



Type Systems: Part II

Soundness and Features

CIS352 — Spring 2024

Kris Micinski

The Curry-Howard Isomorphism

The Curry-Howard Isomorphism is a name given to the idea that every **typed lambda calculus** expression is a computational interpretation of a **theorem** in a suitable constructive logic.

For STLC: every well-typed term in STLC is a **theorem** in intuitionistic propositional logic (STLC \sim IPL).

So far, we have discussed four rules in STLC: Var, Const, App, and Lam

These rules ***exactly mirror*** corresponding rules in IPL

The **Var** rule corresponds to the **Assumption** rule

In IPL, Γ is a **set** of propositions (assumed true)

In STLC, Γ is a **map** from type variables to their types

$$\frac{x \mapsto t \in \Gamma}{\Gamma \vdash x : t} \quad \mathbf{Var}$$

$$\mathbf{Assumption} \frac{}{\Gamma, P \vdash P}$$

$\Gamma : \mathbf{Var} \rightarrow \mathbf{Type}$

$\Gamma : \mathbf{Set}(\mathbf{Proposition})$

$$\frac{x \mapsto t \in \Gamma}{\Gamma \vdash x : t} \quad \mathbf{Var}$$

$$\mathbf{Assumption} \frac{}{\Gamma, P \vdash P}$$

$$\frac{\Gamma \vdash e : A \rightarrow B \quad \Gamma \vdash e' : A}{\Gamma \vdash (e \ e') : B} \quad \mathbf{App}$$

$$\Rightarrow \mathbf{E} \frac{\Gamma \vdash A \Rightarrow B \quad \Gamma \vdash A}{\Gamma \vdash B}$$

The **App** rule corresponds to modus ponens in IPL
 Notice how the type is $A \rightarrow B$ but in IPL it is $A \Rightarrow B$

The **Lam** rule introduces assumptions, just as $\Rightarrow\text{I}$ does in IPL

$$\frac{x \mapsto t \in \Gamma}{\Gamma \vdash x : t} \quad \mathbf{Var}$$

$$\frac{\Gamma \vdash e : A \rightarrow B \quad \Gamma \vdash e' : A}{\Gamma \vdash (e \ e') : B} \quad \mathbf{App}$$

$$\frac{\Gamma, \{x \mapsto t\} \vdash e : A}{\Gamma \vdash (\lambda (x : t) \ e) : A \rightarrow B} \quad \mathbf{Lam}$$

$$\mathbf{Assumption} \frac{}{\Gamma, P \vdash P}$$

$$\Rightarrow\mathbf{E} \frac{\Gamma \vdash A \Rightarrow B \quad \Gamma \vdash A}{\Gamma \vdash B}$$

$$\Rightarrow\mathbf{I} \frac{\Gamma, A \vdash B}{\Gamma \vdash A \Rightarrow B}$$

What this means is that any time you write a proof tree in STLC, you *could have* written it in IPL instead

There is an *exact correspondence* between proof trees in IPL and STLC

This deep result means that we can execute proofs in constructive logic as programs, and that we can build programs which contain proof-relevant components (e.g., for building correct-by-construction systems).

Aside from the theoretical results, we will now use the Curry-Howard Isomorphism for a more practical reason: to add typing rules necessary for us to write more complicated programs, using forms such as pairs, etc...

We can add pairs to STLC via several additional rules, inspired by intuitionistic logic

$\begin{aligned} e ::= & (\text{lambda } (x : t) e) \\ & (e e) \\ & \dots \\ & (\text{cons } e e) \text{ ;; pairs} \\ & (\text{car } e) \mid (\text{cdr } e) \end{aligned}$	$\begin{aligned} t ::= & \text{int} \mid \text{bool} \mid \dots \\ & t \times t \text{ ;; product types} \end{aligned}$
---	---

The *type* of a pair is a product type:

The *computational* interpretation of \wedge is a pair, so we add syntax for pairs into our language

$(\text{cons } 5 \text{ \#}t) : \text{int} \times \text{bool}$

"If e is a pair, $(\text{car}/\text{cdr } e)$ is the type of its first/second element"

"If e_0 is type A and e_1 is type B , $(\text{cons } e_0 e_1)$ is type $A \times B$ "

$$\mathbf{\times E1} \quad \frac{\Gamma \vdash e : A \times B}{\Gamma \vdash (\text{car } e) : A}$$

$$\mathbf{\times E2} \quad \frac{\Gamma \vdash e : A \times B}{\Gamma \vdash (\text{cdr } e) : B}$$

$$\mathbf{\times I} \quad \frac{\Gamma \vdash e_0 : A \quad \Gamma \vdash e_1 : B}{\Gamma \vdash (\text{cons } e_0 e_1) : A \times B}$$

There are also **discriminated union** types

You may know these as C tagged **unions**

```
e ::= ... ;; previous forms
    | left e
    | right e
    | case e of
        (left e0 => e0')
        (right e1 => e1')
```

The computational interpretation
of v is a *discriminated union*

```
t ::= ... | t + t
```

Now we have **sum** types

```
(inj_left 42) : int × bool
```

Also many other types

```
(inj_left 42) : int × int
```

```
(inj_left 42) : int × (int -> int)
```

```
(inj_left 42) : int × (int × int)
```

...

