Type Systems

CIS352 — Fall 2023

Kris Micinski

- Last week: Intuitionistic Propositional Logic (IPL) and natural deduction, in which we define inference rules (schemas)
- Whole "proofs" are built by chaining together inference rules

- This week, we will build static type systems for PLs
  - These type systems <u>rule out</u> programs containing possible type errors
  - No well-typed program will crash due to a runtime type error

- These type systems have a close relationship to constructive logics:
  - ***Curry-Howard Isomorphism***: well-typed programs correspond to valid proofs of theorems in constructive logic

Racket's ***contract system*** tracks runtime type errors—the problem is that contract checking adds lots of overhead

"I take in a positive and produce a positive."

```
(define/contract (fib x)
  (-> positive? positive?)
  (cond
    [(= x 0) 1]
    [(= x 1) 1]
    [else (+ (fib (- x 1)) (fib (- x 2)))]))
```

```
Welcome to DrRacket, version 7.2 [3m].
Language: racket, with debugging; memory limit: 128 MB.
> (fib 2)
2
>
```

When I mess up

```
(define/contract (fib x)
  (-> positive? positive?)
  (cond
    [(= x 0) 1]
    [(= x 1) 1]
    [else (+ (fib (- x 1)) (fib (- x 2)))]))
```

```
> (fib -2)
fib: contract violation
expected: positive?
given: -2
in: the 1st argument of
    (-> positive? positive?)
contract from: (function fib)
blaming: anonymous-module
 (assuming the contract is correct)
at: unsaved-editor:3.18
>
```

When **I** mess up

```
(define/contract (fib x)
  (-> positive? positive?)
  (cond
    [(= x 0) 1]
    [(= x 1) 1]
    [else (+ (fib (- x 1)) (fib (- x 2)))]))
```

```
> (fib -2)
fib: contract violation
  expected: positive?
  given: -2
  in: the 1st argument of
      (-> positive? positive?)
  contract from: (function fib)
  blaming: anonymous-module
   (assuming the contract is correct)
  at: unsaved-editor:3.18
>
```

Racket blames **me**
(anonymous-module)

When **fib** messes up

```
(define/contract (fib x)
  (-> positive? positive?)
  (cond
    [(= x 0) -200]
    [(= x 1) 1]
    [else (+ (fib (- x 1)) (fib (- x 2)))]))
```

```
Welcome to DrRacket, version 7.2 [3m].
Language: racket, with debugging; memory limit: 128 MB.
> (fib 20)
fib: broke its own contract
 promised: positive?
 produced: -829435
 in: the range of
      (-> positive? positive?)
 contract from: (function fib)
 blaming: (function fib)
  (assuming the contract is correct)
 at: unsaved-editor:3.18
>
```

Racket blames **fib**

Earlier...

Note that contracts are checked at **runtime**

(**Not** compile time!)

But sometimes we want to know our program won't break **before** it runs!

Type Systems

A **type system** assigns each source fragment with a given **type**: a specification of how it will behave

Type systems include **rules**, or **judgements** that tells us how we compositionally build types for larger fragments from smaller fragments

The **goal** of a type system is to **rule out** programs that would exhibit run time type errors!

Simply-Typed λ-calculus

STLC is a restriction of the untyped λ-calculus
(It is a restriction in the sense that not all terms are well-typed.)

Expressions in STLC, assuming t is a type (we'll show this soon):

```
e ::= (lambda (x : t) e)
    | (e e)
    | (prim e e)
    | x
    | n
    | (e : t)
```

All lambdas ***must*** be annotated with their type

Optionally, any subexpression may be ***annotated*** with a type

```
prim ::= + | * | …
```

```
;; Expressions are ifarith, with several special builtins
(define (expr? e)
  (match e
    ;; Variables
    [(? symbol? x) #t]
    ;; Literals
    [(? bool-lit? b) #t]
    [(? int-lit? i) #t]
    ;; Applications
    [`(,(? expr? e0) ,(? expr? e1)) #t]
    ;; Annotated expressions
    [`(,(? expr? e) : ,(? type? t)) #t]
    ;; Anotated lambdas
    [`(lambda (,(? symbol? x) : ,(? type? t)) ,(? expr? e)) #t]))
```

The ***simply typed*** lambda calculus is a type system built on top of a small kernel of the lambda calculus

Crucially, STLC is *less expressive* than the lambda calculus (e.g., we cannot type Ω, Y, or U!)

In practice, STLC's restrictions make it unsuitable for serious programming—but it is the basis for many modern type systems in real languages (e.g., OCaml, Rust, Swift, Haskell, …)

Terms ***inhabit*** types
(via the typing judgement)

Term Syntax

```
e ::= (lambda (x : t) e)
    | (e e)
    | (prim e e)
    | x
    | n
    | (e : t)

prim ::= + | * | …
```

Type Syntax

```
t ::= num
    | bool
    | t -> t
```

Term Syntax

```
e ::= (lambda (x : t) e)
    | (e e)
    | (prim e e)
    | x
    | n
    | (e : t)

prim ::= + | * | …
```

Type Syntax

```
t ::= num
    | bool
    | t -> t
```

Function Types

Term Syntax

```
e ::= (lambda (x : t) e)
    | (e e)
    | (prim e e)
    | x
    | n
    | (e : t)

prim ::= + | * | …
```

Type Syntax

```
t ::= num
    | bool
    | t -> t
```

Examples…

```
bool -> num
num -> (num -> num)
num -> num
(num -> num) -> num
(bool -> (num -> bool)) -> num
```

- Type checking happens hierarchically (just as proofs in IPL are tree-shaped)
- Literals (0, #f) have their obvious types (these are the “axiom” cases)
- More complex forms (lambda, apply) require us to type subexpressions

For example, let’s say we have this lambda, which we want to type check:

$$\left(\lambda(x : \text{num})\ \left(\text{if } (x = 0)\ x\ (+\ x\ 1)\right)\right)$$

First we see the input type is num. Assuming x is num, we type check the body (an if). We see both sides of the if result in a number, so we know the lambda’s output is also a number.

Thus, the type is `num -> num`

Notice that in STLC, all lambdas ***must*** bind their argument by naming a type explicitly. Thus, the following is ***not*** an STLC term.

However, the term has an infinite number of possible types:

$$\left(\lambda(x)\ \left(\text{if \#f}\ (x\ 5)\ (x\ 8)\right)\right)$$

The term may be ***monomorphized*** by instantiating once for each type T such that T is something like…

$$\left(\lambda(x : T_0 \rightarrow T_1)\ \left(\text{if \#f}\ (x\ 5)\ (x\ 8)\right)\right)$$

Question: why $T_0 \rightarrow T_1$ rather than any type T? ***Answer***: x is applied (must be function)

Exercise: Write three possible monomorphizations, what is the type of the lambda as a whole?

