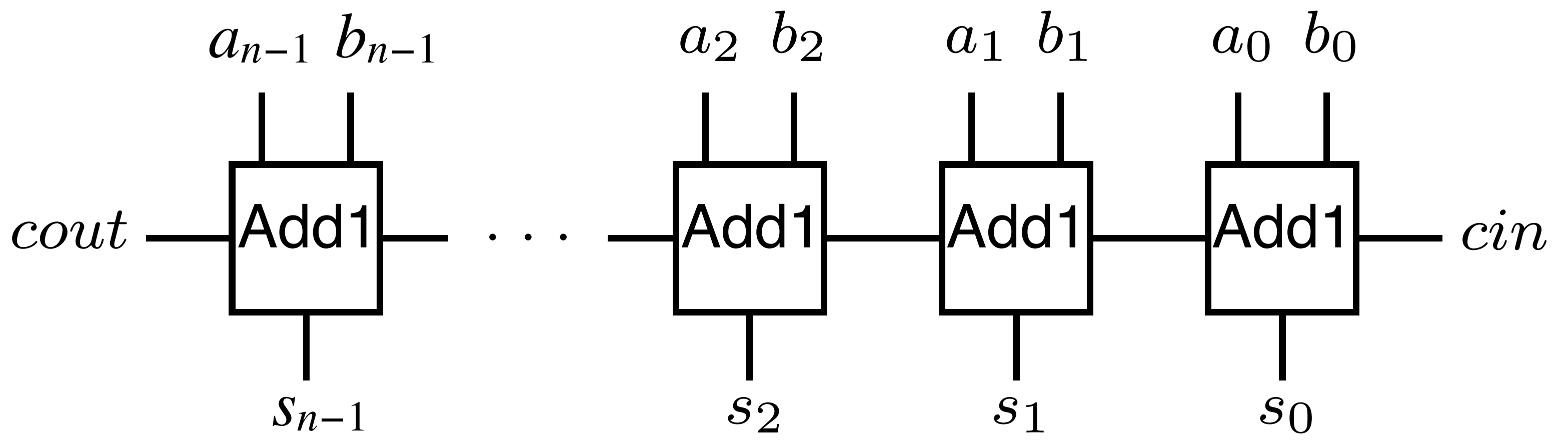
An *n*-bit Ripple-Carry Adder



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

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

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

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

Adder Specification

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

values of n-bit words

Adder Specification

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

values of n-bit words

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

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

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

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

Adder Specification

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

values of n-bit words

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

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

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

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

Adder Specification

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