A discriminated union $A \times B$ says:

“I carry *either* information of type A , or information of type B ; but I can’t promise it’s exactly A or exactly B —thus, to interact with the information, you must *always* do case analysis (i.e., matching).

```
e ::= ... ;; previous forms
    | left e
    | right e
    | case e of
        (left e0 => e0')
        (right e1 => e1')
```

The computational interpretation
of v is a *discriminated union*

```
(case (right 5) of
  (left e => e)
  (right e => 7)) ;; 7
```

```
;; In OCaml, we would write this:
# type ('a, 'b) t = Left of 'a | Right of 'b;;
type ('a, 'b) t = Left of 'a | Right of 'b
# Left (5);;
-: (int, 'a) t = Left 5
;; OCaml's type system supports general ADTs
```

Vanilla STLC

$$\begin{array}{c}
 \frac{\Gamma \vdash e : t \rightarrow t' \quad \Gamma \vdash e' : t}{\Gamma \vdash (e \ e') : t'} \mathbf{App} \quad \frac{}{\Gamma \vdash n : \mathbf{num}} \mathbf{Const} \quad \frac{x \mapsto t \in \Gamma}{\Gamma \vdash x : t} \mathbf{Var} \quad \frac{\Gamma, \{x \mapsto t\} \vdash e : t'}{\Gamma \vdash (\lambda (x : t) \ e) : t \rightarrow t'} \mathbf{Lam}
 \end{array}$$

Products (pairs)

$$\begin{array}{c}
 \mathbf{\times E1} \quad \frac{\Gamma \vdash e : A \times B}{\Gamma \vdash (\text{car } e) : A} \quad \mathbf{\times E2} \quad \frac{\Gamma \vdash e : A \times B}{\Gamma \vdash (\text{cdr } e) : B} \quad \mathbf{\times I} \quad \frac{\Gamma \vdash e_0 : A \quad \Gamma \vdash e_1 : B}{\Gamma \vdash (\text{cons } e_0 \ e_1) : A \times B}
 \end{array}$$

Sums (discriminated unions)

$$\begin{array}{c}
 \mathbf{+I1} \quad \frac{\Gamma \vdash e : A}{\Gamma \vdash (\text{left } e) : A + B} \quad \mathbf{+I2} \quad \frac{\Gamma \vdash e : B}{\Gamma \vdash (\text{right } e) : A + B} \quad \mathbf{+E} \quad \frac{\Gamma \vdash e : A + B \quad \Gamma, e_0 : A \vdash e'_0 : C \quad \Gamma, e_1 : A \vdash e'_1 : C}{\Gamma \vdash (\text{case } e \text{ of } (\text{left } e_0 \Rightarrow e'_0) \ (\text{right } e_1 \Rightarrow e'_1)) : C}
 \end{array}$$

Our full type system: STLC, products, unions, and negation

This type system corresponds precisely to intuitionistic logic with \Rightarrow , \wedge , \vee , and \perp

Negation

$$\neg A \text{ is } A \rightarrow \perp \quad \frac{\Gamma \vdash e : \perp}{\Gamma \vdash (\text{case } e \text{ of}) : t}$$

A family of logics / type systems

Curry-Howard Isomorphism says we can keep adding logic / language features—adding rules to the logics force corresponding rules in the type system

IPL is **boring**—it can't say much. Expressive power is *limited* to propositional logic

To prove interesting theorems, we want to say things like:

$\forall (l : \text{list } A) : \{l' : \text{sorted } l' \wedge \forall x. (\text{member } l \ x) \Rightarrow (\text{member } l' \ x)\}$

- For all input lists l
- The output is a list l' , along with a proof that:
 - (a) l' is sorted (specified elsewhere)
 - (b) every member of l is also a member of l'
- Any issues?
 - (Maybe we also want to assert length is the same?)

Completeness of STLC

- **Incomplete:** Reasonable functions we can't write in STLC
 - E.g., any program using recursion
- Several useful **extensions** to STLC
 - **Fix operator** to allow typing recursive functions
 - **Algebraic data types** to type structures
 - **Recursive types** for full algebraic data types
 - `tree = Leaf (int) | Node(int, tree, tree)`

Typing the Y Combinator

$$\frac{\Gamma \vdash f : t \rightarrow t}{\Gamma \vdash (Yf) : t} \quad \mathbf{Y}$$

The “real” solution is quite nontrivial—we need *recursive types*, which may be formalized in a variety of ways

- We will not cover recursive types in this lecture, I am happy to offer pointers

Our hacky solution works in practice, but is not sound in general

- More precisely, the logic induced by the type system is no longer sound (can prove \perp and therefore everything)

Typing the Y Combinator

Think of how this would look for **fib**

$$\frac{\Gamma \vdash f : t \rightarrow t}{\Gamma \vdash (Yf) : t} \quad \mathbf{Y}$$

(let ([fib

(Y (λ (f) (λ (x)

(if (= x 0)

1

(* x (fib (- x 1))))))])

What would t be here?

Error States

A program steps to an **error state** if its evaluation reaches a point where the program has not produced a value, and yet cannot make progress

$$((+ \ 1) \ (\lambda \ (x) \ x))$$

Gets “stuck” because + can’t operate on λ

Error States

A program steps to an **error state** if its evaluation reaches a point where the program has not produced a value, and yet cannot make progress

$$((+ \ 1) \ (\lambda \ (x) \ x))$$

Gets “stuck” because + can’t operate on λ

(Note that this term is **not typable** for us!)

Soundness

A type system is **sound** if no typable program will ever evaluate to an error state

“Well typed programs cannot go wrong.”
— Milner

(You can **trust** the type checker!)

Proving Type Soundness

Theorem: if e has some type derivation, then it will evaluate to a value.

Relies on two lemmas

Progress

If e typable, then it is either a value or can be further reduced

Preservation

If e has type t , any reduction will result in a term of type t

Progress		Preservation
If e typable, then it is either a value or can be further reduced		If e has type t , any reduction will result in a term of type t

(In our system) not too hard to prove by induction on the typing derivation.