The fact that lambdas must be annotated with a type makes typing easy: parameters are the only true source of non-local control in the lambda calculus, and represent the only ambiguity in type checking

$$\left(\lambda(x : \mathsf{num} \to \mathsf{num})\ \left(\mathsf{if}\ \#\mathsf{f}\ (x\ 5)\ (x\ 8)\right)\right)$$

One possible monomorphization

Bad thought experiment

$$\left(\text{if \#f } (x\ 5)\ (x\ 8)\right)$$

Let's say x is the Racket lambda:

```
(λ (x) (if (< x 6) #t 5))
```

Now, when x is less than 6, we return something of type `bool`; but otherwise, we return something of type `num`.

$$\Big(+\ \ 3\ \left(\text{if \#f } (x\ 5)\ (x\ 8)\right)\Big)$$

In this case, the + operation works as long as `(x 8)` returns a `num`, but what if `(x 8)` returns a bool?

A few examples…

$$\big(\lambda(x : \text{num})\ (\lambda\ (y : \text{bool})\ y)\big) : \text{num} \to \text{bool} \to \text{bool}$$

$$\Big(\lambda(x : \text{num} \to \text{num})\ (x\ 5)) : (\text{num} \to \text{num}) \to \text{num}$$

$$\Big(\lambda(x : \text{num} \to \text{num})\ \big(\text{if \#f}\ (x\ 5)\ (x\ 8)\big)\Big) : (\text{num} \to \text{num}) \to \text{num}$$

A type system for STLC

Type rules are written in natural-deduction style
(Like IPL, big-step semantics, etc…)

Assumptions above the line

(No assumptions here.)

$$\frac{}{\Gamma \vdash n : \textbf{num}}\ \textbf{Const}$$

Typing environment
(Irrelevant for now…)

Conclusions below the line

The rule reads “in any typing environment Γ, we may conclude the literal number n has type num”

$$\frac{}{\Gamma \vdash n : \textbf{num}}\ \textbf{Const}$$

Variable Lookup

We assume a **typing environment** which maps variables to their types

$$\frac{\Gamma(x) = t}{\Gamma \vdash x : t} \textbf{ Var}$$

If x maps to type t in Γ, we may conclude that x has type t under the type environment Γ

Exercise: using the **Var** rule, complete the proof

$$\frac{}{\{x \mapsto (\textbf{num} \to \textbf{num}), y \mapsto \textbf{bool}\} \vdash x : ???}$$

$$\frac{\Gamma(x) = t}{\Gamma \vdash x : t}\ \textbf{Var}$$

Solution

$$\frac{\{x \mapsto (\textbf{num} \to \textbf{num}), y \mapsto \textbf{bool}\}(x) = \textbf{num} \to \textbf{num}}{\{x \mapsto (\textbf{num} \to \textbf{num}), y \mapsto \textbf{bool}\} \vdash x : (\textbf{num} \to \textbf{num})}\ \textbf{Var}$$

$$\frac{\Gamma(x) = t}{\Gamma \vdash x : t}\ \textbf{Var}$$

Typing Functions

If, assuming x has type x, you can conclude the body e has type t', then the whole lambda has type t → t'

$$\frac{\Gamma[x \mapsto t] \vdash e : t'}{\Gamma \vdash (\lambda\,(x : t)\;\, e) : t \to t'} \quad \textbf{Lam}$$

If, assuming x has type x, you can conclude the body e has type t’, then the whole lambda has type t → t’

$$\frac{\Gamma[x \mapsto t] \vdash e : t'}{\Gamma \vdash (\lambda\,(x : t)\;e) : t \to t'}\ \textbf{Lam}$$

Notice: if we didn’t have type t here, we would have to ***guess***, which could be quite hard. We will have to do this when we move to allow *type inference*

$$\frac{\Gamma[x \mapsto t] \vdash e : t'}{\Gamma \vdash (\lambda\,(x : t)\ e) : t \to t'}\ \textbf{Lam}$$

Example: let's use the Lam rule to ascertain the type of the following expression.

$$\frac{}{\texttt{(lambda (x : num) 1)}}$$

$$\frac{\Gamma[x \mapsto t] \vdash e : t'}{\Gamma \vdash (\lambda\,(x : t)\; e) : t \to t'}\ \textbf{Lam}$$

Start with the empty environment (since this term is closed)

$$\Gamma = \{\} \vdash \frac{}{\texttt{(lambda (x : num) 1)}} : ? \to ?$$

$$\frac{\Gamma[x \mapsto t] \vdash e : t'}{\Gamma \vdash (\lambda\,(x : t)\ e) : t \to t'}\ \textbf{Lam}$$

$$\Gamma = \{\} \vdash \overline{\texttt{(lambda (x : num) 1)}} : t \to t'$$

We **suppose** there are two types, t and t’, which will make the derivation work.

Because x is tagged, it must be **num**

$$\frac{\{x \mapsto \textbf{num}\} \vdash 1 : t'}{\Gamma = \{\} \vdash \texttt{(lambda (x : num) 1)} : \textbf{num} \rightarrow t'}$$

We **suppose** there are two types, t and t', which will make the derivation work.

The **Const** rule allows us to conclude 1 : **num**

$$\dfrac{\dfrac{}{\{x \mapsto \textbf{num}\} \vdash 1 : t'}}{\Gamma = \{\} \vdash \texttt{(lambda (x : num) 1)} : \textbf{num} \to t'}\ \textbf{Lam}$$

We **suppose** there are two types, t and t', which will make the derivation work.

So t’ = **num**

Notice: **Const** demands no subgoals

$$\dfrac{}{\{x \mapsto \mathbf{num}\} \vdash 1 : \mathbf{num}}\ \textbf{Const}$$

$$\dfrac{\{x \mapsto \mathbf{num}\} \vdash 1 : \mathbf{num}}{\Gamma = \{\} \vdash \texttt{(lambda (x : num) 1)} : \mathbf{num} \rightarrow \mathbf{num}}\ \textbf{Lam}$$

Function Application

$$\frac{\Gamma \vdash e : t \to t' \qquad \Gamma \vdash e' : t}{\Gamma \vdash (e\ e') : t'}\ \textbf{App}$$

Function Application

If (under Gamma), e has type `t -> t’`

And `e’` (its argument) has type t

$$\frac{\Gamma \vdash e : t \to t' \qquad \Gamma \vdash e' : t}{\Gamma \vdash (e\ e') : t'}\ \textbf{App}$$

Then the application of e to `e’` results in a `t’`

Our type system so far…

$$\frac{}{\Gamma \vdash n : \textbf{num}}\ \textbf{Const}$$

(Also **False**)

$$\frac{}{\Gamma \vdash \textbf{\#t} : \textbf{bool}}\ \textbf{True}$$

$$\frac{\Gamma(x) = t}{\Gamma \vdash x : t}\ \textbf{Var}$$

$$\frac{\Gamma \vdash e : t \rightarrow t' \qquad \Gamma \vdash e' : t}{\Gamma \vdash (e\ e') : t'}\ \textbf{App}$$

$$\frac{\Gamma[x \mapsto t] \vdash e : t'}{\Gamma \vdash (\lambda\,(x : t)\ e) : t \rightarrow t'}\ \textbf{Lam}$$

Almost everything! What about builtins?