values of n-bit words

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

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

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

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

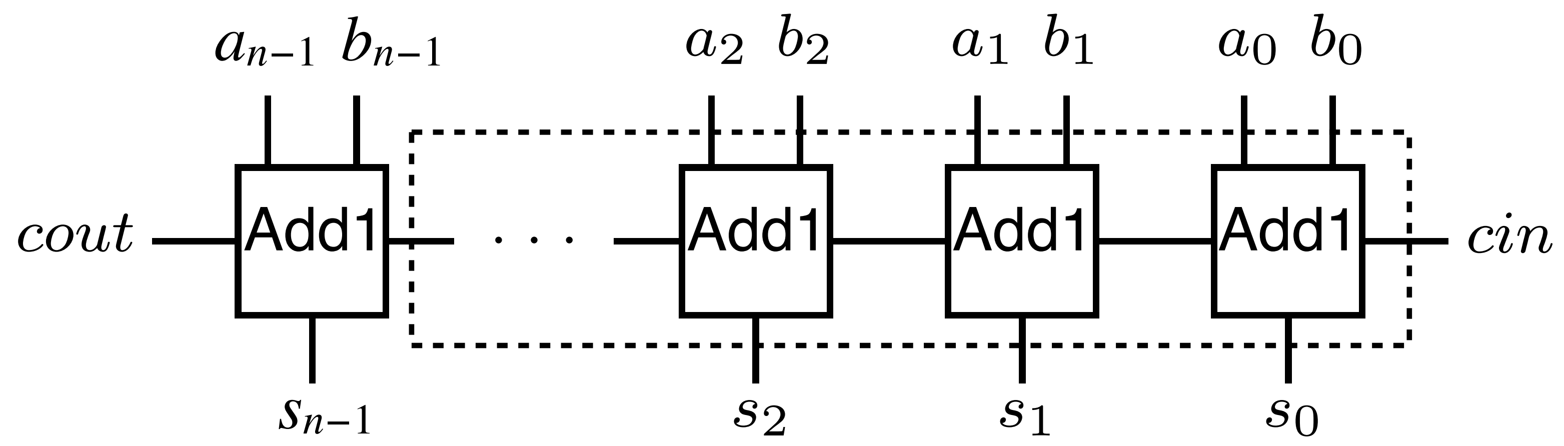
Adder Specification

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

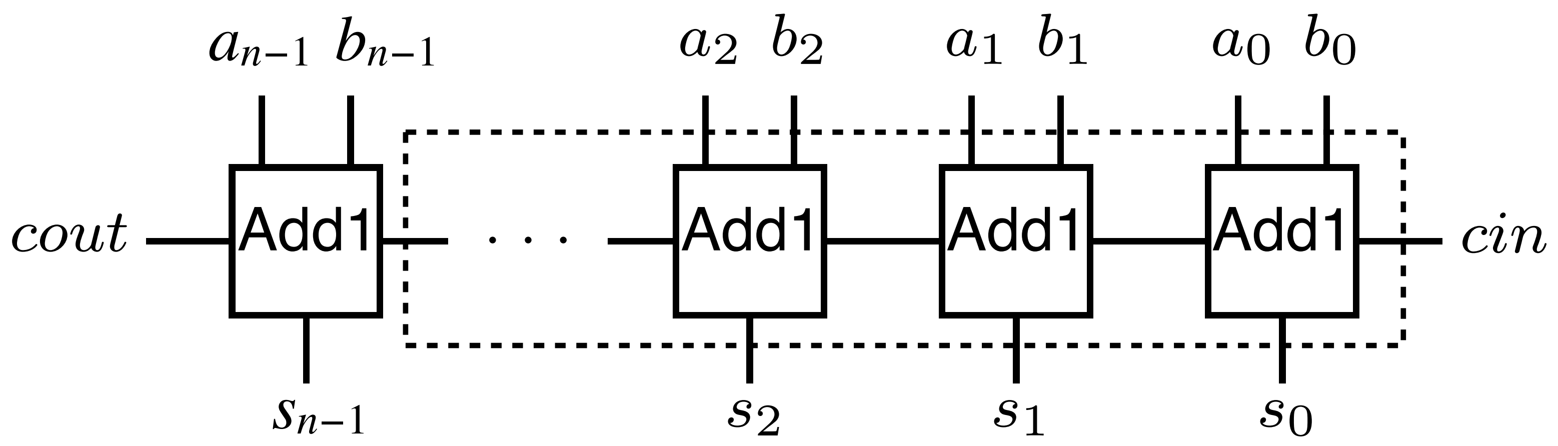
values of n-bit words

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

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

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

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

Adder Implementation

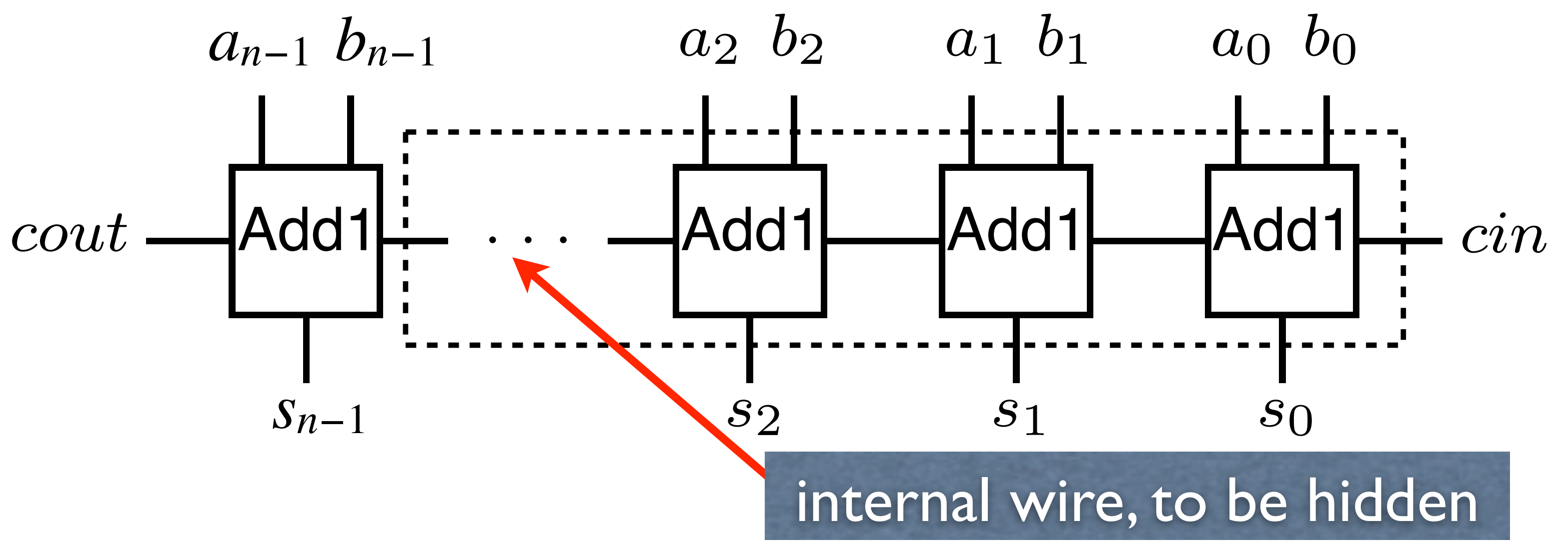


Adder Implementation



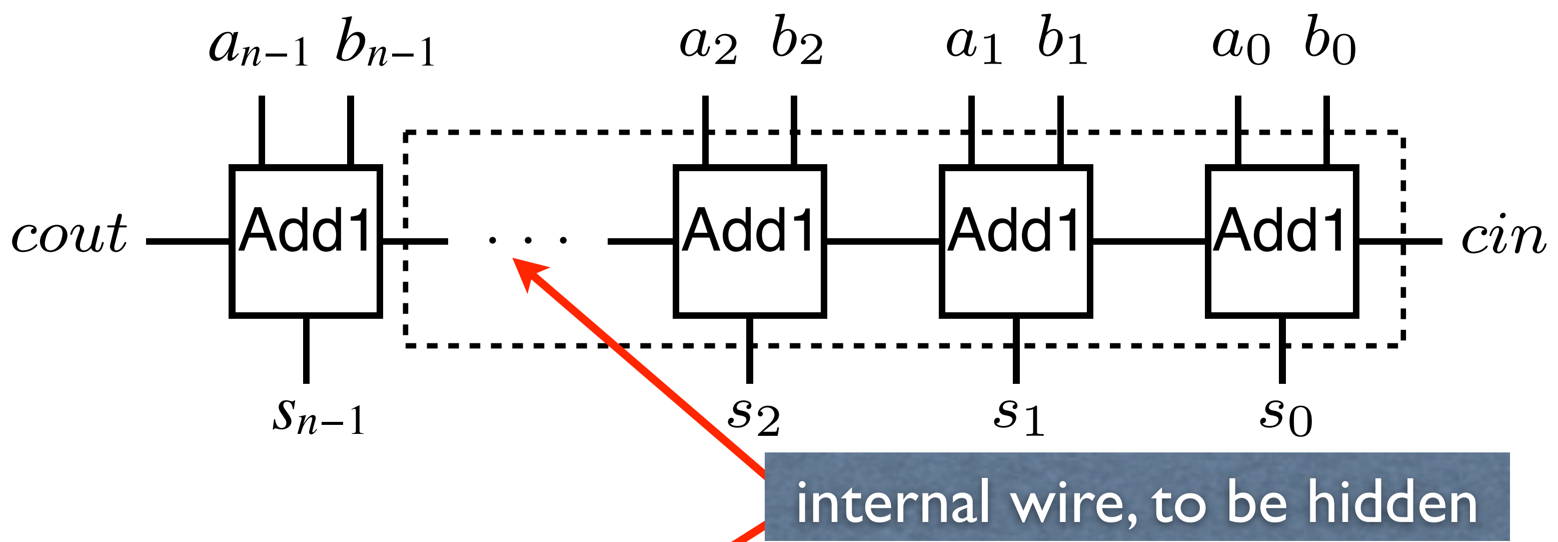

```
text{* Implementation of an n-bit ripple-carry adder*}

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))"
```

Adder Implementation



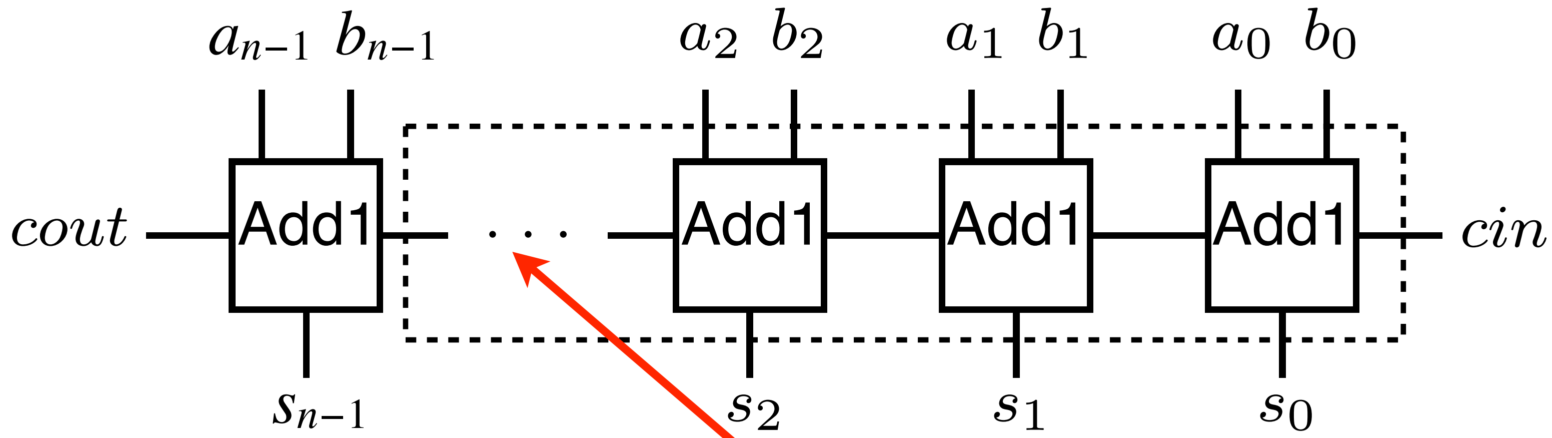

```
text{* Implementation of an n-bit ripple-carry adder*}

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))"
```

