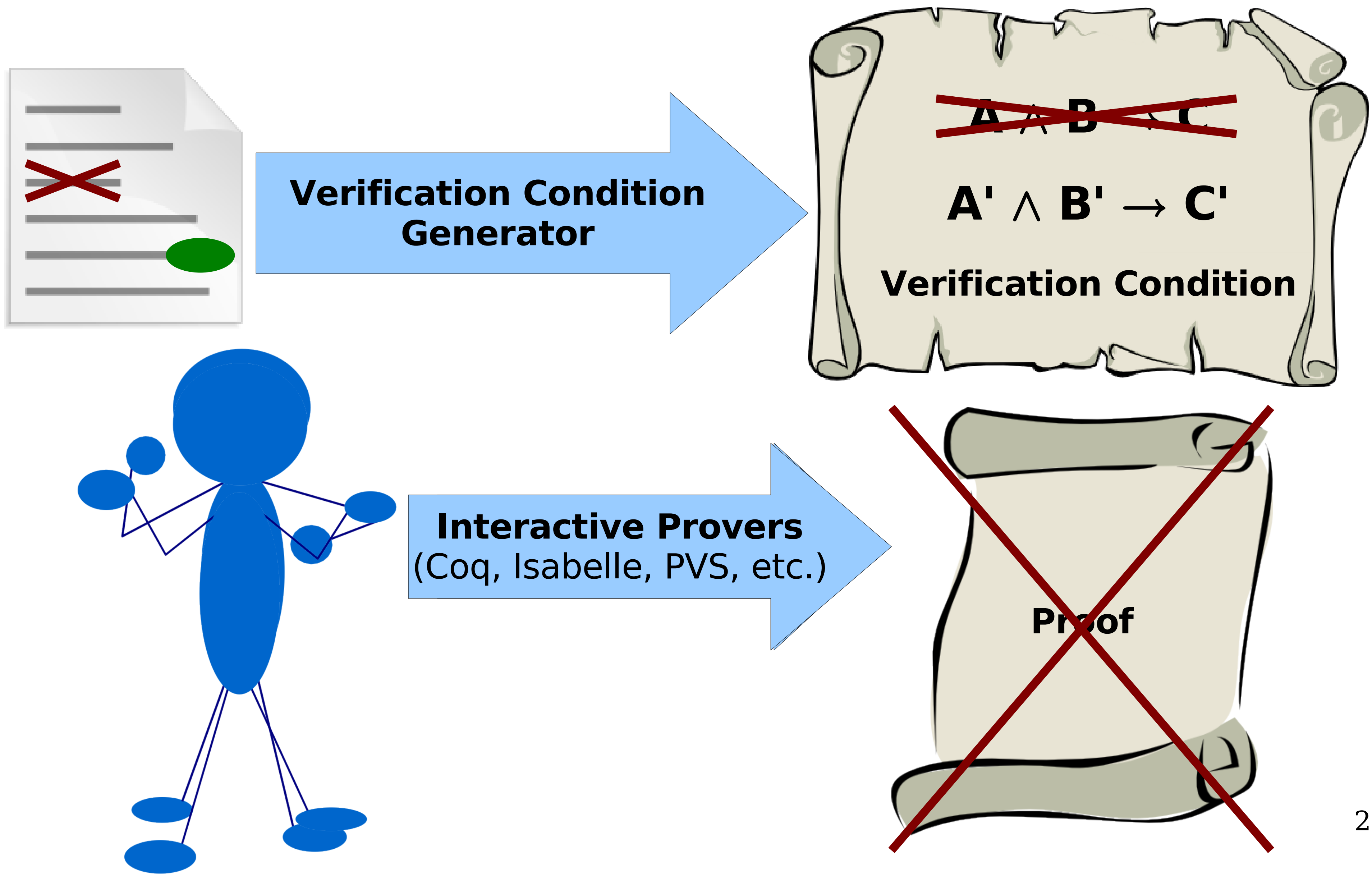
Interactive Computer Theorem Proving

Lecture 7: Programming with Proofs

CS294-9
October 5, 2006
Adam Chlipala
UC Berkeley

Classical Program Verification



Problem #1: Error diagnosis during evolut

Definition foo := ~~....~~ ....'

