Simply-Typed Lambda Calculus, & Programs as Proofs

CIS352 — Fall 2022

Kris Micinski

- Types are a static system guaranteed by your program
- Types serve as evidence of a particular property, that relates to the **structure** of information
- For the lambda calculus, and base values, the only structure to be had is **lambdas**
- Type systems are designed to ensure certain static properties of the language. These properties can be relatively superficial, or fairly involved
- The simply-typed lambda calculus is one specific type system for the lambda calculus that models all of the things that could "go wrong" at the type level
- Start by type system for IfArith

Higher-order contract systems track program labels alongside contracts to ***properly assign blame*** when failure occurs.

“Correct blame for contracts”. **Dimoulas. 2011.**

“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!

A type system for STLC

(Simply-Typed Lambda Calculus)

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

prim ::= + | * | …
```

Term Syntax

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

prim ::= + | * | …
```

Type Syntax

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

Term Syntax

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

prim ::= + | * | …
```

Type Syntax

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

Function Types

Term Syntax

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

prim ::= + | * | …
```

Type Syntax

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

Examples…

```
(int -> int) -> int

bool -> int

bool -> (int -> bool)
```

```
;; 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]))
```

A type system for STLC

Type rules are written in natural-deduction style

(Like our big-step operational semantics.)

Assumptions above the line

(No assumptions here.)

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

Conclusions below the line

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

prim ::= + | * | …
```

A type system for STLC

Type rules are written in natural-deduction style

(Like our big-step operational semantics.)

Assumptions above the line

(No assumptions here.)

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

Conclusions below the line

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

prim ::= + | * | …
```

"We may conclude any number n has type **num**"

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

Const revisited…

“We may conclude any constant n is of type **num** under **any** typing environment.”

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

Functions…

If you conclude that e has type t’ with Gamma **plus** assuming x has type t,…

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

Then you can conclude that the entire lambda has type `t -> t’`

Functions…

If you conclude that e has type t' with Gamma **plus** assuming x has type t,…

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

Then you can conclude that the entire lambda has type `t -> t’`

Note

Variables (x) must be **tagged** with a type (e.g., by programmer)

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

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

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

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

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

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

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

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

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

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

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

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

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

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

So t’ = **num**

$$\frac{\overline{\{x \mapsto \textbf{num}\} \vdash 1 : \textbf{num}}}{\Gamma = \{\,\} \vdash \texttt{(lambda (x : num) 1)} : \textbf{num} \rightarrow \textbf{num}}$$

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

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

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

Our type system so far…

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

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

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

Almost everything! Just need builtin functions

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

prim ::= + | * | …
```

Trick! Just **assume** they're part of Γ!

$$\Gamma_\iota = \{ + : \textbf{num} \rightarrow \textbf{num} \rightarrow \textbf{num}, \ldots \}$$

Practice Derivations

Write derivations of the following expressions…

```
((λ (x : int) x) 1)
```

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

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

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

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

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

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

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

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

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…

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

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

Type **Checking**

Type **checking:** verifying the derivation of a **fully-typed** term

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

Notice that each subterm is assigned a “full” type

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

Type checking tells us which rules we **must** apply **if there is** to be a derivation

In the case of fully-annotated STLC, there are no parts where we have to *guess* a type

We can synthesize a type by looking at the annotated parameters for lambdas

This leads us to writing a **syntax-directed** (i.e., structurally-recursive) type synthesizer / checker for fully-annotated STLC

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

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

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` type 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) (((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)

Extending STLC…

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

prim ::= + | * | …
```

Let’s add **if, and, or**

Extending STLC…

```
e ::= (lambda (x) e)
    | (e e)
    | ((prim e) e)
    | (if e e e)
    | (and e e)
    | (or e e)
    | x
    | n | #t | #f

prim ::= + | * | …
```

Now we need typing rules for if!

If needs guard to be a boolean…

Shouldn't be valid for guard to be, e.g., `(+ 1 2)`

```
(if guard
    t
    f)
```

If needs guard to be a boolean…

Shouldn't be valid for guard to be, e.g., `(+ 1 2)`

```
(if guard
    t
    f)
```

$$\frac{\Gamma \vdash e_g : \textbf{bool} \quad \Gamma \vdash e_t : t \quad \Gamma \vdash e_f : t}{\Gamma \vdash (\textbf{if}\ e_g\ e_t\ e_f) : t}\ \textbf{If}$$

If needs guard to be a boolean…

Shouldn't be valid for guard to be, e.g., `(+ 1 2)`

```
(if guard
    t
    f)
```

e_t/e_f must be same type!

$$\frac{\Gamma \vdash e_g : \textbf{bool} \quad \Gamma \vdash \boxed{e_t : t} \quad \Gamma \vdash \boxed{e_f : t}}{\Gamma \vdash (\textbf{if}\ e_g\ e_t\ e_f) : t}\ \textbf{If}$$

Exercise

Can you come up with the type rules for and/or?