- Almost everything! What about builtins?
- A few ways to handle this:
  - Add ***pairs*** to our language
    - Builtins accept pairs

$$\Gamma_\iota = \{ + : (\mathbf{num} \times \mathbf{num}) \to \mathbf{num}, \ldots \}$$

- Or, we could assume that primitives are simply curried—in that case we would have, e.g., ((+ 1) 2) and then…

$$\Gamma_\iota = \{ + : \mathbf{num} \to (\mathbf{num} \to \mathbf{num}), \ldots \}$$

- ***Our exercise does this!!***

Two possibilities (pairs/currying)

```
e ::= (lambda (x : t) e)
    | (e e)
    | (prim (e, e))  ; pairs
    | ((prim e) e)   ; curry
    | x
    | n
    | (e : t)

prim ::= + | * | ...
```

Practice Derivations

Write derivations of the following expressions…

```
((λ (x : num) x) 1)
```

$$\frac{}{\Gamma \vdash n : \textbf{num}} \textbf{Const} \qquad \frac{x \mapsto t \in \Gamma}{\Gamma \vdash x : t} \textbf{Var}$$

$$\frac{\Gamma \vdash e : t \rightarrow t' \quad \Gamma \vdash e' : t}{\Gamma \vdash (e\ e') : t'} \textbf{App}$$

$$\frac{\Gamma, \{x \mapsto t\} \vdash e : t'}{\Gamma \vdash (\lambda\,(x : t)\ e) : t \rightarrow t'} \textbf{Lam}$$

```
((λ (x : int) x) 1)
```

$$
\textbf{App}\;\dfrac{\textbf{Lam}\;\dfrac{\textbf{Var}\;\dfrac{}{\{x \mapsto \textbf{num}\} \vdash x : \textbf{num}}}{\{\,\} \vdash (\lambda\,(x : \textbf{num})\ x) : \textbf{num} \rightarrow \textbf{num}} \qquad \dfrac{}{\{\,\} \vdash 1 : \textbf{num}}\;\textbf{Const}}{\{\,\} \vdash \big((\lambda\,(x : \textbf{num})\ x)\ 1\big) : \textbf{num}}
$$

```
((λ (f : num -> num) (f 1)) (λ (x : num) x))
```

$$\frac{}{\Gamma \vdash n : \textbf{num}} \quad \textbf{Const}$$

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

$$\frac{\Gamma \vdash e : t \to t' \qquad \Gamma \vdash e' : t}{\Gamma \vdash (e\ e') : t'} \quad \textbf{App}$$

$$\frac{\Gamma, \{x \mapsto t\} \vdash e : t'}{\Gamma \vdash (\lambda\,(x : t)\ e) : t \to t'} \quad \textbf{Lam}$$

Typability in STLC

Not all terms can be given types…

```
(λ (f : num -> num) (f f))
```

It is impossible to write a derivation for the above term!

f is `num->num` but would **need** to be `num`!

Typability

Not all terms can be given types…

```
((λ (f) (f f))
 (λ (f) (f f)))
```

It is **impossible** to write a derivation for Ω!

Consider what would happen if `f` were:

- `num -> num`
- `(num -> num) -> num`

Always just out of reach…

Type **Checking**

Type checking asks: given this fully-typed term, is the type checking done correctly?

```
((λ (x:num) x:num) : num -> num)
```

In practice, as long as we annotate arguments (of λs) with specific types, we can elide the types of variables, literals, and applications

The "simply typed" nature of STLC means that type inference is very simple…

Exercise

For each of the following expressions, do they type check?
I.e., is it possible to construct a typing derivation for them?
If so, what is the type of the expression?

```
(λ (f : num -> num -> num) (((f 2) 3) 4))
```

```
((λ (f : num -> num) f) (λ (x:num) (λ (x:num) x)))
```

Solution

Neither *type checks.*

This subexpression results in **num**, which cannot be applied.

```
(λ (f : num -> num -> num) (((f 2) 3) 4))
```

```
((λ (f : num -> num) f) (λ (x:num) (λ (x:num) x)))
```

Solution

Neither *type checks.*

```
(λ (f : num -> num -> num) (((f 2) 3) 4))

((λ (f : num -> num) f) (λ (x:num) (λ (x:num) x)))
```

This binder ***demands*** its argument is of type `num -> num`,
but its argument is ***really*** of type `num -> num -> num`

In the case of fully-annotated STLC, we never have to *guess* a type

In STLC, type *inference* is no harder than type *checking*

Our type checker will be **syntax-directed**

Next lecture, we will look at type inference for **un-annotated** STLC

- This will require generating, and then solving, constraints

The basic approach is to observe that each of the rules applies to a different *form*

$$\frac{}{\Gamma \vdash n : \textbf{num}}\ \textbf{Const} \qquad \frac{\Gamma(x) = t}{\Gamma \vdash x : t}\ \textbf{Var}$$

For example, if we hit *any* application expression (e e'), we know that we *have* to use the **App** rule

$$\frac{\Gamma \vdash e : t \to t' \quad \Gamma \vdash e' : t}{\Gamma \vdash (e\ e') : t'}\ \textbf{App}$$

Thus, we write our type checker as a structurally-recursive function over the input expression.

$$\frac{\Gamma[x \mapsto t] \vdash e : t'}{\Gamma \vdash (\lambda\, (x : t)\ e) : t \to t'}\ \textbf{Lam}$$

```
;; Synthesize a type for e in the environment env
;; Returns a type
(define (synthesize-type env e)
  (match e
    ;; Literals
    [(? integer? i) 'int]
    [(? boolean? b) 'bool]
```

$$\frac{}{\Gamma \vdash n : \textbf{num}}\ \textbf{Const}$$

Recognizing literals is easy

```
;; Synthesize a type for e in the environment env
;; Returns a type
(define (synthesize-type env e)
  (match e
    ;; Literals
    [(? integer? i) 'int]
    [(? boolean? b) 'bool]
    ;; Look up a type variable in an environment
    [(? symbol? x) (hash-ref env x)]
```

$$\frac{\Gamma(x) = t}{\Gamma \vdash x : t} \quad \textbf{Var}$$

```
;; Synthesize a type for e in the environment env
;; Returns a type
(define (synthesize-type env e)
  (match e
    ;; Literals
    [(? integer? i) 'int]
    [(? boolean? b) 'bool]
    ;; Look up a type variable in an environment
    [(? symbol? x) (hash-ref env x)]
    ;; Lambda w/ annotation
    [`(lambda (,x : ,A) ,e)
     `(,A -> ,(synthesize-type (hash-set env x A) e))]
```

$$\frac{\Gamma[x \mapsto t] \vdash e : t'}{\Gamma \vdash (\lambda\,(x : t)\ e) : t \to t'}\ \textbf{Lam}$$

```
;; Synthesize a type for e in the environment env
;; Returns a type
(define (synthesize-type env e)
  (match e
    ;; Literals
    [(? integer? i) 'int]
    [(? boolean? b) 'bool]
    ;; Look up a type variable in an environment
    [(? symbol? x) (hash-ref env x)]
    ;; Lambda w/ annotation
    [`(lambda (,x : ,A) ,e)
     `(,A -> ,(synthesize-type (hash-set env x A) e))]
    ;; Arbitrary expression
    [`(,e : ,t) (let ([e-t (synthesize-type env e)])
                  (if (equal? e-t t)
                    t
                    (error (format "types ~a and ~a are different" e-t t))))]
```