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))"
```

Adder Implementation

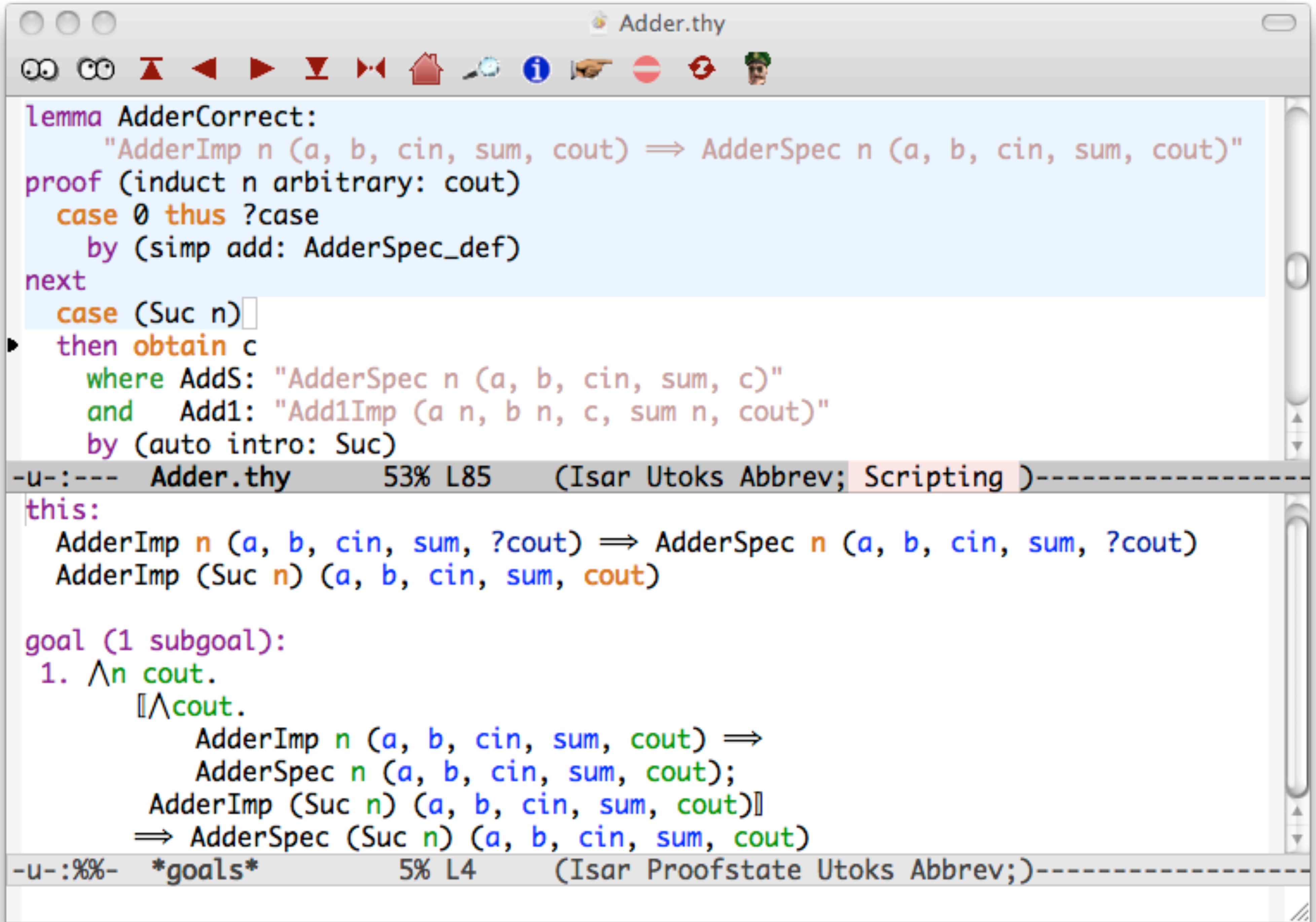


internal wire, to be hidden

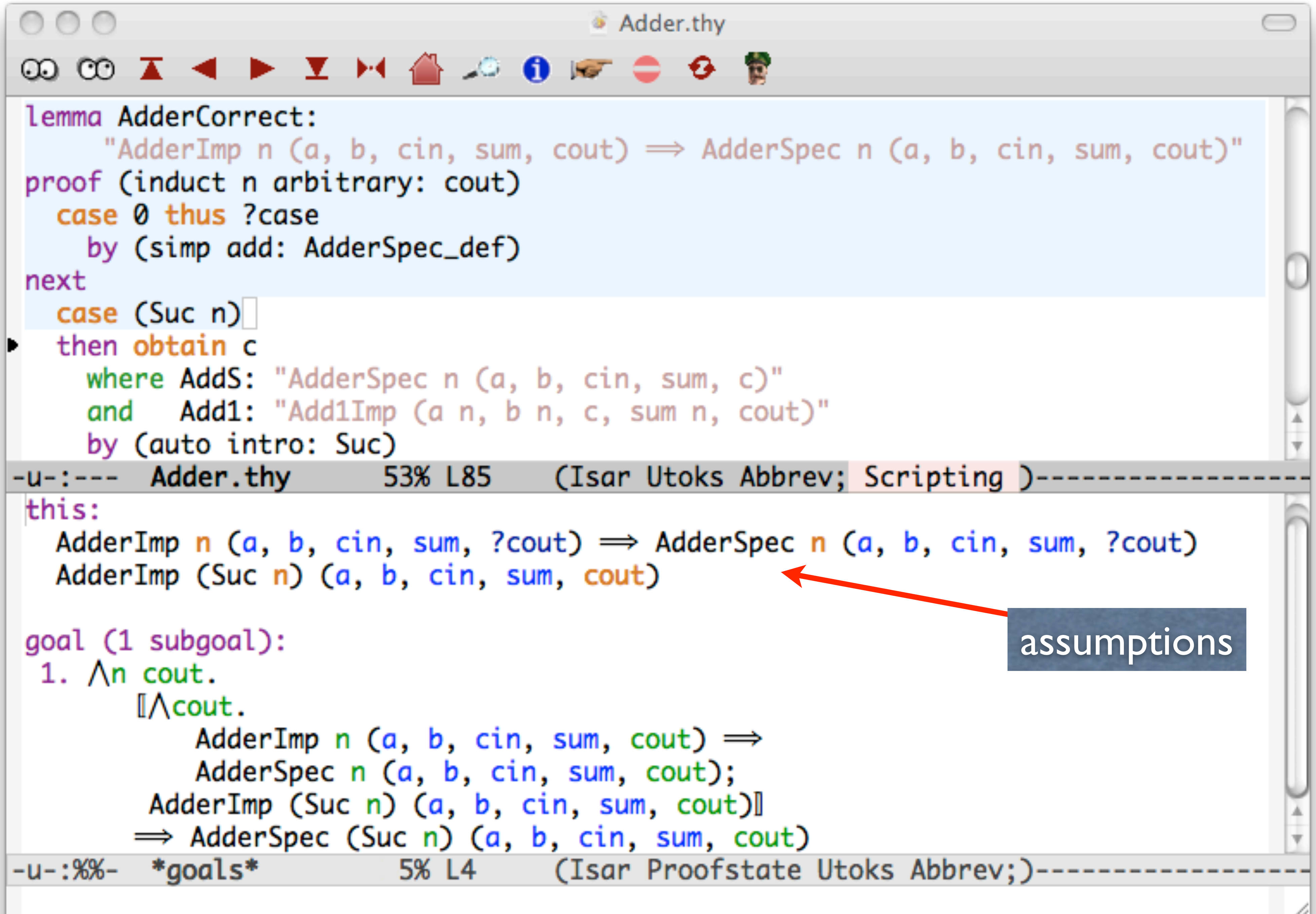
```
text{* Implementation of an n-bit ripple-carry adder*}

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

a zero-bit adder simply connects the carry lines!