Definition bar := ....

Lemma L : **forall** x, P (foo x) (bar x).

tactic1.

tactic2.

Qed.

```
Error:
In environment
bound : nat
fm : formula
bound' : nat
_ : True
s0 : {fm' : formula &
      {l : lit |
      (forall a : asgn, satFormula fm a -> satLit l a) /\
      (forall a : asgn,
       (satFormula fm (upd a l) -> satFormula fm' a) /\
       (satFormula fm' a -> satFormula fm (upd a l)))}}
fm' : formula
s1 : {l : lit |
      (forall a : asgn, satFormula fm a -> satLit l a) /\
      (forall a : asgn,
       (satFormula fm (upd a l) -> satFormula fm' a) /\
       (satFormula fm' a -> satFormula fm (upd a l)))}
l : lit
n : forall a : asgn, satFormula fm' a -> False
a0 : asgn
H : satFormula fm a0
H0 : forall a : asgn, satFormula fm a -> satLit l a
H1 : forall a : asgn,
      (satFormula fm (upd a l) -> satFormula fm' a) /\
      (satFormula fm' a -> satFormula fm (upd a l))
a : {fm' : formula &
     {l : lit |
     (forall a : asgn, satFormula fm a -> satLit l a) /\
     (forall a : asgn,
      (satFormula fm (upd a l) -> satFormula fm' a) /\
      (satFormula fm' a -> satFormula fm (upd a l)))}}
a1 : {al : alist | satFormula fm' (interp_alist al)}
H2 : forall fm : formula,
     option
       ({al : alist | satFormula fm (interp_alist al)} +
        {(forall a : asgn, satFormula fm a -> False)})
The term "0" has type "nat" while it is expected to have type
"asgn"
```

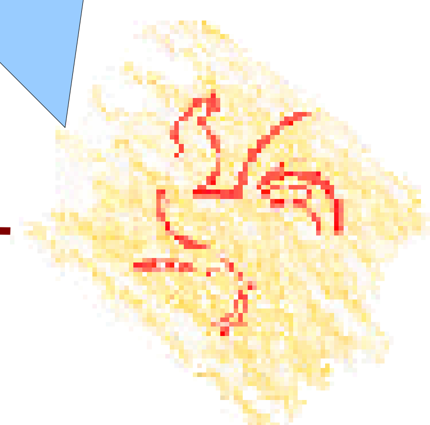
Theorem T : **forall** x, Q (foo x) (bar x).

tactic1.

tactic2.

...

tactic3253.



Problem #2: Effective Modular Development

Definition foo **:=** ….

Definition bar **:=** ….

Import BeppoLib.

Definition baz **:=** … foo … bar ….

Definition baz' **:=** … foo … bar … beppo ….

Theorem foo_correct.

….

Qed.

Theorem bar_correct.

….

Qed.

Theorem baz_correct.

….