We haven't written this rule yet—but notice how the t's are implicitly unified (i.e., asserted to be the same) in the rule

$$\frac{\Gamma \vdash e : t}{\Gamma \vdash (e : t) : t} \textbf{ Chk}$$

```
;; Synthesize a type for e in the environment env
;; Returns a type
(define (synthesize-type env e)
  (match e
    ;; Literals
    [(? integer? i) 'int]
    [(? boolean? b) 'bool]
    ;; Look up a type variable in an environment
    [(? symbol? x) (hash-ref env x)]
    ;; Lambda w/ annotation
    [`(lambda (,x : ,A) ,e)
     `(,A -> ,(synthesize-type (hash-set env x A) e))]
    ;; Arbitrary expression
    [`(,e : ,t) (let ([e-t (synthesize-type env e)])
                  (if (equal? e-t t)
                    t
                    (error (format "types ~a and ~a are different" e-t t))))]
    ;; Application
    [`(,e1 ,e2)
     (match (synthesize-type env e1)
       [`(,A -> ,B)
        (let ([t-2 (synthesize-type env e2)])
          (if (equal? t-2 A)
            B
            (error (format "types ~a and ~a are different" A t-2))))])]))
```

$$\frac{\Gamma \vdash e : t \to t' \quad \Gamma \vdash e' : t}{\Gamma \vdash (e\ e') : t'}\ \textbf{App}$$

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 ~= 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

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

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

$$\Gamma : \textbf{Var} \to \textbf{Type}$$

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

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

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

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

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

The **App** rule corresponds to modus ponens in IPL

Notice how the type is A → B but in IPL it is A ⇒ B

The **Lam** rule introduces assumptions, just as **⇒I** does in IPL

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

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

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

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

$$\frac{\Gamma, \{x \mapsto t\} \vdash e : A}{\Gamma \vdash (\lambda\,(x : t)\ e) : A \to B} \ \textbf{Lam}$$

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

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 begs a question: we have covered ***this*** (in STLC) so far, what about ***the rest***

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

∧E1 $$\frac{\Gamma \vdash P \wedge Q}{\Gamma \vdash P}$$

∧E2 $$\frac{\Gamma \vdash P \wedge Q}{\Gamma \vdash Q}$$

∧I $$\frac{\Gamma \vdash P \quad \Gamma \vdash Q}{\Gamma \vdash P \wedge Q}$$

∨I1 $$\frac{\Gamma \vdash P}{\Gamma \vdash P \vee Q}$$

∨I2 $$\frac{\Gamma \vdash Q}{\Gamma \vdash P \vee Q}$$

∨E $$\frac{\Gamma \vdash A \vee B \qquad \Gamma, A \vdash C \qquad \Gamma, B \vdash C}{\Gamma \vdash C}$$

$$\frac{\Gamma \vdash A \Rightarrow B \qquad \Gamma \vdash A}{\Gamma \vdash B}$$ **⇒E**

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

$$\frac{\Gamma \vdash \bot}{\Gamma \vdash P}$$ **⊥E**

¬P is sugar for P ⇒ ⊥

This is an *exciting* question because it asks: what is the computational interpretation of ∧, ∨, and ⊥

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

$$\wedge\textbf{E1}\ \frac{\Gamma \vdash P \wedge Q}{\Gamma \vdash P} \qquad \wedge\textbf{E2}\ \frac{\Gamma \vdash P \wedge Q}{\Gamma \vdash Q} \qquad \wedge\textbf{I}\ \frac{\Gamma \vdash P \quad \Gamma \vdash Q}{\Gamma \vdash P \wedge Q}$$

$$\vee\textbf{I1}\ \frac{\Gamma \vdash P}{\Gamma \vdash P \vee Q} \qquad \vee\textbf{I2}\ \frac{\Gamma \vdash Q}{\Gamma \vdash P \vee Q} \qquad \vee\textbf{E}\ \frac{\Gamma \vdash A \vee B \quad \Gamma, A \vdash C \quad \Gamma, B \vdash C}{\Gamma \vdash C}$$

$$\frac{\Gamma \vdash A \Rightarrow B \quad \Gamma \vdash A}{\Gamma \vdash B}\ \Rightarrow\textbf{E} \qquad \frac{\Gamma, A \vdash B}{\Gamma \vdash A \Rightarrow B}\ \Rightarrow\textbf{I}$$

$$\frac{\Gamma \vdash \bot}{\Gamma \vdash P}\ \bot\textbf{E}$$

¬P is sugar for P ⇒ ⊥

Let's just start with $\wedge$, we need to come up with type-theoretic analogues for these rules

$$\wedge\textbf{E1}\quad \frac{\Gamma \vdash P \wedge Q}{\Gamma \vdash P} \qquad \wedge\textbf{E2}\quad \frac{\Gamma \vdash P \wedge Q}{\Gamma \vdash Q} \qquad \wedge\textbf{I}\quad \frac{\Gamma \vdash P \quad \Gamma \vdash Q}{\Gamma \vdash P \wedge Q}$$

```
e ::= (lambda (x : t) e)
    | (e e)
    | …
    | (cons e e) ;; ∧
    | (car e) | (cdr e)
```

```
t ::= num | bool | …
    | t × t ;; product types
```

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

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

```
(cons 5 #t) : num × bool
```

Now, we define the type rules for product (×) types
CHI tells us the rules should look like the yellow ones

$$\wedge\textbf{E1}\quad \frac{\Gamma \vdash P \wedge Q}{\Gamma \vdash P} \qquad \wedge\textbf{E2}\quad \frac{\Gamma \vdash P \wedge Q}{\Gamma \vdash Q} \qquad \wedge\textbf{I}\quad \frac{\Gamma \vdash P \quad \Gamma \vdash Q}{\Gamma \vdash P \wedge Q}$$

"If e is a pair, (car/cdr e) is the type of its first/second element"

"If e_0 is type A and e_1 is type B, (cons e_0 e_1) is type A × B"

$$\times\textbf{E1}\quad \frac{\Gamma \vdash e : A \times B}{\Gamma \vdash (\text{car } e) : A} \qquad \times\textbf{E2}\quad \frac{\Gamma \vdash e : A \times B}{\Gamma \vdash (\text{cdr } e) : B} \qquad \times\textbf{I}\quad \frac{\Gamma \vdash e_0 : A \quad \Gamma \vdash e_1 : B}{\Gamma \vdash (\text{cons } e_0\ e_1) : A \times B}$$