Combination of progress and preservation says: you can either take a well-typed step and maintain the invariant, or you are done (at a value).

We will skip the proof—it depends on understanding induction over derivations, chat with me if interested...

Type Inference

Allows us to leave some **placeholder** variables that will be “filled in later”

$$((\lambda (x:t) x:t') : \text{int} \rightarrow \text{int})$$

The $\text{int} \rightarrow \text{int}$ constraint then **forces** $t = \text{int}$
and $t' = \text{int}$

Type Inference

Type inference can **fail**, too...

```
(λ (x) (λ (y:int->int) ((+ (x y)) x))))
```

No **possible** type for x! Used as fn and arg to +

Type Inference has been of interest (research and practical) for many years

It allows you to write **untyped** programs (much less painful!) and automatically *synthesize* a type for you—as long as the type **exists** (catch your mistakes)

$$\begin{array}{c} (\lambda (f) (((f\ 2)\ 3)\ 4)) \\ \downarrow \text{Type inference} \\ (\lambda (f : \text{int} \rightarrow \text{int} \rightarrow \text{int} \rightarrow \text{int}) (((f\ 2)\ 3)\ 4)) \end{array}$$

Type inference can be seen as einterating **all possible type assignments** to infer a valid typing. You can think of it as solving the equation:

$$\exists T. (\lambda (f : T) (((f\ 2)\ 3)\ 4))$$

What is the correct type?

```
(lambda (f) (lambda (x) (if (if-zero? (f x)) 1 0)))
```

Is it:

- (a) $f = \text{int} \rightarrow \text{int}$, $x = \text{int}$
- (b) $f = \text{bool} \rightarrow \text{int}$, $x = \text{bool}$
- (c) $f = (\text{int} \rightarrow \text{int}) \rightarrow \text{int}$, $x = \text{int} \rightarrow \text{int}$

What is the correct type?

```
(lambda (f) (lambda (x) (if (if-zero? (f x)) 1 0)))
```

Is it:

- (a) $f = \text{int} \rightarrow \text{int}$, $x = \text{int}$
- (b) $f = \text{bool} \rightarrow \text{int}$, $x = \text{bool}$
- (c) $f = (\text{int} \rightarrow \text{int}) \rightarrow \text{int}$, $x = \text{int} \rightarrow \text{int}$
- (d) *All of the above***

What is the correct type?

```
(lambda (f) (lambda (x) (if (if-zero? (f x)) 1 0)))
```

Is it:

(a) $f = \text{int} \rightarrow \text{int}$, $x = \text{int}$

(b) $f = \text{bool} \rightarrow \text{int}$, $x = \text{bool}$

(c) $f = (\text{int} \rightarrow \text{int}) \rightarrow \text{int}$, $x = \text{int} \rightarrow \text{int}$

(d) All of the above

Lesson: pick a *principal* which subsumes them all, to avoid enumerating infinitely-many types.

Type Variables

```
(lambda (f) (lambda (x) (if (if-zero? (f x)) 1 0)))
```

Lesson:

We can't pick *just one* type. Instead, we need to be able to instantiate f and x whenever a suitable type may be found.

For example, what if we **let-bind** the lambda and use it in two different ways!?

```
(let ([g (lambda (f) (lambda (x) (if (if-zero? (f x)) 1 0)))])  
  (+ ((g (lambda (x) x)) 0) ((g (lambda (x) 1)) #f)))
```

This usage requires $f = \text{nat} \rightarrow \text{nat}$ and $x = \text{nat}$

This usage requires $f = \text{bool} \rightarrow \text{nat}$ and $x = \text{bool}$

Generalizations

```
(lambda (f) (lambda (x) (if (if-zero? (f x)) 1 0)))
```

Instead, we can keep a generalized type by using a **type variable**, allowing a good type inference system to derive (for this example, using type var T):

Type of f = $T \rightarrow \text{int}$

Type of x = T

Notice that this system *demands* we must be able to compare T for equality! This is actually *nontrivial* when we add polymorphism, but is simple in STLC (structural equality)

Constraint-Based Typing

The crucial trick to implementing type inference is to use a **constraint-based** approach. In this setting, we *walk over* each subterm in the program and generate a **constraint**

Unannotated lambdas generate new **type variables**, which are later constrained by their usages

Later, we will **solve** these constraints by using a process named **unification**

```

(define (build-constraints env e)
  (match e
    ;; Literals
    [(? integer? i) (cons `(:,i : int) (set))]
    [(? boolean? b) (cons `(:,b : bool) (set))]
    ;; Look up a type variable in an environment
    [(? symbol? x) (cons `(:,x : ,(hash-ref env x)) (set))]
    ;; Lambda w/o annotation
    [`(lambda (,x) ,e)
     ;; Generate a new type variable using gensym
     ;; gensym creates a unique symbol
     (define T1 (fresh-tyvar))
     (match (build-constraints (hash-set env x T1) e)
       [(cons `(:,e+ : ,T2) S)
        (cons `((lambda (,x : ,T1) ,e+) : (,T1 -> ,T2)) S)]]])
    ;; Application: constrain input matches, return output
    [`(,e1 ,e2)
     (match (build-constraints env e1)
       [(cons `(:,e1+ : ,T1) C1)
        (match (build-constraints env e2)
          [(cons `(:,e2+ : ,T2) C2)
           (define X (fresh-tyvar))
           (cons `(((,e1+ : ,T1) (,e2+ : ,T2)) : ,X)
                (set-union C1 C2 (set ` (= ,T1 (,T2 -> ,X))))))]])
       ;; Type stipulation against t--constrain
       [`(,e : ,t)
        (match (build-constraints env e)
          [(cons `(:,e+ : ,T) C)
           (define X (fresh-tyvar))
           (cons `((,e+ : ,T) : ,X) (set-add (set-add C ` (= ,X ,T)) ` (= ,X ,t)))]])
       ;; If: the guard must evaluate to bool, branches must be
       ;; of equal type.
       [`(if ,e1 ,e2 ,e3)
        (match-define (cons `(:,e1+ : ,T1) C1) (build-constraints env e1))
        (match-define (cons `(:,e2+ : ,T2) C2) (build-constraints env e2))
        (match-define (cons `(:,e3+ : ,T3) C3) (build-constraints env e3))
        (cons `((if (,e1+ : ,T1) (,e2+ : ,T2) (,e3+ : ,T3)) : ,T2)
              (set-union C1 C2 C3 (set ` (= ,T1 bool) ` (= ,T2 ,T3))))])