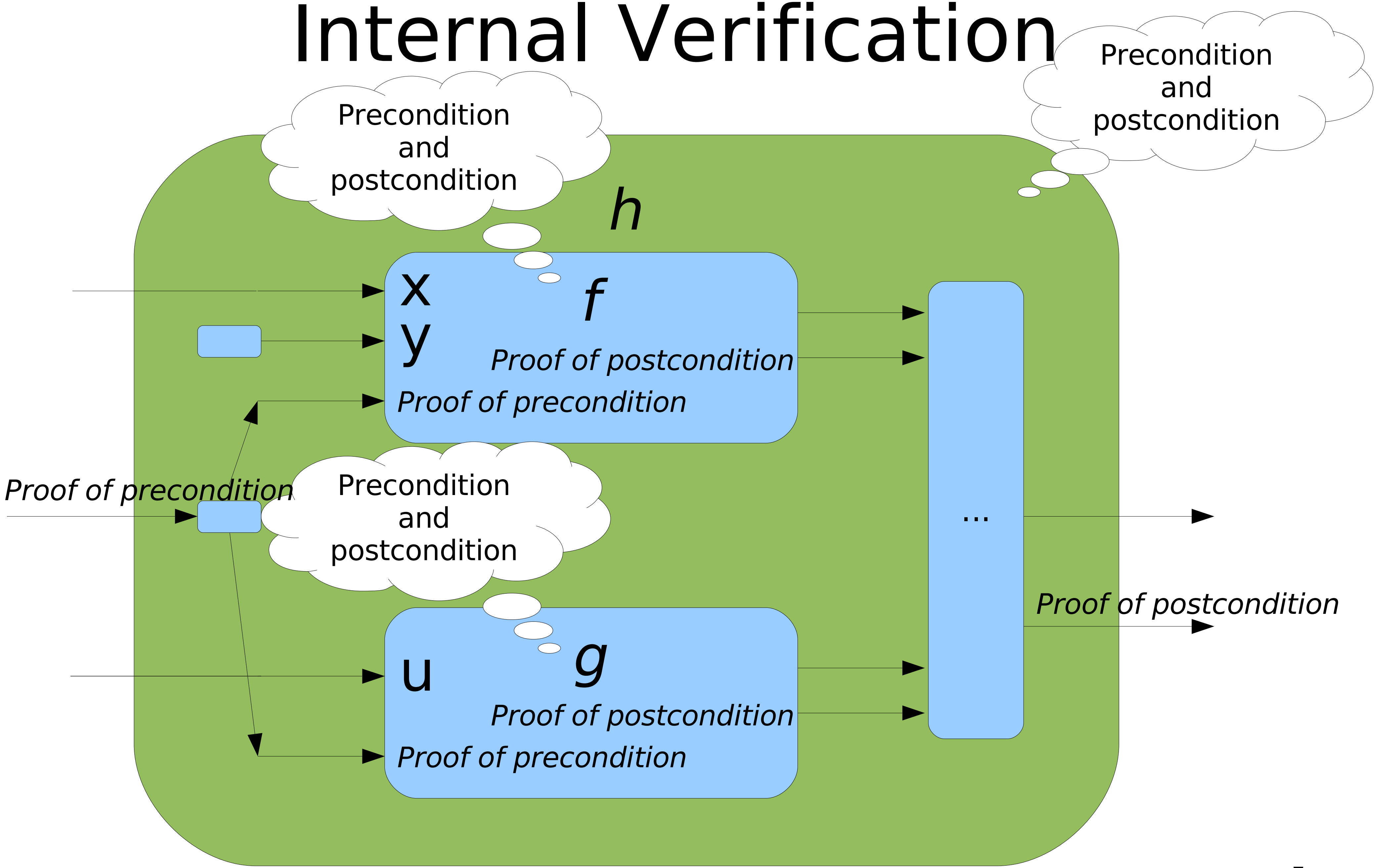
apply foo_correct.

….