Partial Correctness Proof



Partial Correctness Proof



Partial Correctness Proof

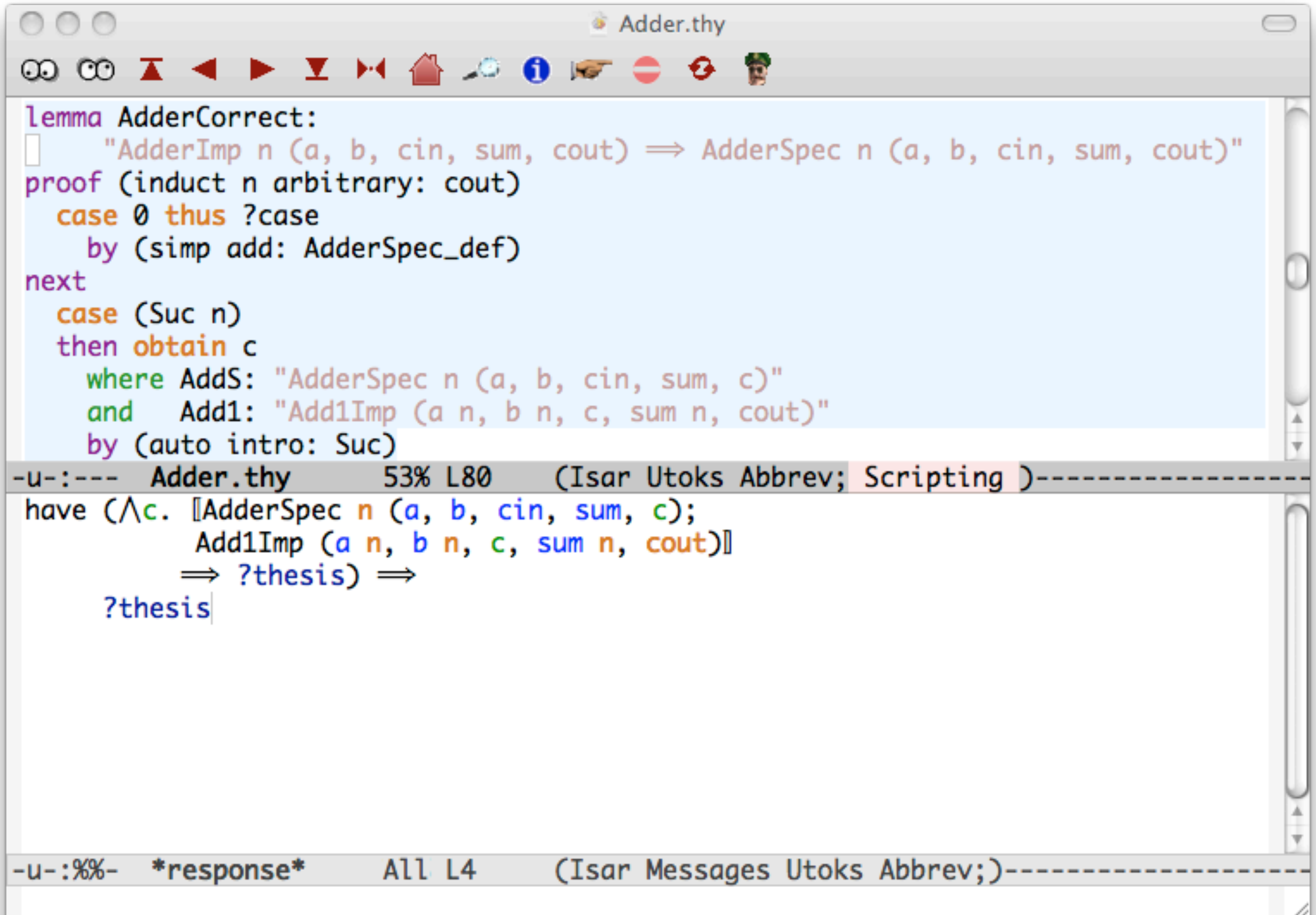
Adder.thy

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

goal (1 subgoal):
 1. ⋀n cout.
       ⟦⋀cout.
           AdderImp n (a, b, cin, sum, cout) ⟹
           AdderSpec n (a, b, cin, sum, cout);
        AdderImp (Suc n) (a, b, cin, sum, cout)⟧
       ⟹ AdderSpec (Suc n) (a, b, cin, sum, cout)