```

Building Constraints

Unification

At the end of constraint-building, we have a ton of equality constraints between base types and type variables

```
tv0 = int
ty1 = tv0 -> tv2
tv2 = tv3
tv3 = tv4
```

(lambda (x : ty1) ...)

In this example, what is `ty1`?

Answer: think about constraints and equalities: `ty1` must be `int->int`


```

;; within the constraint constr, substitute S for T
(define (ty-subst ty X T)
  (match ty
    [(? ty-var? Y) #:when (equal? X Y) T]
    [(? ty-var? Y) Y]
    ['bool 'bool]
    ['int 'int]
    [`(,T0 -> ,T1) `(,(ty-subst T0 X T) -> ,(ty-subst T1 X T))]))

(define (unify constraints)
  ;; Substitute into a constraint
  (define (constr-subst constr S T)
    (match constr
      [`(= ,C0 ,C1) `(= ,(ty-subst C0 S T) ,(ty-subst C1 S T))]))
  ;; Is t an arrow type?
  (define (arrow? t)
    (match t [`(, _ -> ,_) #t] [_ #f]))
  ;; Walk over constraints one at a time
  (define (for-each constraints)
    (match constraints
      ['() (hash)]
      [`((= ,S ,T) . ,rest)
       (cond [(equal? S T)
              (for-each rest)]
             [(and (ty-var? S) (not (set-member? (free-type-vars T) S)))
              (hash-set (unify (map (lambda (constr) (constr-subst constr S T)) rest)) S T)]
             [(and (ty-var? T) (not (set-member? (free-type-vars S) T)))
              (hash-set (unify (map (lambda (constr) (constr-subst constr T S)) rest)) T S)]
             [(and (arrow? S) (arrow? T))
              (match-define `(,S1 -> ,S2) S)
              (match-define `(,T1 -> ,T2) T)
              (unify (cons `(= ,S1 ,T1) (cons `(= ,S2 ,T2) rest)))]
             [else (error "type failure")])]))))

```

Unification

Why Type Theory?

Why is type synthesis / checking useful?

- Can write **fully-verified** programs.
 - Cons: type systems are esoteric, complicated, academic, etc...
 - Popular languages (Swift, Rust, etc...) *are tending towards more elaborate type systems as they evolve*
- Type synthesis offers me “proofs for free:”
 - “If my program type checks it works” — **not** true in C/C++/...
- Less **mental burden**, like CoPilot (etc... tools), type systems can integrate into IDEs to use synthesis information in guiding programming
 - In some ways, this reflects the logical statements underlying the type system’s design (Curry Howard)

“Proofs as Programs”