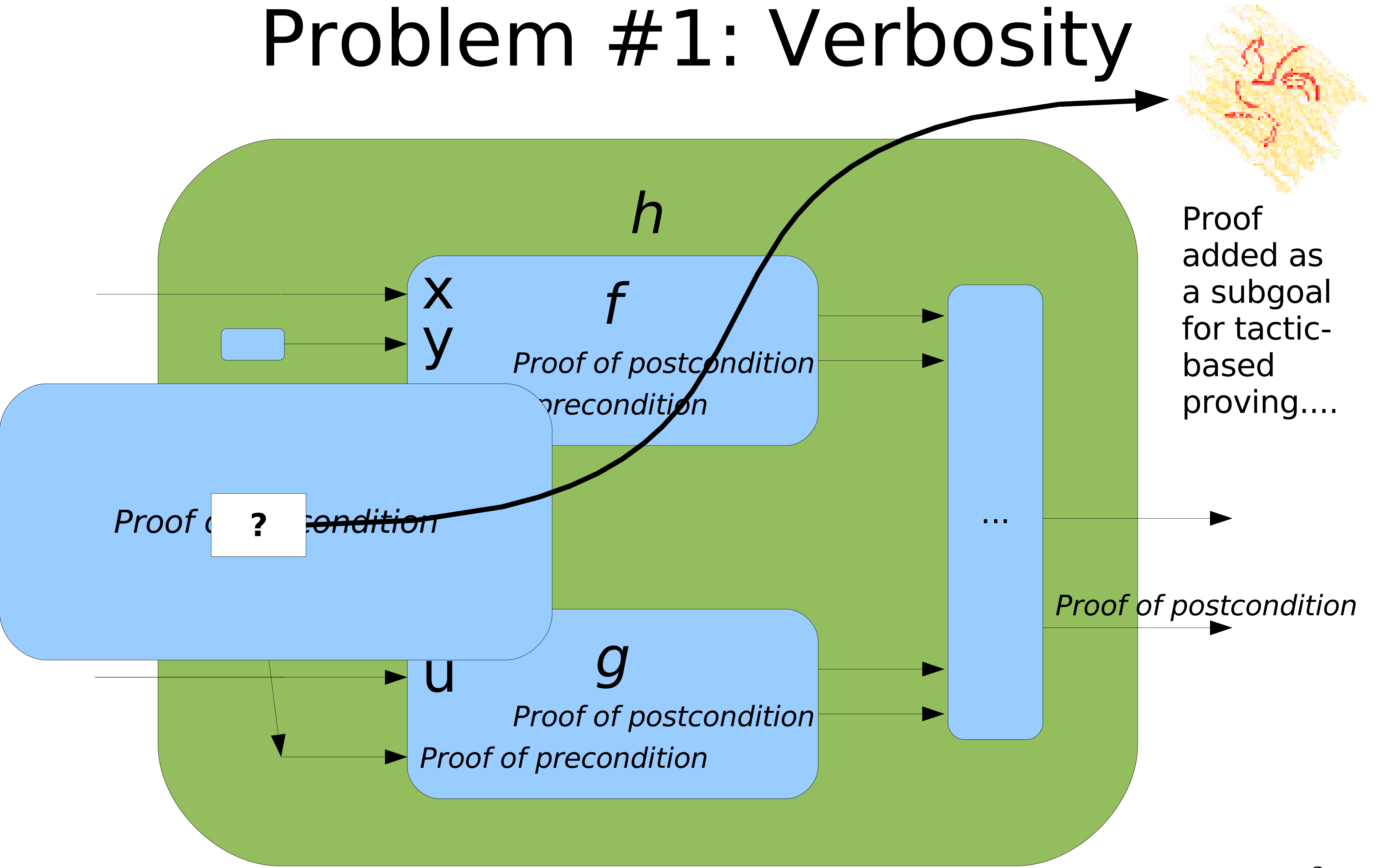
apply bar_correct.

….

Internal Verification



Problem #1: Verbosity



Problem #2: Runtime Inefficiency

- Manipulating proof objects at runtime will slow programs down a lot.
- Some simple algorithms have much larger and more complicated proofs, so proof processing would dominate performance.
- Your computer probably doesn't have enough RAM to store explicit proof objects arising from large inputs to interesting programs.

Program Extraction

The Big Idea:
Proofs are a technical device for establishing program soundness statically. There's no reason to include them in the final programs to execute!

We want some kind of **proof-erasing compiler**.

Challenges:

- What do we cut and what do we keep?
- How can we ensure compilation soundness?

Prop vs. Set

- We've come to the sole reason for distinguishing between types in **Prop** and **Set**: **Prop values are erased during compilation, while Set values are kept**.
- If we didn't want extraction, we could collapse them into a single domain.

> With the default flags, Set and Prop are also differentiated by predicativity.

A Tale of Two Sorts

Inductive unit : **Set** :=
  tt : unit.

Inductive Empty_set : **Set** := .

Inductive prod (A B : **Set**) : **Set** :=
  pair : A -> B -> prod A B.

Inductive sum (A B : **Set**) : **Set** :=
  inl : A -> sum A B
 | inr : B -> sum A B.