`(and` e_1 e_2`)`

$$\frac{\Gamma \vdash e_1 : \textbf{bool} \qquad \Gamma \vdash e_2 : \textbf{bool}}{\Gamma \vdash (\textbf{and}\ e_1\ e_2) : \textbf{bool}}\ \textbf{And}$$

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** to allow typing recursive structures
    - `tree = Leaf (int) | Node(int,tree,tree)`

Typing the Y Combinator

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

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

Typing the Y Combinator

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

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

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

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

Preservation

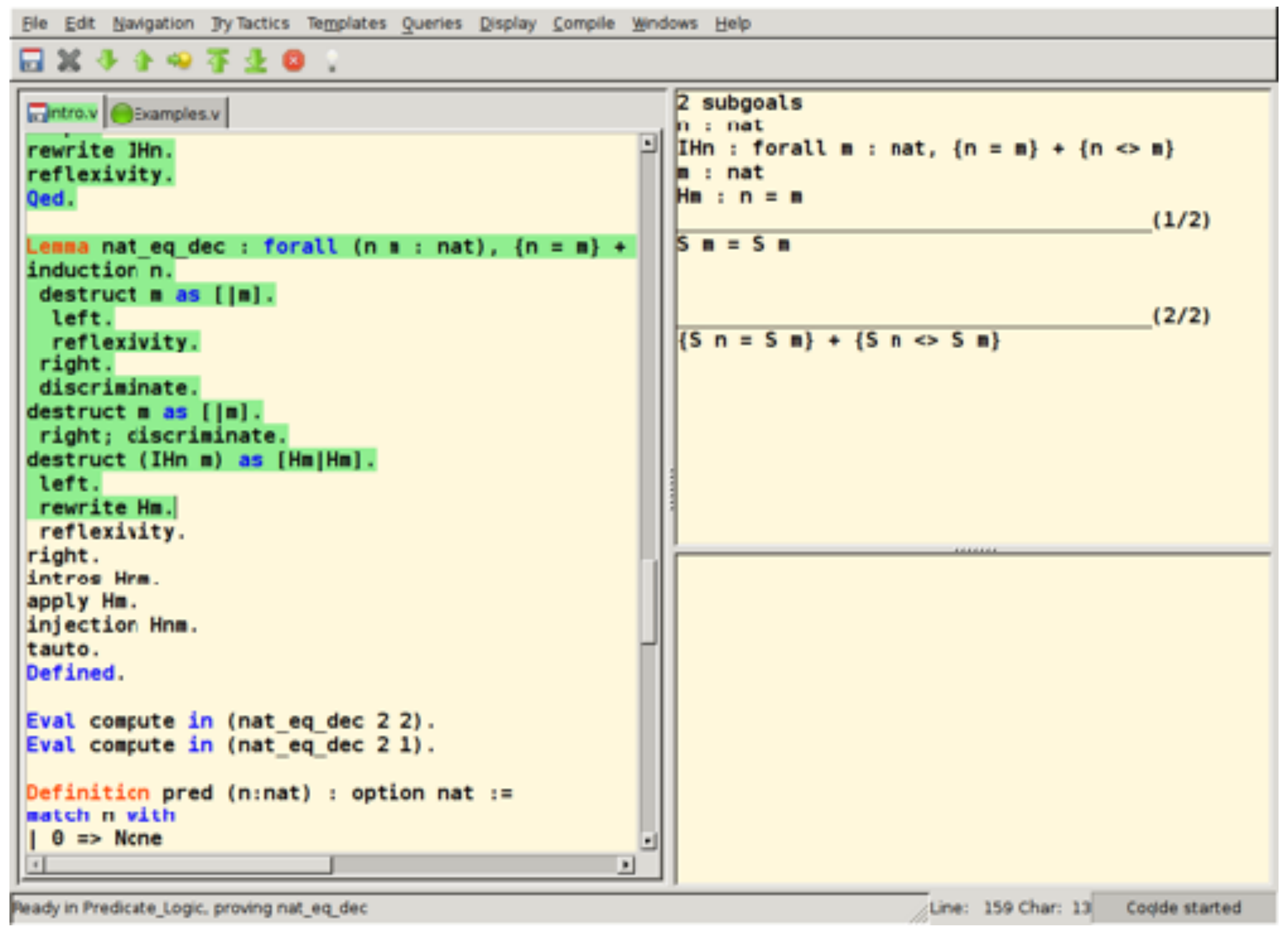
If e has type `t`, any reduction will result in a term of type `t`