-u-:%%-  *goals*         5% L4     (Isar Proofstate Utoks Abbrev;)----------------
```

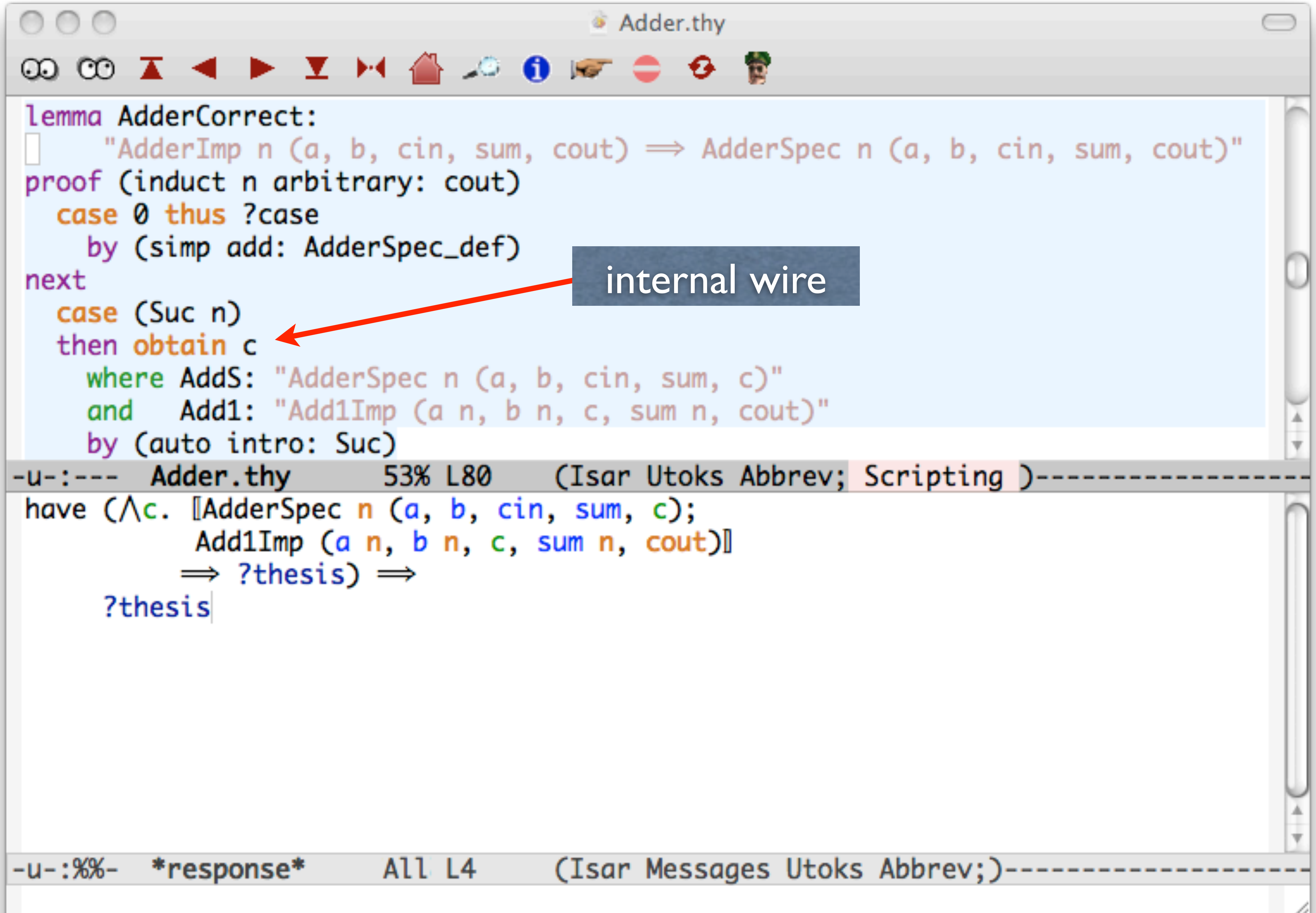
assumptions

conclusion

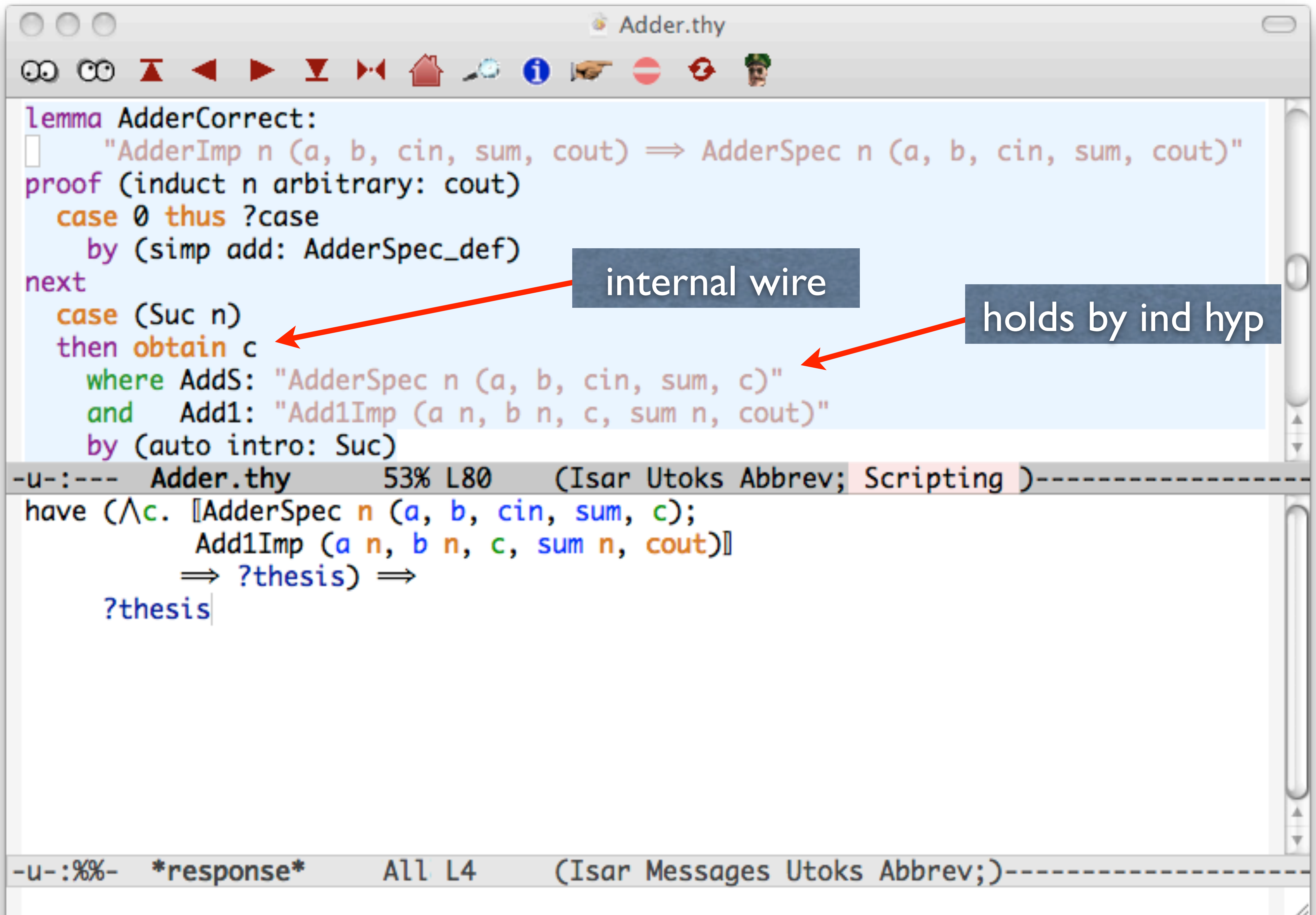
Using the Induction Hypothesis



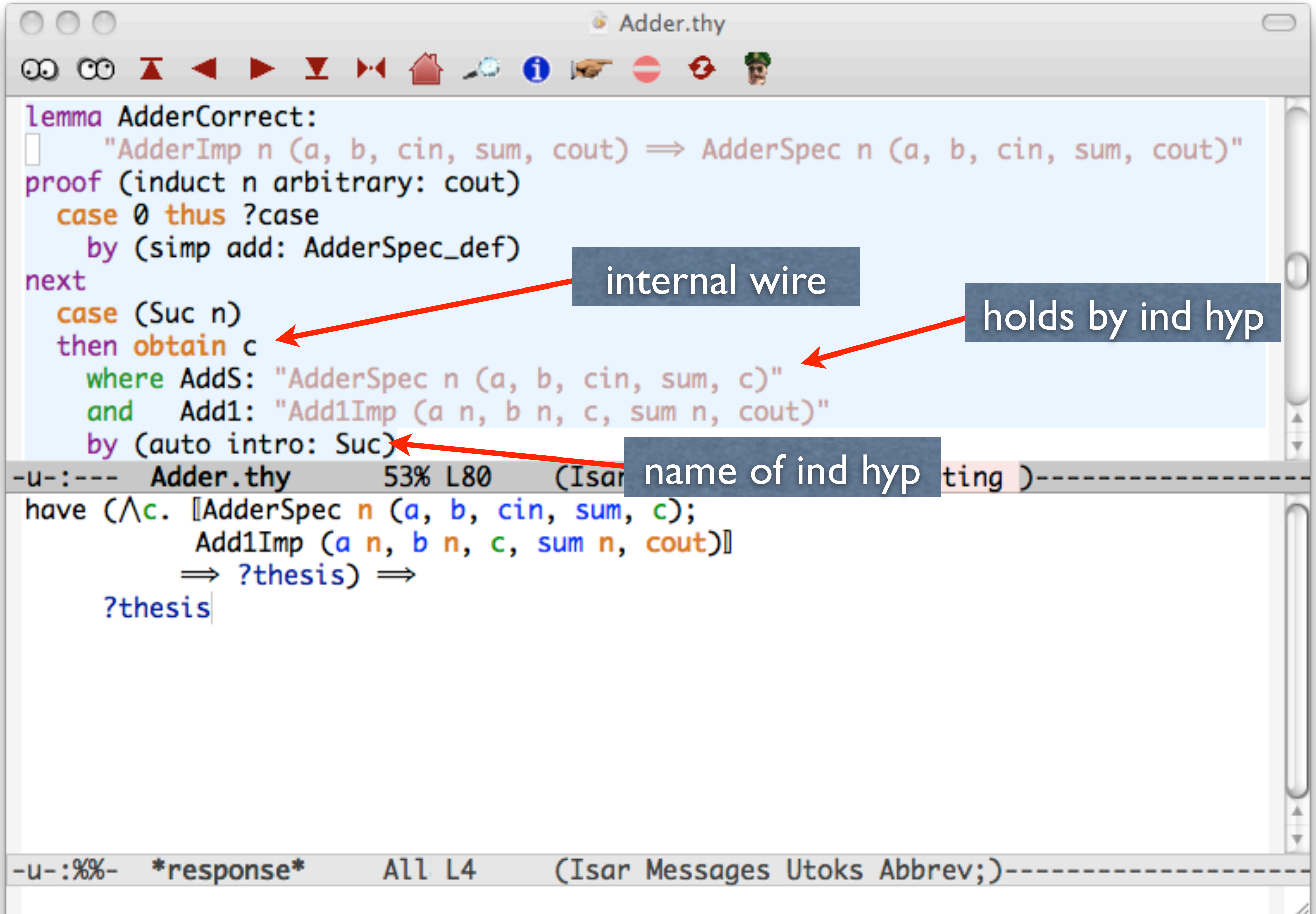
Using the Induction Hypothesis



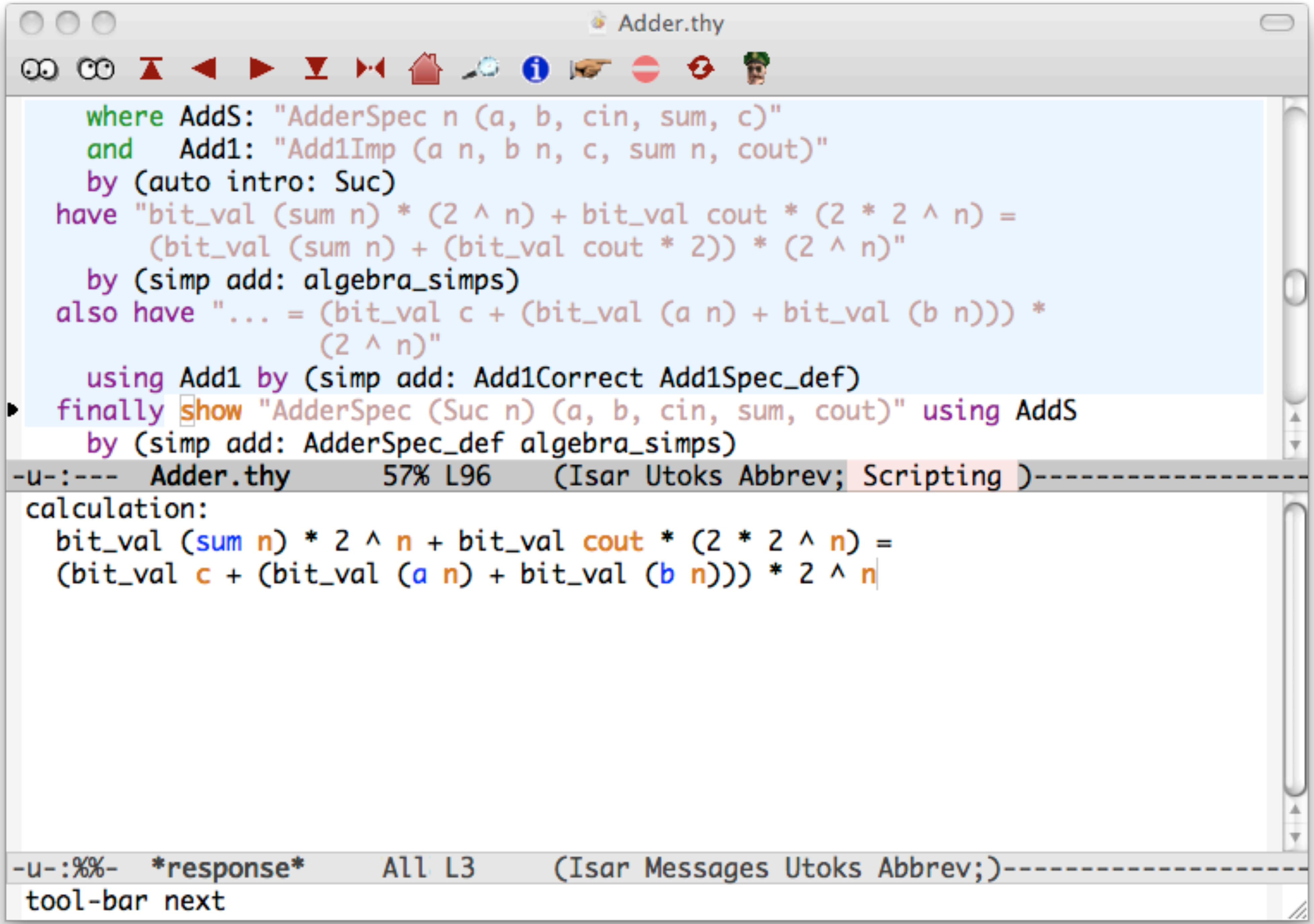
Using the Induction Hypothesis



Using the Induction Hypothesis



A Tiresome Calculation

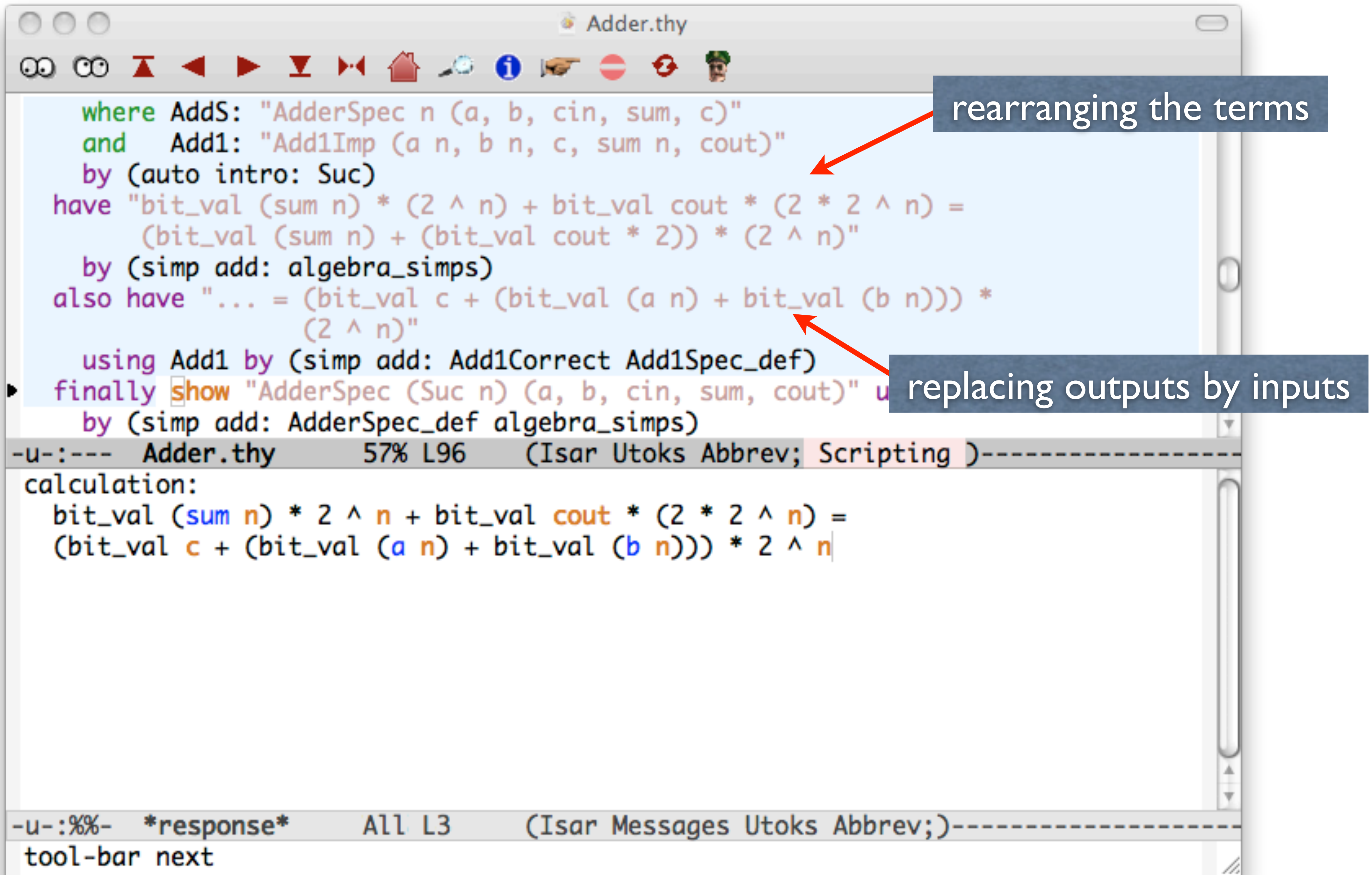


A Tiresome Calculation