Next, let's move to ∨

$$\vee\mathbf{I1}\quad \frac{\Gamma \vdash P}{\Gamma \vdash P \vee Q} \qquad \vee\mathbf{I2}\quad \frac{\Gamma \vdash Q}{\Gamma \vdash P \vee Q} \qquad \vee\mathbf{E}\quad \frac{\Gamma \vdash A \vee B \qquad \Gamma, A \vdash C \qquad \Gamma, B \vdash C}{\Gamma \vdash C}$$

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

The computational interpretation of ∨ is a *discriminated union*

```
t ::= … | t + t
```

Now we have ***sum*** types

```
(inj_left 42) : num × bool
```

Also many other types

```
(inj_left 42) : num × num
(inj_left 42) : num × (num -> nu
(inj_left 42) : num × (num × num
```

…

A discriminated union A + 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 ∨ 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
```

Now, we define the type rules for product (×) types
CHI tells us the rules should look like the yellow ones

$$\vee\mathbf{I1}\quad \frac{\Gamma \vdash P}{\Gamma \vdash P \vee Q} \qquad \vee\mathbf{I2}\quad \frac{\Gamma \vdash Q}{\Gamma \vdash P \vee Q}$$

"Using e, we can witness either the left or right choice."

$$\mathbf{+I1}\ \frac{\Gamma \vdash e : A}{\Gamma \vdash (\text{left } e) : A + B} \qquad \mathbf{+I2}\ \frac{\Gamma \vdash e : B}{\Gamma \vdash (\text{right } e) : A + B}$$

The elimination rule for ∨ is interesting; we are obligated to prove two subgoals: (a) assuming A, prove C, and (b) assuming B, prove C

$$\vee\mathbf{E}\quad \frac{\Gamma \vdash A \vee B \qquad \Gamma, A \vdash C \qquad \Gamma, B \vdash C}{\Gamma \vdash C}$$

In our setting, we recognize ∨ as +, and thus A ∨ B is a discriminated union, i.e., a value of a type either A or B—but we can only know which by matching

The two subgoals are functions (callbacks) which observe a value of type A or B

$$\vee\mathbf{E}\ \frac{\Gamma \vdash A \vee B \qquad \Gamma, A \vdash C \qquad \Gamma, B \vdash C}{\Gamma \vdash C}$$

A handler that, assuming e_1 is of type B, builds an object of type C

A value of either type A or type B

A handler that, assuming e_0 is of type A, builds an object of type C

$$\mathbf{+E}\ \frac{\Gamma \vdash e : A + B \quad \Gamma, e_0 : A \vdash e_0' : C \quad \Gamma, e_1 : A \vdash e_1' : C}{\Gamma \vdash \left(\text{case } e \text{ of } (\text{left } e_0 \Rightarrow e_0') \ (\text{right } e_1 \Rightarrow e_1')\right) : C}$$

$$\vee\mathbf{E}\ \frac{\Gamma \vdash A \vee B \qquad \Gamma, A \vdash C \qquad \Gamma, B \vdash C}{\Gamma \vdash C}$$

A handler that, assuming e_1 is of type B, builds an object of type C

A value of either type A or type B

A handler that, assuming e_0 is of type A, builds an object of type C

$$+\mathbf{E}\ \frac{\Gamma \vdash e : A + B \qquad \Gamma, e_0 : A \vdash e_0' : C \qquad \Gamma, e_1 : A \vdash e_1' : C}{\Gamma \vdash \left(\text{case } e \text{ of } (\text{left } e_0 \Rightarrow e_0')\ (\text{right } e_1 \Rightarrow e_1')\right) : C}$$

Notice that the handlers must produce the same type!

The constructive notion of negation says two things:

- You're never allowed to construct a proof of false:
  - thus, $\bot$ has no introduction rules
- If you can prove $\bot$ using what you currently know, then you must be in a contradiction, and you can freely admit anything.
  - Like a lucid dream

$$\frac{\Gamma \vdash \bot}{\Gamma \vdash P}\ \bot\mathbf{E}$$

Now, we need to ask: what's the ***computational*** interpretation of $\bot$?

The constructive notion of negation says two things:

- You're never allowed to construct a proof of false:
  - thus, $\perp$ has no introduction rules
- If you can prove $\perp$ using what you currently know, then you must be in a contradiction, and you can freely admit anything.
  - Like a lucid dream

$$\frac{\Gamma \vdash \perp}{\Gamma \vdash P}\;\perp\mathbf{E}$$

Now, we need to ask: what's the ***computational*** interpretation of $\perp$?

First: there is no rule to introduce $\perp$. Second, if there is some expression which we can type which is $\perp$, we know we are in a contradiction and are allowed to materialize a value of any type we please.

Any type we want!

$$\frac{\Gamma \vdash e : \perp}{\Gamma \vdash (\text{case } e \text{ of}) : t}\;\perp\mathbf{E}$$

Empty match statement—because no value!

Vanilla STLC

$$\frac{\Gamma \vdash e : t \to t' \quad \Gamma \vdash e' : t}{\Gamma \vdash (e\ e') : t'}\ \textbf{App} \qquad \frac{}{\Gamma \vdash n : \textbf{num}}\ \textbf{Const} \qquad \frac{x \mapsto t \in \Gamma}{\Gamma \vdash x : t}\ \textbf{Var} \qquad \frac{\Gamma, \{x \mapsto t\} \vdash e : t'}{\Gamma \vdash (\lambda\,(x : t)\ e) : t \to t'}\ \textbf{Lam}$$

Products (pairs)

$$\textbf{×E1}\ \frac{\Gamma \vdash e : A \times B}{\Gamma \vdash (\text{car } e) : A} \qquad \textbf{×E2}\ \frac{\Gamma \vdash e : A \times B}{\Gamma \vdash (\text{cdr } e) : B} \qquad \textbf{×I}\ \frac{\Gamma \vdash e_0 : A \quad \Gamma \vdash e_1 : B}{\Gamma \vdash (\text{cons } e_0\ e_1) : A \times B}$$

Sums (discriminated unions)

$$\textbf{+I1}\ \frac{\Gamma \vdash e : A}{\Gamma \vdash (\text{left } e) : A + B} \qquad \textbf{+I2}\ \frac{\Gamma \vdash e : B}{\Gamma \vdash (\text{right } e) : A + B} \qquad \textbf{+E}\ \frac{\Gamma \vdash e : A + B \quad \Gamma, e_0 : A \vdash e_0' : C \quad \Gamma, e_1 : A \vdash e_1' : C}{\Gamma \vdash \left(\text{case } e \text{ of } (\text{left } e_0\ \Rightarrow e_0')\ (\text{right } e_1 \Rightarrow e_1')\right) : C}$$

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

This type system corresponds precisely to IPL

Negation

¬A is A → ⊥

$$\frac{\Gamma \vdash e : \bot}{\Gamma \vdash (\text{case } e \text{ of}) : t}$$

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:

```
∀ (l : list A) : {l′ : sorted l′ ∧ ∀ x. (member l x) ⇒ (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

$$\frac{\Gamma \vdash f : t \to t}{\Gamma \vdash (Y f) : t}\ \textbf{Y}$$

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**

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

What would `t` be here?

```
(let ([fib
 (Y (λ (f) (λ (x)
              (if (= x 0)
                  1
                  (* x (fib (- x 1)))))))]))
```

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) (λ (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) (λ (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	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`

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"

```
((λ (x:t) x:t') : num -> num)
```

The `num->num` constraint then **forces** `t = num` and `t' = num`

Type **Inference**

Type inference can **fail**, too…

```
(λ (x) (λ (y:num->num) ((+ (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)

```
(λ (f) (((f 2) 3) 4))
```

↓ Type inference

```
(λ (f : num -> num -> num -> num) (((f 2) 3) 4))
```

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

```
∃T. (λ (f : T) (((f 2) 3) 4))
```

How hard is this problem (tractability)?

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

```
∃T. (λ (f : T) (((f 2) 3) 4))
```

There are an infinite number of *possible* T (e.g., `int, bool, int->int, bool->bool`, …) that we *could* check, in principle

So it is *not* obvious that this is a terminating process. *But*: humans almost always write "reasonable" types:
`((a -> ((a -> b) -> ((a -> b) -> (b -> c))) -> …)` is *possible* but uncommon

We will see next lecture that a procedure exists which finds a typing, *if* a typing exists. This relies on *unification* (a principle from logic programming)

What is the correct type?

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

Is it:

(a) f = int->int, x = int

(b) f = bool->int, x = bool

(c) f = (int->int)->int, x = int->int

What is the correct type?

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

Is it:

(a) f = int->int, x = int

(b) f = bool->int, x = bool

(c) f = (int->int)->int, x = int->int

(d) All of the above

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 = nat->nat and x = nat

This usage requires f = bool->nat and x = 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 -> 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**

Building Constraints

```
(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))))]))
```

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`

Unification

```
;; 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")])])))
```

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++/…

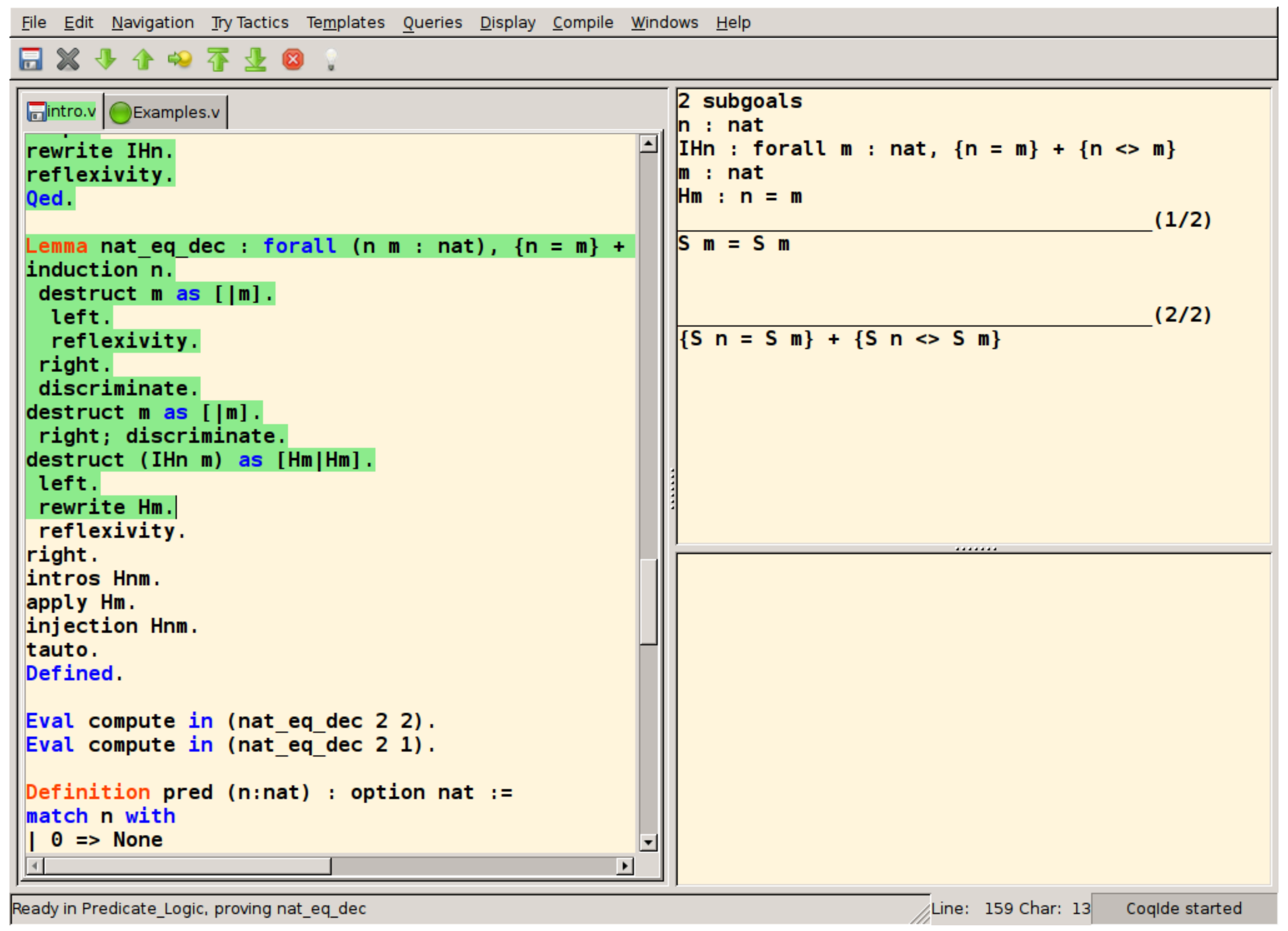
- Less **mental burden**, like CoPilot (etc… tools), type systems can integrate into IDEs to use synthesis information in guiding programming
  - In some ways, this reflects the logical statements underlying the type system’s design (Curry Howard)

“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!