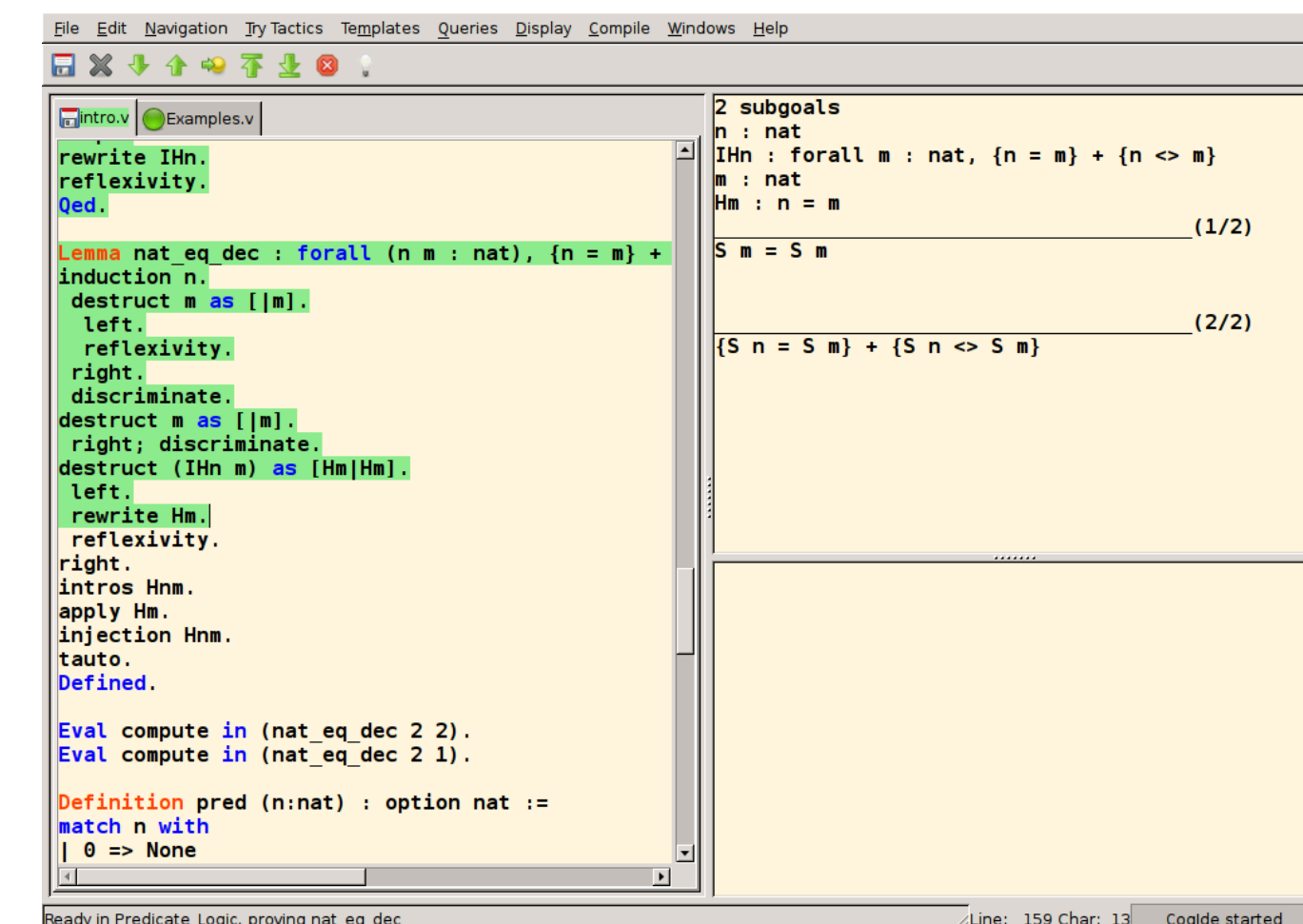
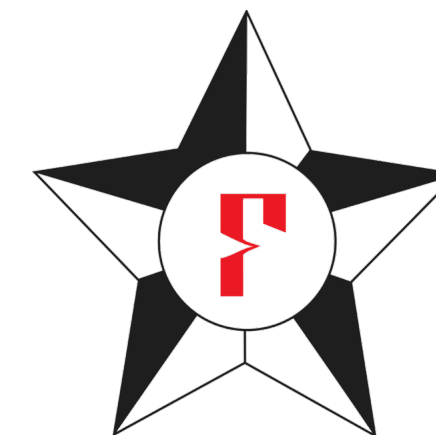
A significant amount of interest has been given to programming languages which use **powerful type systems** to write programs *alongside a proof of the program’s correctness*

Imagine how nice it would be to write a **completely-formally-verified** program—no bugs ever again!



LEVIN  Agda

```
"prove(M,I) :- append(Q,[C|R],M), \+member(-_,C),
  append(Q,R,S), prove([], [[-!|C]|S], [], I).
prove([],_,_,_).
prove([L|C],M,P,I) :- (-N=L; -L=N) -> (member(N,P);
  append(Q,[D|R],M), copy_term(D,E), append(A,[N|B],E),
  append(A,B,F), (D==E -> append(R,Q,S); length(P,K), K<I,
  append(R,[D|Q],S)), prove(F,S,[L|P],I)), prove(C,M,P,I)."
```



```
File Edit Navigation Try Tactics Templates Queries Display Compile Windows Help
Intro.v Examples.v
rewrite IHn.
reflexivity.
Qed.

Lemma nat_eq_dec : forall (n m : nat), {n = m} + {n <> m}
induction n.
destruct m as [|m].
left.
reflexivity.
right.
discriminate.
destruct m as [|m].
right; discriminate.
destruct (IHn m) as [Hm|Hm].
left.
rewrite Hm.
reflexivity.
right.
intros Hnm.
apply Hm.
injection Hnm.
tauto.
Defined.

Eval compute in (nat_eq_dec 2 2).
Eval compute in (nat_eq_dec 2 1).

Definition pred (n:nat) : option nat :=
match n with
| 0 => None
```

2 subgoals
n : nat
IHn : forall m : nat, {n = m} + {n <> m}
m : nat
Hm : n = m
S m = S m (1/2)
{S n = S m} + {S n <> S m} (2/2)

Ready in Predicate Logic, proving nat_eq_dec Line: 159 Char: 13 Coqide started

Dependent Type Systems

We can construct type systems / programming languages where terms can be of type (something like)

$$\forall (l : \text{list } A) : \{l' : \text{sorted } l' \wedge \forall (x : A). (\text{member } l \ x) \Rightarrow (\text{member } l' \ x)\}$$

These are called *dependent types*, because types can depend on *values*

- This allows expressing that l' is sorted
- Unfortunately, these type systems are way more complicated
- Worse, even type *checking* may be **undecidable** (in general)

Precise formalization of these systems is beyond the scope of this class

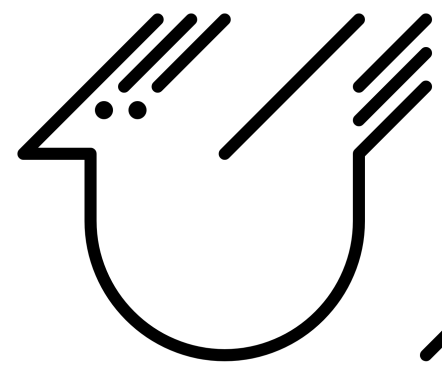
A huge family of languages have popped up to implement dependent type systems and subsequently enable “fully-verified” programming

They hit a variety of expressivity points. The fundamental trade off is: (a) expressivity vs. (b) automation.