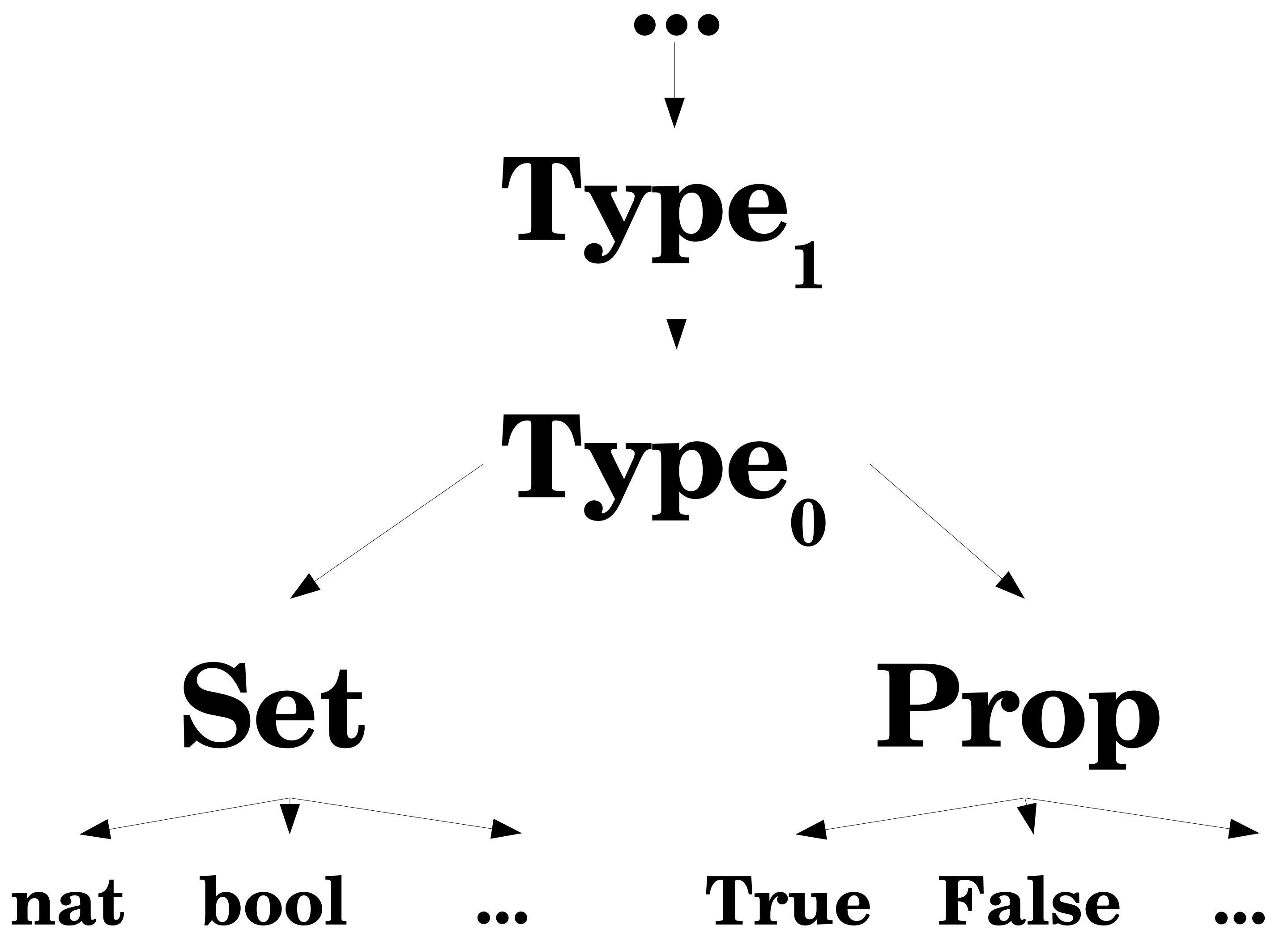
Inductive True : **Prop** :=
  I : True.

Inductive False : **Prop** := .

Inductive and (A B : **Prop**) : **Prop** :=
  conj : A -> B -> and A B.

Inductive or (A B : **Prop**) : **Prop** :=
  or_introl : A -> or A B
 | or_intror : B -> or A B.

The Coq Type Hierarchy



What Coq Extraction Does

- Translate Coq code to OCaml, Haskell, or Scheme *(straightforward part)*
- Systematically replace all Props with unit, so that their values carry no usable information *(tricky part)*

Example:

Definition f : **forall** (n : nat), n > 0 -> nat := ....

↓ Erase Prop

Definition f : **forall** (n : nat), unit -> nat := ....