```
Adder.thy

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

rearranging the terms

A Tiresome Calculation



The Finished Proof

Adder.thy

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

```
-u-:---  Adder.thy      51% L78     (Isar Utoks Abbrev; Scripting )----------------
```

The Finished Proof

Adder.thy

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

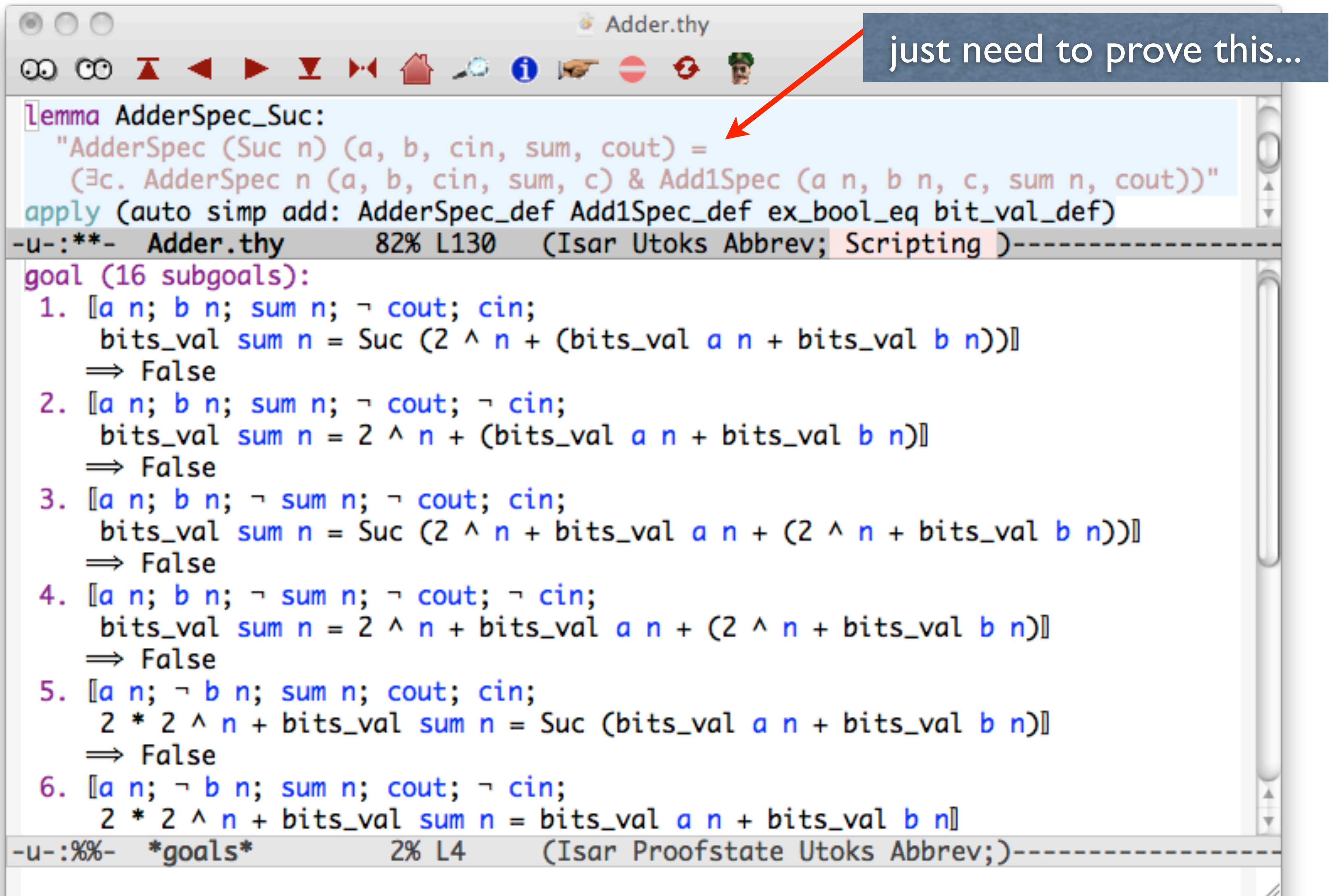
implementation ⇒ specification

```
-u-:---  Adder.thy      51% L78     (Isar Utoks Abbrev; Scripting )----------------
```

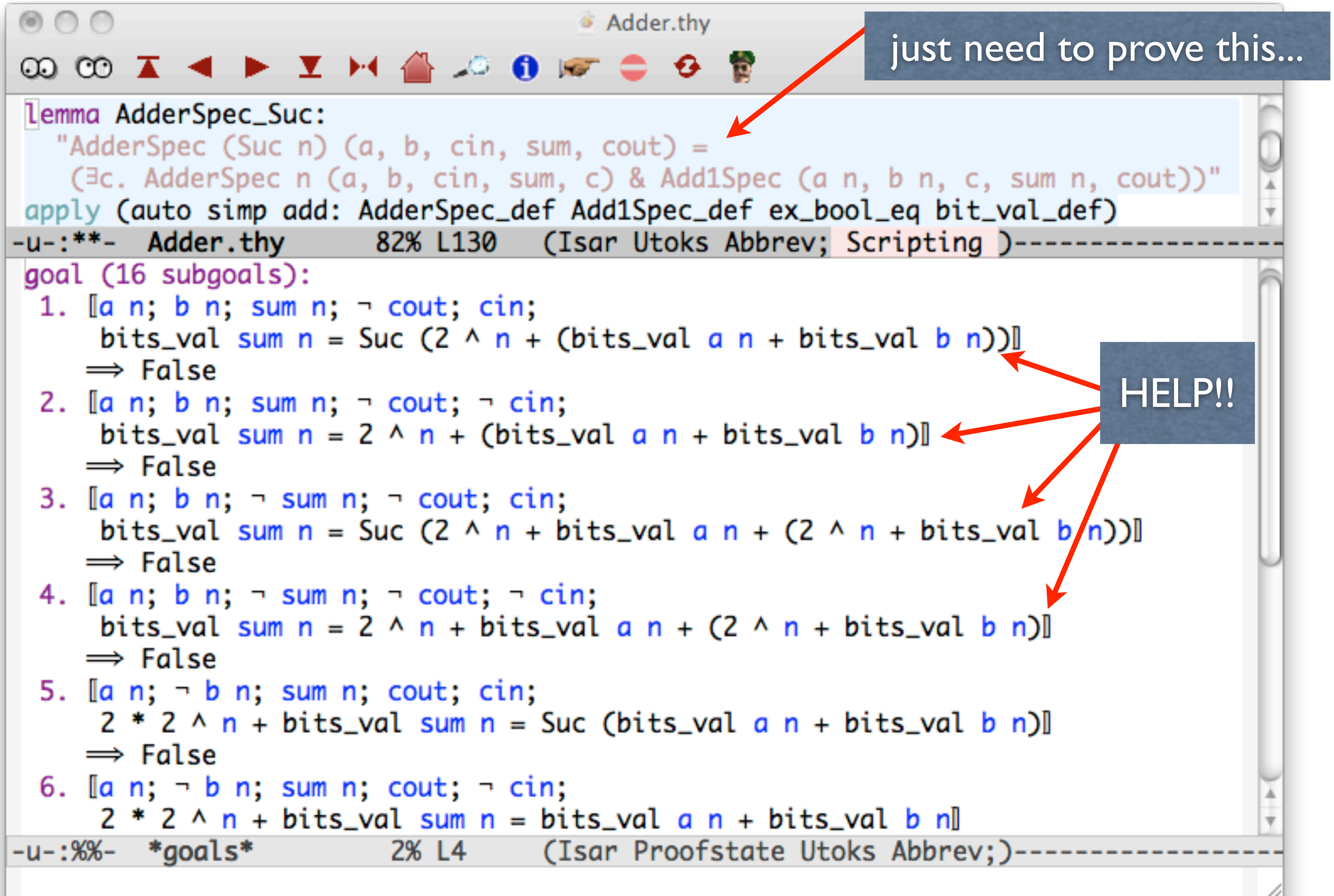
Proving Equivalence

```
Adder.thy

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

Proving Equivalence



Proving Equivalence



A Crucial Lemma

Adder.thy

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

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

-u-:---  Adder.thy      85% L139    (Isar Utoks Abbrev; Scripting )----------------
proof (prove): step 1

using this:
  bits_val a n < 2 ^ n
  bits_val b n < 2 ^ n
  bits_val sum n < 2 ^ n

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

-u-:%%-  *goals*         1% L2      (Isar Proofstate Utoks Abbrev;)----------------
```