```
"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)."
```



Dependent Type Systems

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

```
∀ (l : list A) : {l′ : sorted l′ ∧ ∀ (x : A). (member l x) ⇒ (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)

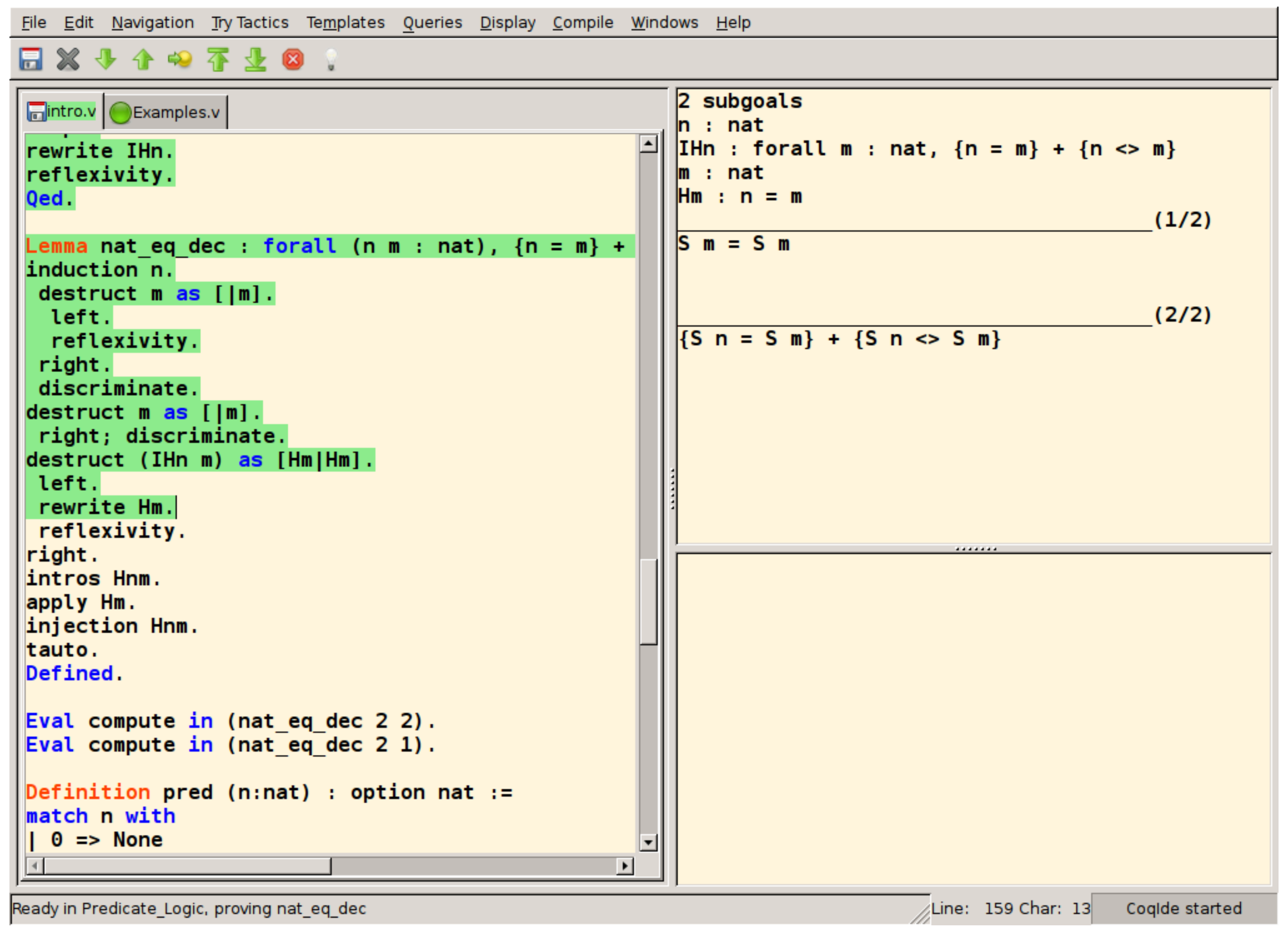
Precise formalization of these systems is beyond the scope of this class

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.

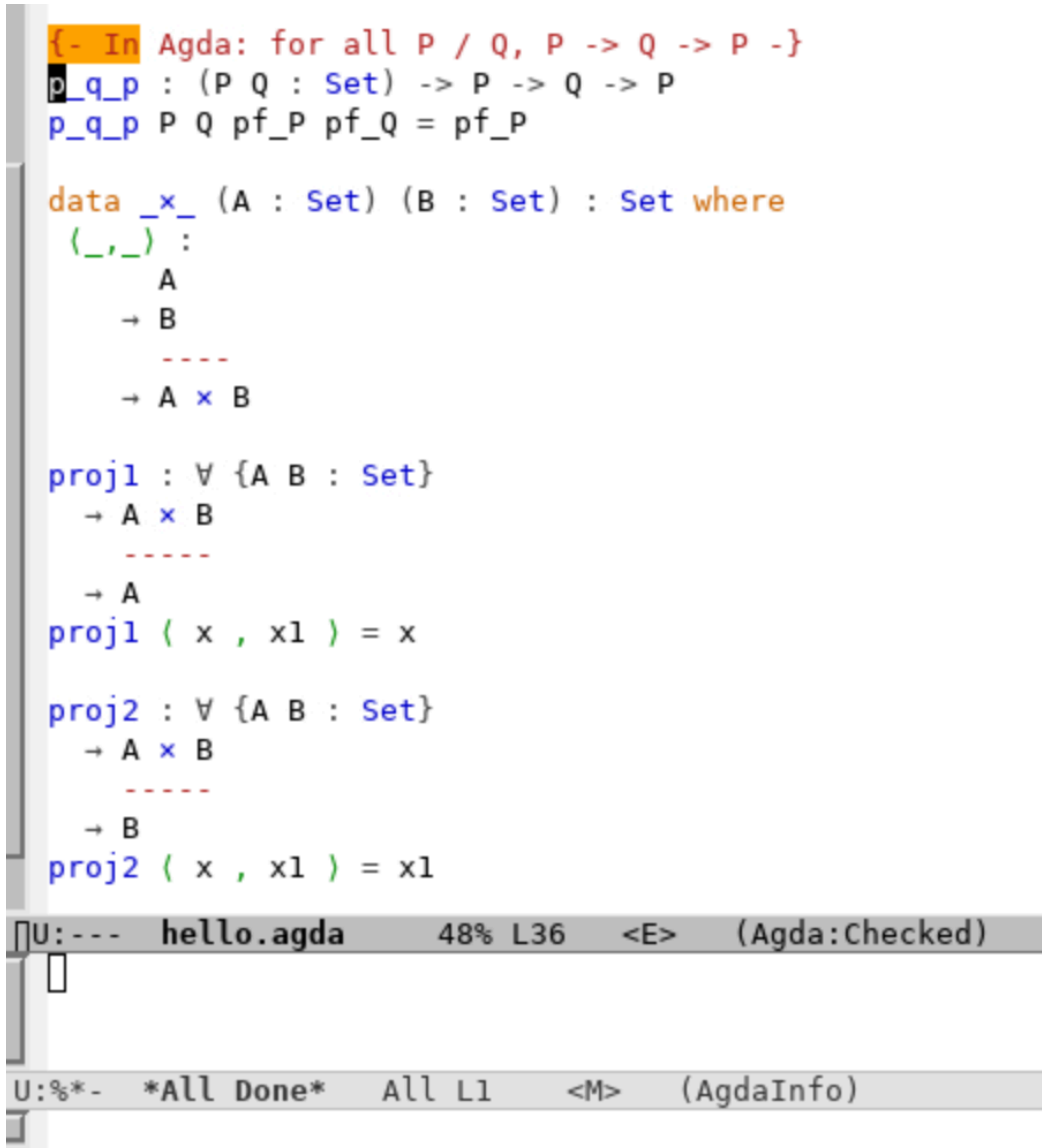
Highly-expressive systems require you to write all the proofs yourself, and a lot of manual annotation (potentially).

```
"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)."
```