↓ Simplify types involving unit

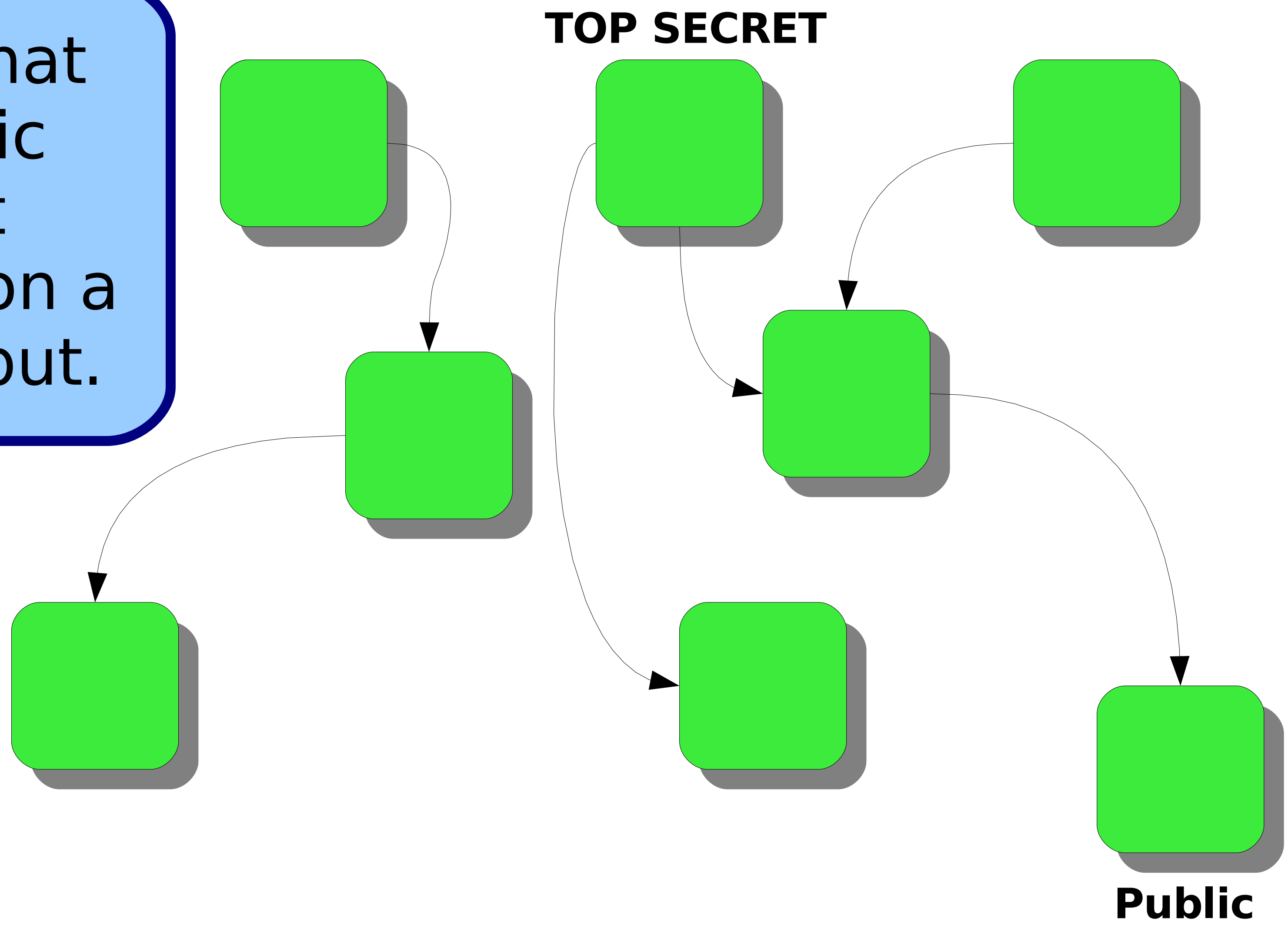
Definition f : **forall** (n : nat), nat := ....

↓ Drop dependent typing and translate to OCaml syntax

let f : nat -> nat := ...