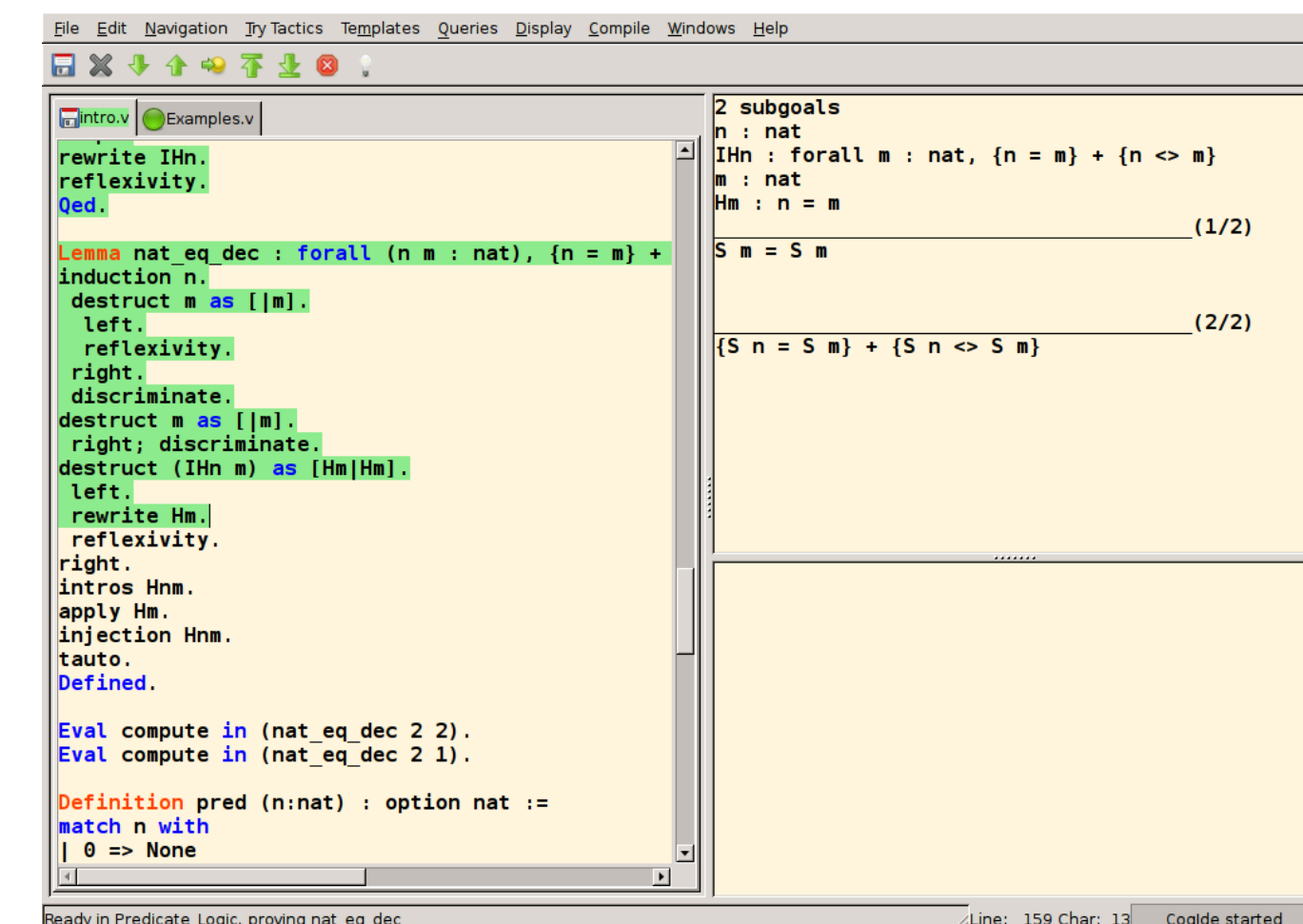
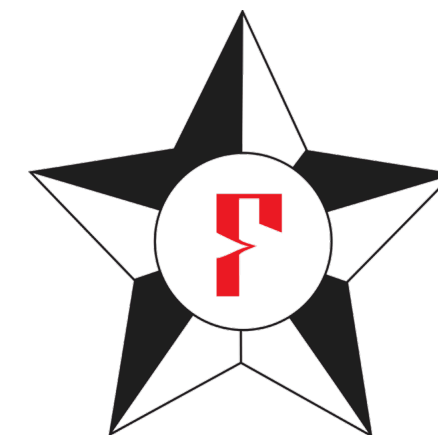
Highly-expressive systems require you to write all the proofs yourself, and a lot of manual annotation (potentially).



```
"prove(M,I) :- append(Q,[C|R],M), \+member(-,C),
  append(Q,R,S), prove([!],[[-!|C]|S],[],I).
prove([],_,_,_).
prove([L|C],M,P,I) :- (-N=L; -L=N) -> (member(N,P);
  append(Q,[D|R],M), copy_term(D,E), append(A,[N|B],E),
  append(A,B,F), (D==E -> append(R,Q,S); length(P,K), K<I,
  append(R,[D|Q],S)), prove(F,S,[L|P],I)), prove(C,M,P,I)."
```



Agda



Explicit Theorem Proving / Hole-Based Synth

Here I give an Agda definition for products

```
{- In Agda: for all P / Q, P -> Q -> P -}  
p_q_p : (P Q : Set) -> P -> Q -> P  
p_q_p P Q pf_P pf_Q = pf_P  
  
data _x_ (A : Set) (B : Set) : Set where  
  ( _,_ ) :  
    A  
    → B  
    ----  
    → A × B  
  
proj1 : ∀ {A B : Set}  
  → A × B  
  ----  
  → A  
proj1 ( x , x1 ) = x  
  
proj2 : ∀ {A B : Set}  
  → A × B  
  ----  
  → B  
proj2 ( x , x1 ) = x1
```