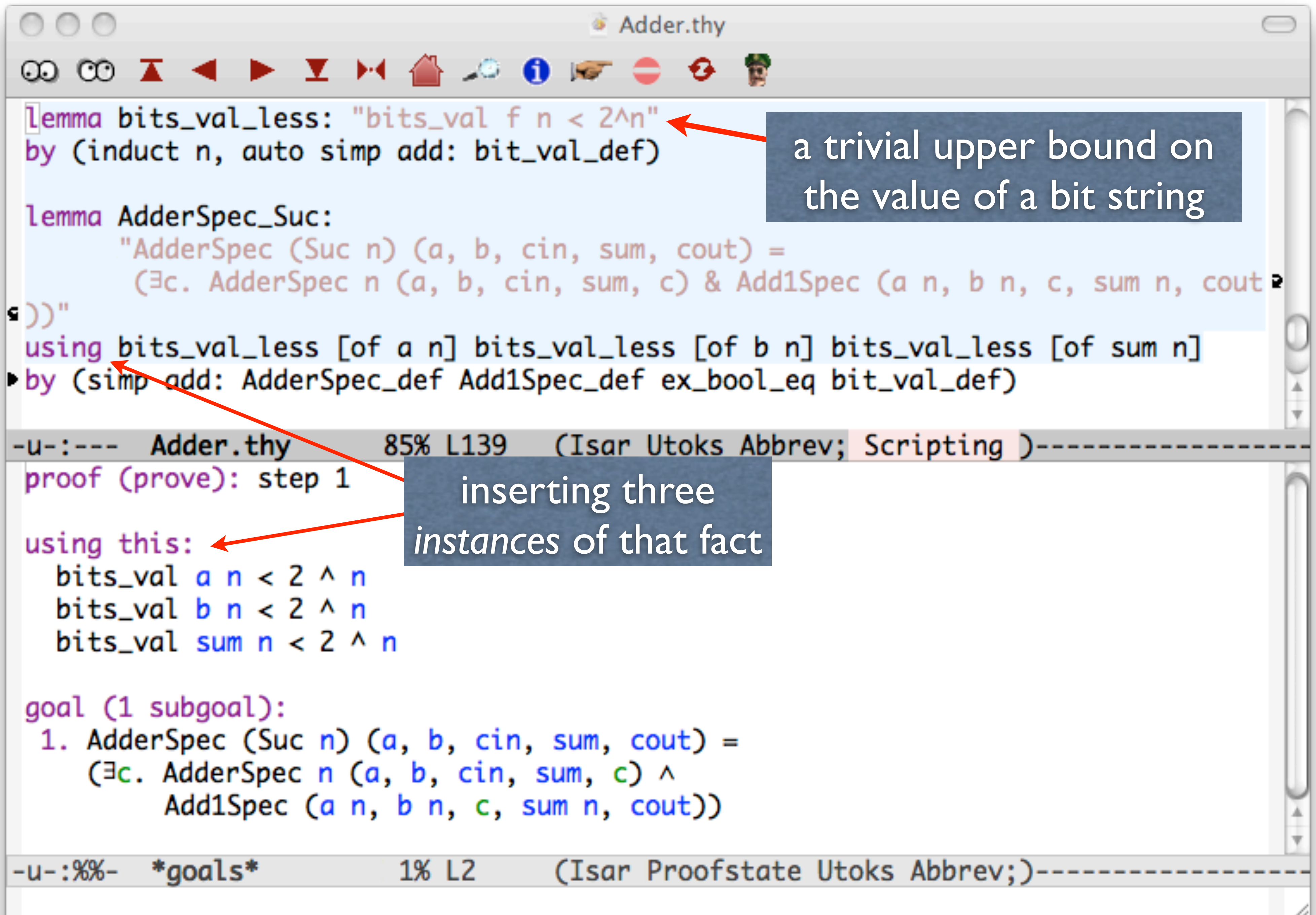
A Crucial Lemma

Adder.thy

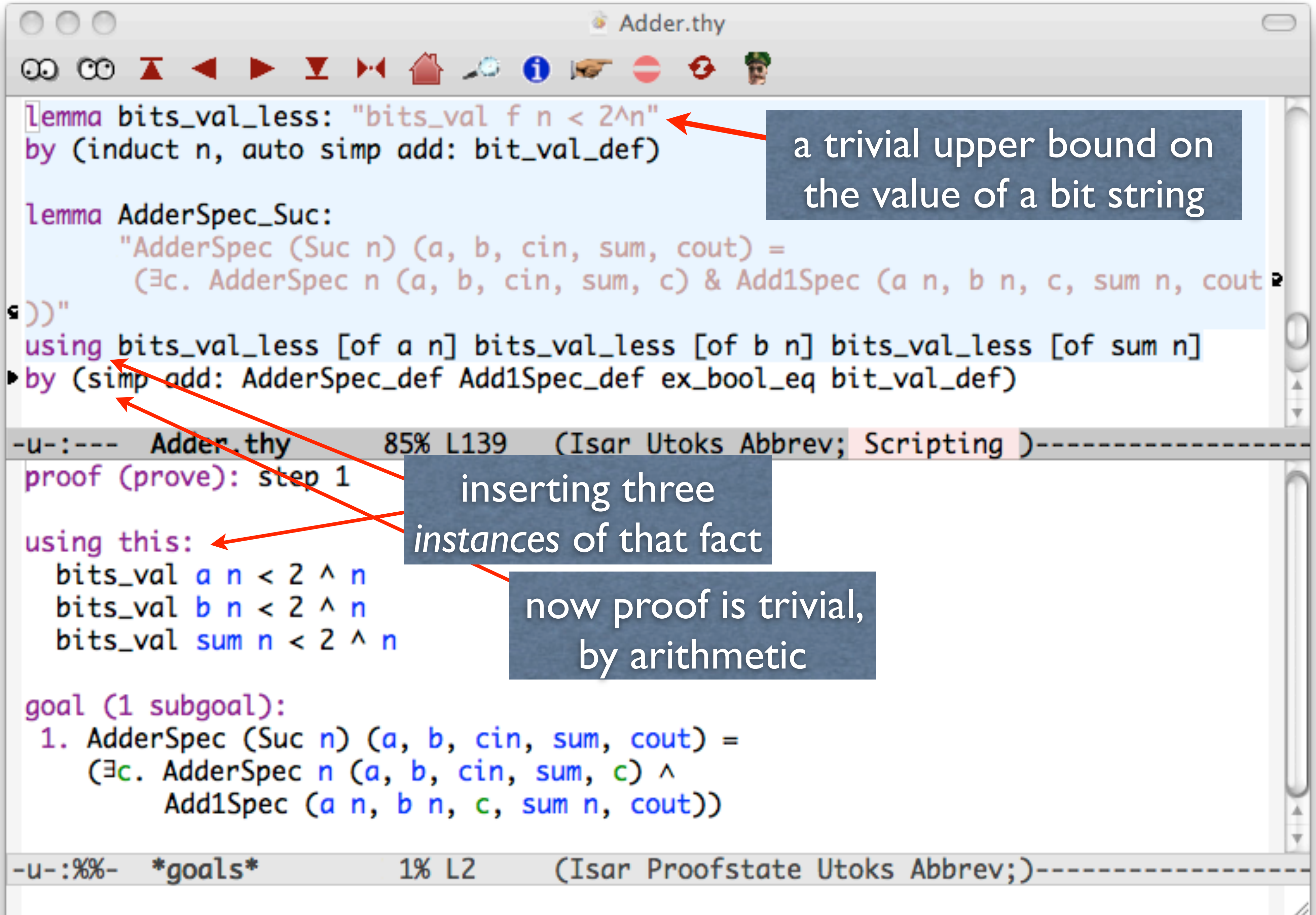
```
lemma bits_val_less: "bits_val f n < 2^n"
by (induct n, auto simp add: bit_val_def)

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

-u-:---  Adder.thy       85% L139    (Isar Utoks Abbrev; Scripting )----------------
proof (prove): step 1

using this:
  bits_val a n < 2 ^ n
  bits_val b n < 2 ^ n
  bits_val sum n < 2 ^ n

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

-u-:%%-  *goals*          1% L2      (Isar Proofstate Utoks Abbrev;)----------------
```