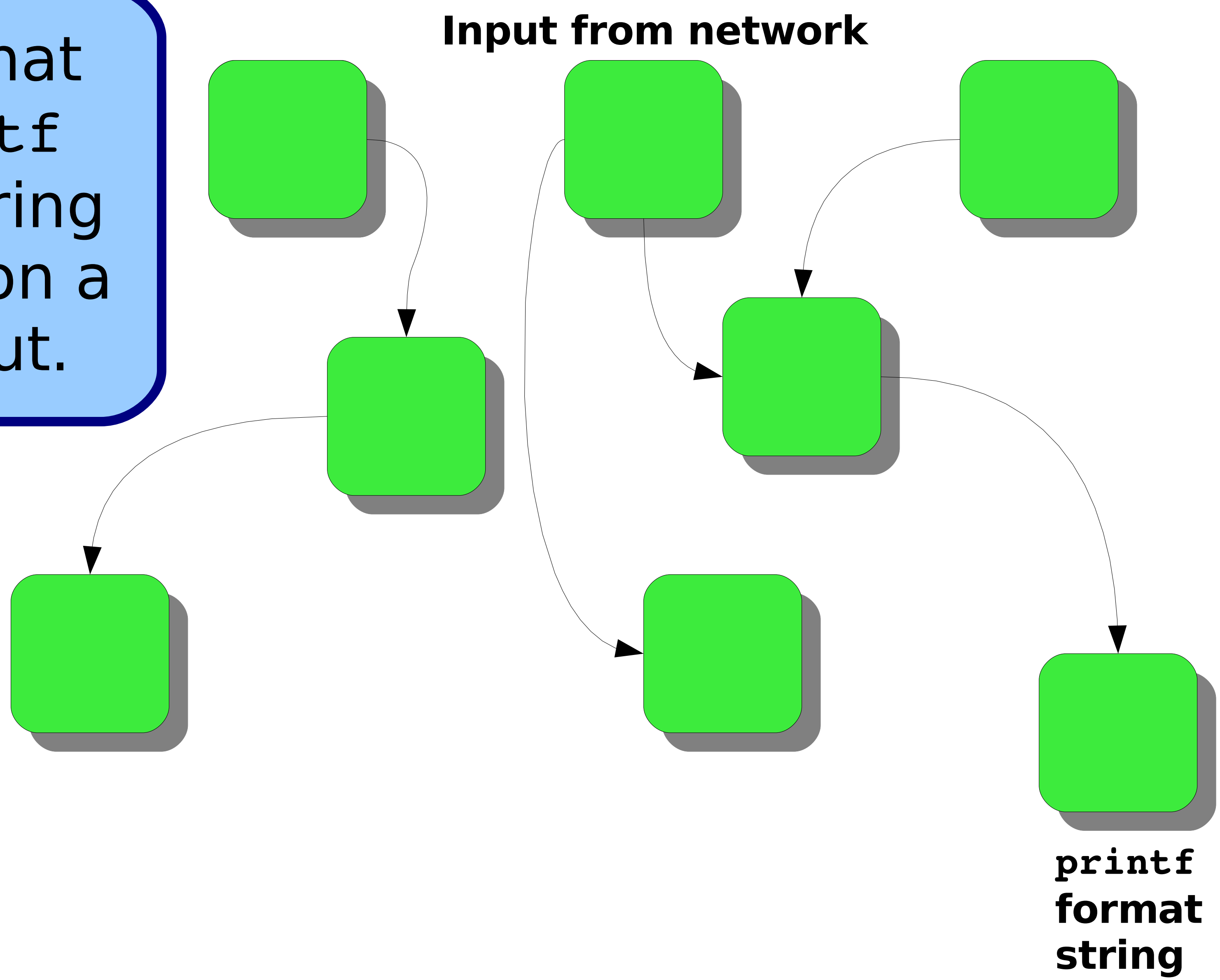
Related Problem: Secure Information Flow

Ensure that no public output depends on a secret input.