U:--- hello.agda 48% L36 <E> (Agda:Checked)

U:%*- *All Done* All L1 <M> (AgdaInfo)

Explicit Theorem Proving / Hole-Based Synth

```
p : (P Q : Set) -> P × (Q × P) -> Q
p P Q pf =  

{- proj1 (proj2 pf) -}
```

```
U:--- hello.agda Bot L57 <E> (Agda)
13 : Q [ at /home/guest/hello.agda:59,12-13 ]
```

```
U:%*- *All Goals* All L1 <M> (AgdaInfo)
```

Agda will tell me what I need to fill in,
allows me to use “holes” and then helps
me hunt for a working proof.

```
proj1 : ∀ {A B : Set}
       → A × B
       -----
       → A
proj1 ( x , x1 ) = x

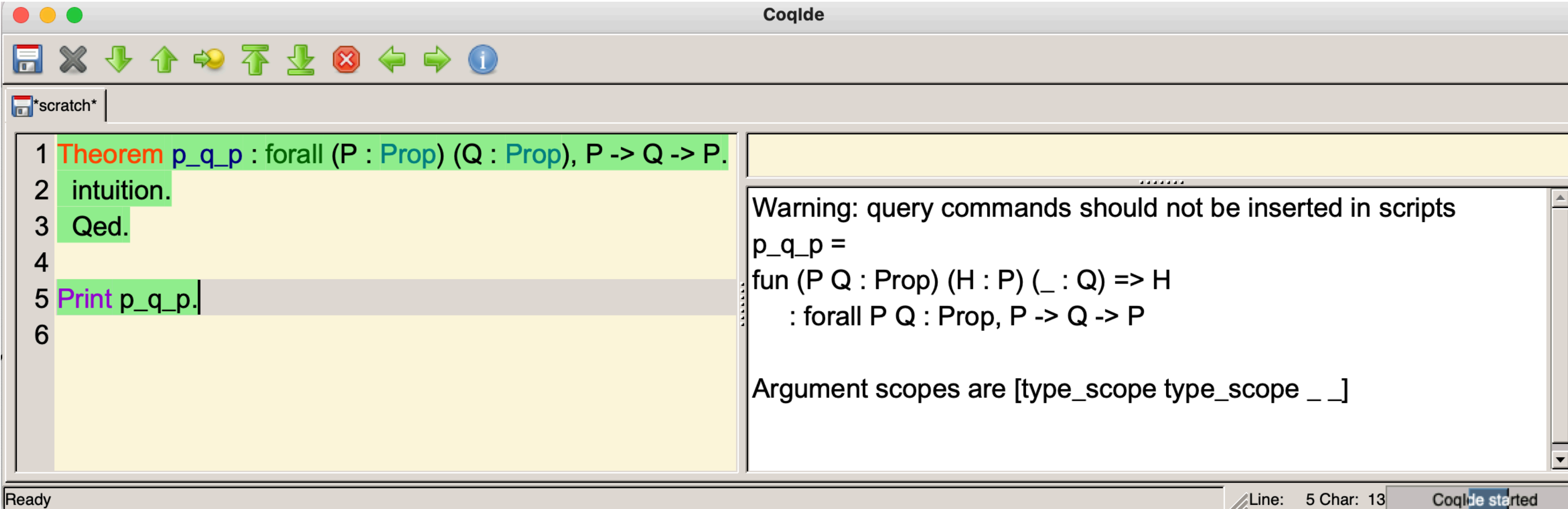
proj2 : ∀ {A B : Set}
       → A × B
       -----
       → B
proj2 ( x , x1 ) = x1
```

```
p : (P Q : Set) -> P × (Q × P) -> Q
p P Q pf = (proj1 (proj2 pf))
```

Tactic-Based Theorem Proving

Some systems provide logic-programming (i.e., *proof search*) to help assist users

- CHI tells us that proof search is tantamount to *program synthesis*
- Here I use Coq's “intuition” tactic to automatically construct a proof for me



The screenshot shows the CoqIDE window with a file named `*scratch*`. The editor contains the following Coq script:

```
1 Theorem p_q_p : forall (P : Prop) (Q : Prop), P -> Q -> P.  
2 intuition.  
3 Qed.  
4  
5 Print p_q_p.  
6
```

The right-hand pane displays the output of the `Print` command, showing the proof term for `p_q_p`:

```
Warning: query commands should not be inserted in scripts  
p_q_p =  
fun (P Q : Prop) (H : P) (_ : Q) => H  
  : forall P Q : Prop, P -> Q -> P  
  
Argument scopes are [type_scope type_scope _ _]
```

The status bar at the bottom indicates "Ready", "Line: 5 Char: 13", and "CoqIDE started".

(Using Coq to prove $P \Rightarrow Q \Rightarrow P$; left: using the “intuition” tactic, right: printing the proof term)