a trivial upper bound on the value of a bit string

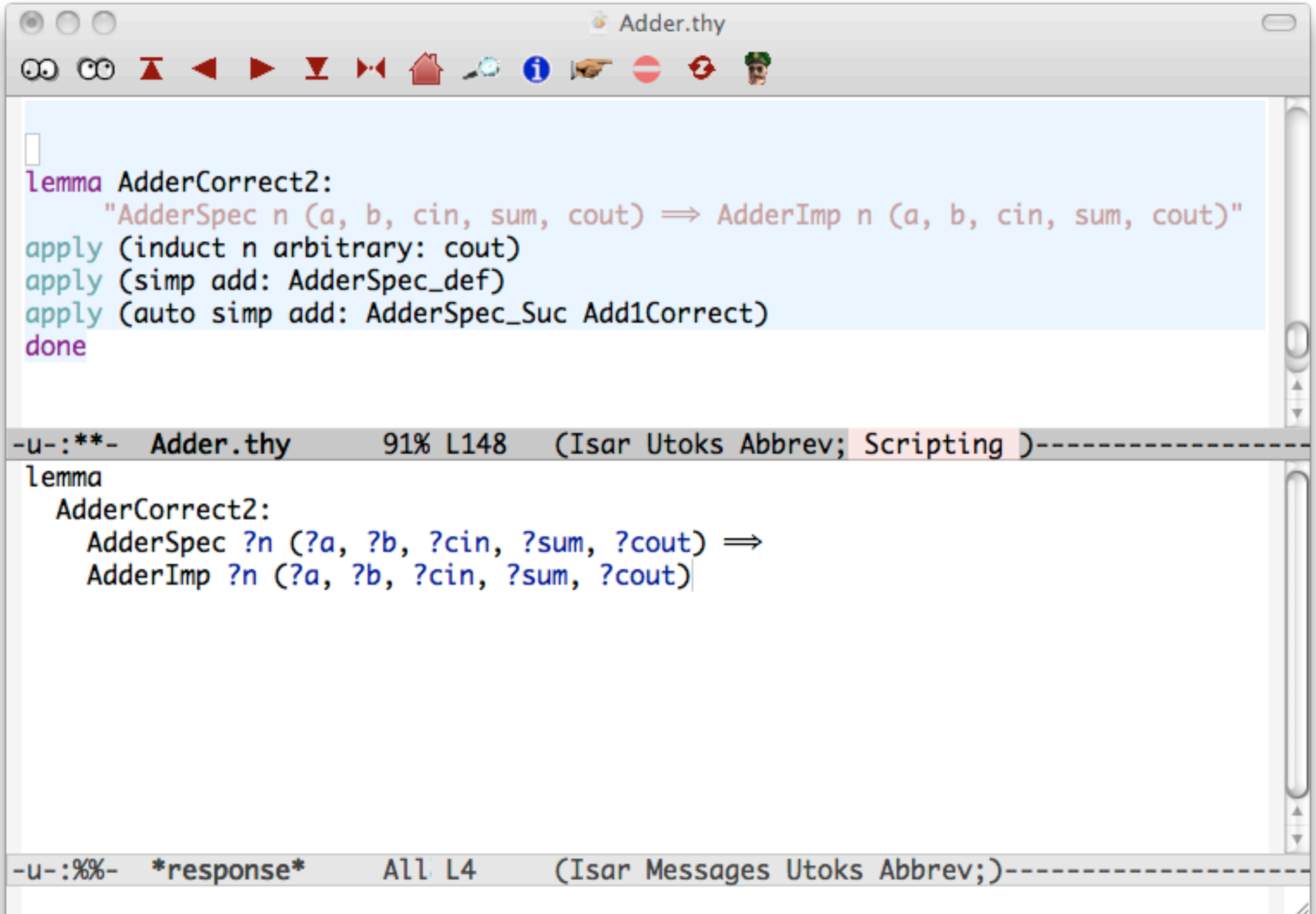
A Crucial Lemma



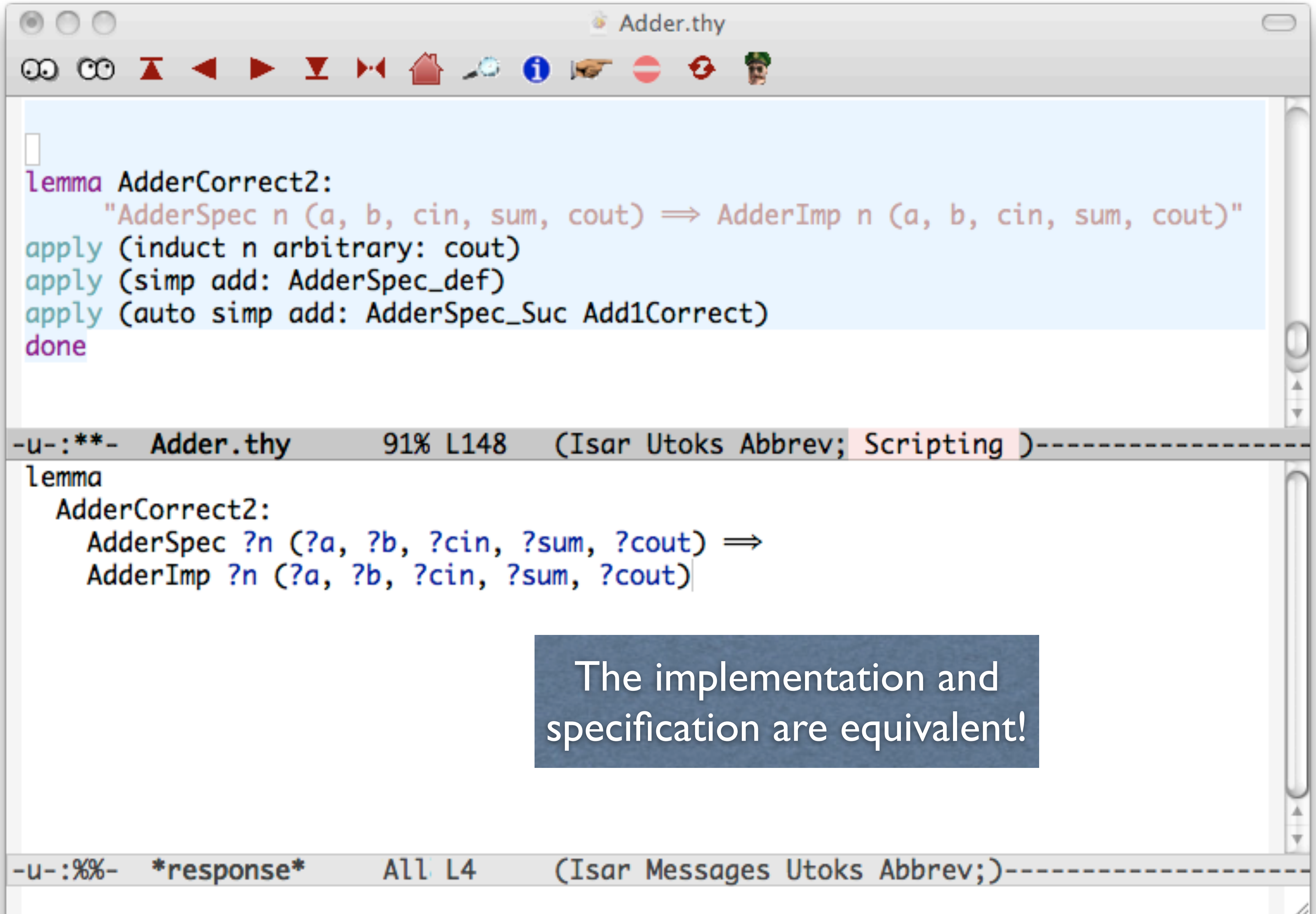
A Crucial Lemma



The Opposite Implication



The Opposite Implication



Making Instances of Theorems

Making Instances of Theorems

- *thm* `[of` *a* *b* *c*`]`
  replaces variables by terms from left to right

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

Making Instances of Theorems

- *thm* `[of` a b c`]`
  replaces variables by terms from left to right
- *thm* `[where` x=a`]`
  replaces the variable x by the term a
- *thm* `[OF` thm_1 thm_2 thm_3`]`
  discharges premises from left to right

Making Instances of Theorems

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

Making Instances of Theorems

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

The End

You know my methods. Apply them!

Sherlock Holmes