Related Problem: Taint Analysis

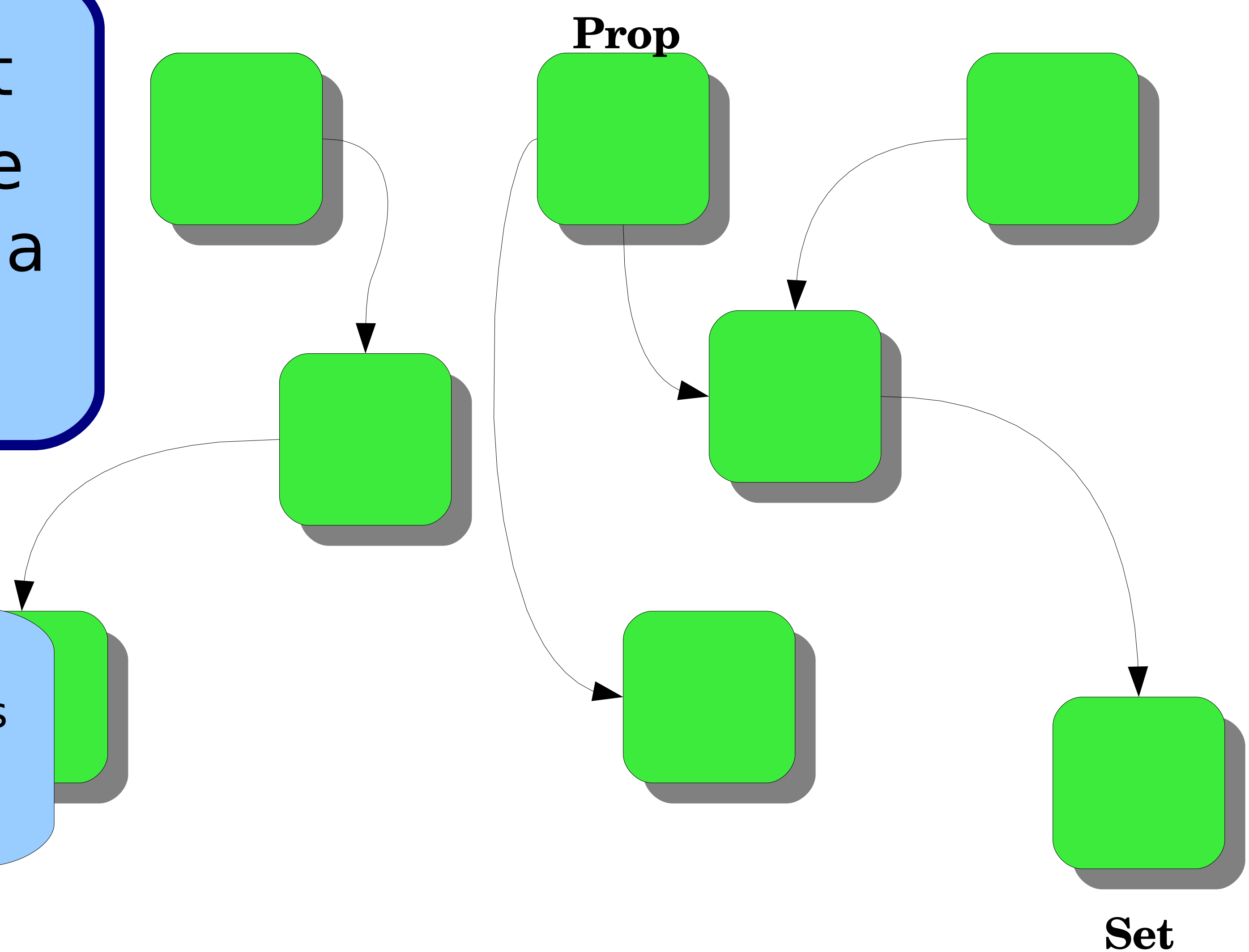
Ensure that no `printf` format string depends on a user input.



The Essence of Coq's Type System Support for Extraction

Ensure that no Set value depends on a Prop value.

In a particular way that makes erasure impossible....



Preventing Bad Dependencies by Limiting Eliminations

Type: nat

Sort: **Set**

match n **with**

| O => O

| S n' => n'

end

Body type: nat

Sort: **Set**

We are eliminating a value of sort **Set** to produce a value of sort **Set**.

Elimination Restrictions

Set *->* ***Set*** *example:*

Adding natural numbers

Prop *->* ***Set*** *example:*

From a proof that there exists *x* satisfying *P*, compute *x*.

Eliminate a...

Producing a...	**Set**	**Prop**
Set	✓	✗
Prop	✓	✓

Set *->* ***Prop*** *example:*

Proving properties of addition

Prop *->* ***Prop*** *example:*

“If there exists *x* satisfying *P*, then there exists *y* satisfying *Q*.”