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 _×_ (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)
```

waterloo.ca/~plragde/747/notes/index.html

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.

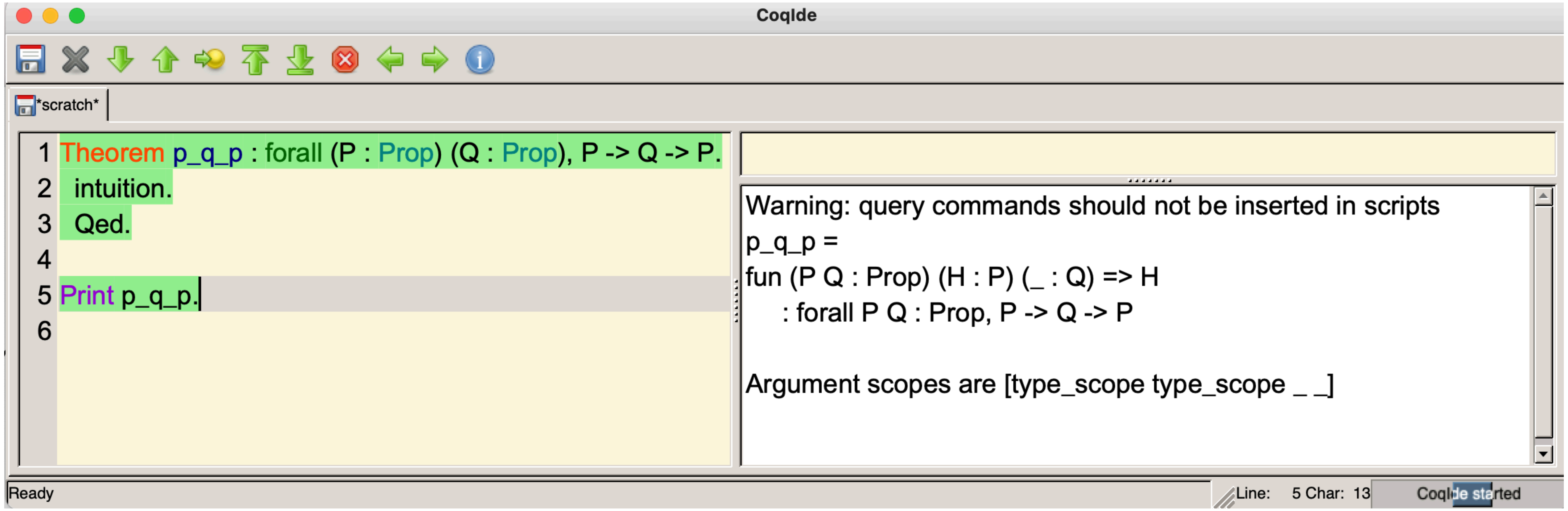
```
proj1 : ∀ {A B : Set}
  → A × B
  -----
  → A
proj1 ⟨ x , x1 ⟩ = x

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



(Using Coq to prove P ⇒ Q ⇒ P; left: using the "intuition" tactic, right: printing the proof term)

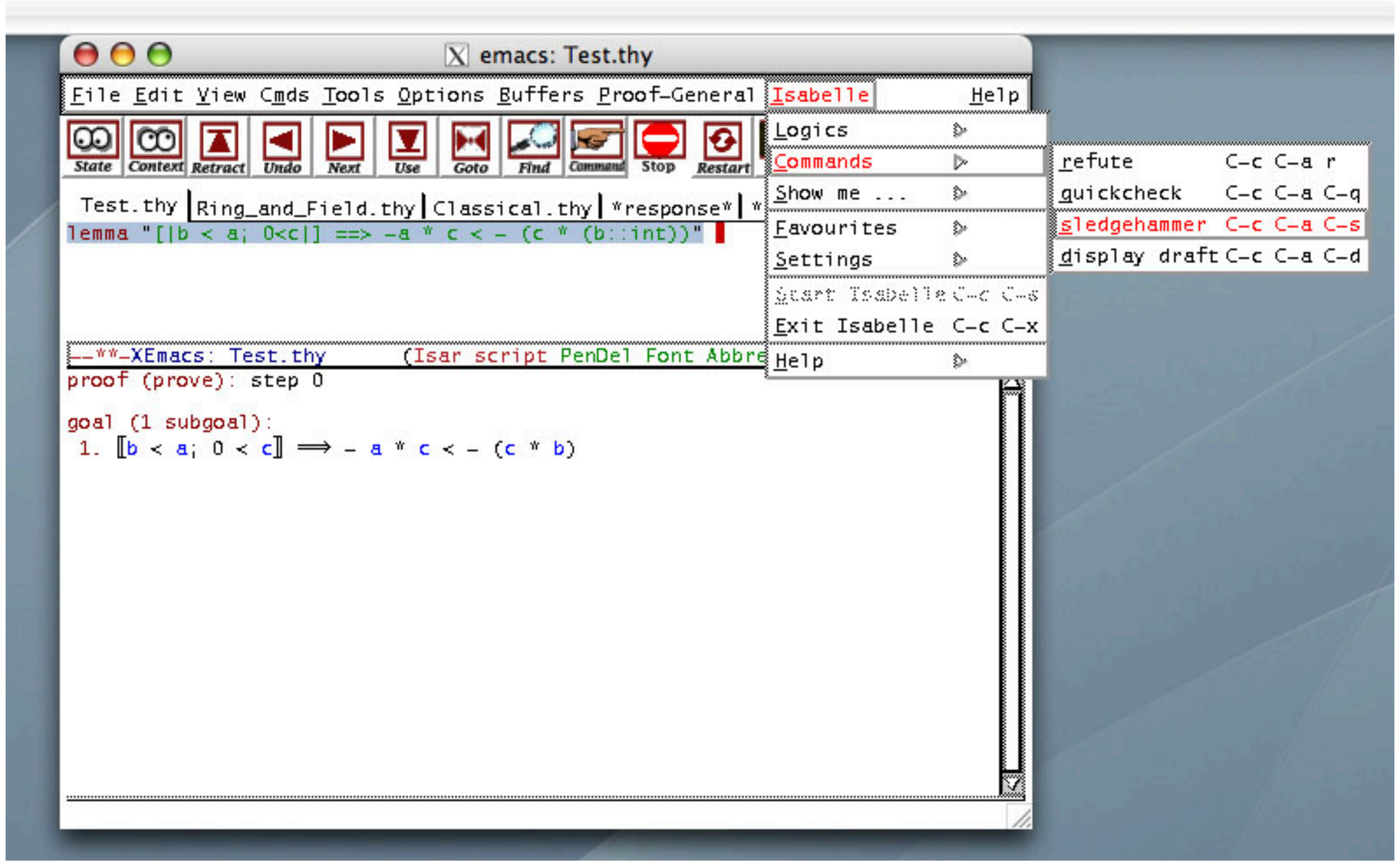
Automating proof via constraint solving

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.

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

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