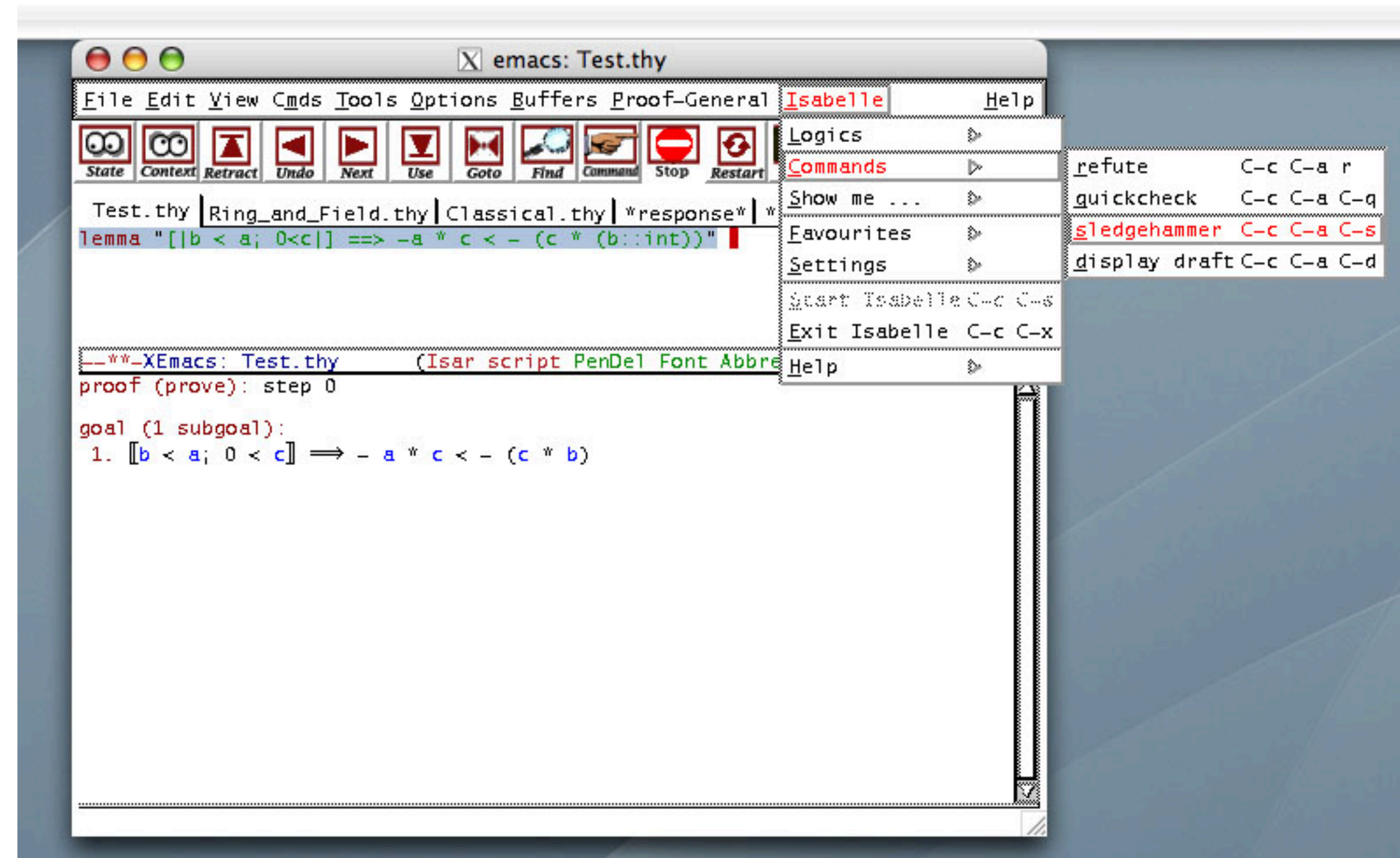
Automating proof via constraint solving

The more expressive the type theory, the more work is required to build proofs.

Some systems translate proof obligations into formulas which are then sent to SMT solvers (solves goals in first-order logic, such as Z3)

This can partially automate many otherwise-tricky proofs—in *certain* situations

F* based on this idea, but other proof search approaches exist (Idris, etc...)



How does this work?

These systems interpret **programs** as **theorems** in higher-order logics (calculus of constructions, etc...)

Unfortunately, no free lunch: this makes the type system *way* more complicated in practical settings

We will see a *taste* of the inspiration for these systems, by reflecting on STLC's expressivity

Valid Contexts.

$$\vdash * \qquad \frac{\Gamma \vdash \Delta}{\Gamma[x:\Delta] \vdash *} \qquad \frac{\Gamma \vdash P : *}{\Gamma[x:P] \vdash *}$$

Product Formation.

$$\frac{\Gamma[x:P] \vdash \Delta}{\Gamma \vdash [x:P]\Delta} \qquad \frac{\Gamma[x:P] \vdash N : *}{\Gamma \vdash [x:P]N : *}$$

Variables, Abstraction, and Application.

$$\frac{\Gamma \vdash *}{\Gamma \vdash x : P} [x:P] \text{ in } \Gamma \quad \frac{\Gamma[x:P] \vdash N : Q}{\Gamma \vdash (\lambda x:P) N : [x:P]Q} \quad \frac{\Gamma \vdash M : [x:P]Q \quad \Gamma \vdash N : P}{\Gamma \vdash (M N) : [N/x]Q}$$

$s, t, A, B ::= x$	variable
$(x : A) \rightarrow B$	dependent function type
$\lambda x. t$	lambda abstraction
$s t$	function application
$(x : A) \times B$	dependent pair type
$\langle s, t \rangle$	dependent pairs
$\pi_1 t \mid \pi_2 t$	projection
Set_i	universes ($i \in \{0..\}$)
1	the unit type
$\langle \rangle$	the element of the unit type
$\Gamma, \Delta ::= \varepsilon$	
$(x : A)\Gamma$	telescopes

What to Know for Midterm 2 on Types

- Know how to read the typing rules we presented throughout this lecture.
- Know how to check that a typing derivation presented is correct, or be able to point out where it is broken.
- Know how to build a typing derivation (i.e., proof tree, the things with the lines and stacked formulas) for small programs using the rules
- Understand the definition of the term “soundness” as it applies to type systems
 - If a PL’s type system is sound, are any dynamic errors possible?