“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)."
```



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.

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

$s,\ t,\ A,\ B ::=$	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
	$\mid\ \pi_1\ t \quad \mid \quad \pi_2\ t$	projection
	$\mid\ \mathsf{Set}_i$	universes ($i \in \{0..\}$)
	$\mid\ 1$	the unit type
	$\mid\ \langle\rangle$	the element of the unit type
$\Gamma,\ \Delta$	$::= \varepsilon$	
	$\mid\ (x\ :\ A)\Gamma$	telescopes

Intuitionistic Propositional Logic

Constructive logic variant of traditional propositional (boolean) logic

Proofs in (intuitionistic) propositional logic are built from natural-deduction rules, including **introduction** and **elimination** rules

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

Conjunction Introduction $$\frac{\Gamma \vdash \phi \qquad \Gamma \vdash \psi}{\Gamma \vdash \phi \wedge \psi}$$

Conjunction Left-Elimination $$\frac{\Gamma \vdash \phi \wedge \psi}{\Gamma \vdash \phi}$$

Conjunction Right-Elimination $$\frac{\Gamma \vdash \phi \wedge \psi}{\Gamma \vdash \phi}$$

More reading:
https://www.classes.cs.uchicago.edu/archive/2003/spring/15300-1/intuitionism.pdf

Implication in IPL

Implication is performed by *introducing-then-discharging*

"If you can prove ψ by assuming ϕ, then you can prove ϕ⇒ ψ"

Sometimes called the *deduction theorem*

Implication **Introduction**

$$\frac{\Gamma, \phi \vdash \psi}{\Gamma \vdash \phi \Rightarrow \psi}$$

"If you have a proof of ϕ⇒ ψ, and a proof of ϕ,

then you can have a proof of ψ"

Sometimes called *modus ponens*

Implication **Elimination**

$$\frac{\Gamma \vdash \phi \Rightarrow \psi \quad \Gamma \vdash \phi}{\Gamma \vdash \psi}$$

Proving P ⇒ (Q ⇒ P)

$$
\dfrac{\dfrac{\dfrac{}{Q, P \vdash P}\ \text{Assumption}}{P \vdash Q \Rightarrow P}\ \Rightarrow \text{Intros}}{\vdash \big(P \Rightarrow (Q \Rightarrow P)\big)}\ \Rightarrow \text{Intros}
$$

Start with a **goal** and then grow a proof according to the rules

Small Point: Proving P ⇒ (Q ⇒ Q)

Should be a simple fix

$$
\dfrac{\dfrac{\dfrac{}{Q, P \vdash P}\,\text{Assumption}}{P \vdash Q \Rightarrow P}\,\Rightarrow\text{Intros}}{\vdash \big(P \Rightarrow (Q \Rightarrow P)\big)}\,\Rightarrow\text{Intros}
$$

$$
\dfrac{\dfrac{\dfrac{}{Q, P \vdash Q}\,\text{Assumption}}{P \vdash Q \Rightarrow P}\,\Rightarrow\text{Intros}}{\vdash \big(P \Rightarrow (Q \Rightarrow P)\big)}\,\Rightarrow\text{Intros}
$$

Unfortunately, our assumption rule **forbids** this:

$$
\dfrac{}{\Gamma, P \vdash P}\,\text{Assumption}
$$

To fix this, we typically add **structural rules** to allow identifying contexts under *reorderings*. Some “sub-structural” logics (linear, affine) explicitly restrict this for particular uses (tracking resources, etc…)

Curry-Howard-Isomorphism

Every well-typed STLC term is a proof of a theorem in intuitionistic propositional logic

```
(lambda (x : int) x) : int -> int
```

Can be interpreted as "P implies P" (P ⇒ P, more properly int ⇒ int)

```
(lambda (x : int) (lambda (y : bool) x)) : (int -> (bool -> int))
```

Can be interpreted "P ⇒ (Q ⇒ P)"

CHI vs. IPL

The key idea is to realize that the typing derivation for STLC **precisely mirrors** the deductive rules of IPL

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

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

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

$$\textbf{⇒I} \quad \frac{\Gamma \vdash \phi \Rightarrow \psi \quad \Gamma \vdash \phi}{\Gamma \vdash \psi}$$

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

$$\textbf{⇒E} \quad \frac{\Gamma, \phi \vdash \psi}{\Gamma \vdash \phi \Rightarrow \psi}$$

This means that every proof tree for STLC can be **trivially-mapped** to a proof tree in IPL. I.e., if `(e : t)` is typeable in STLC, the theorem `t` holds in IPL by construction of the proof built using this mapping

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

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

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

$$\Rightarrow\!\textbf{I} \; \frac{\Gamma \vdash \phi \Rightarrow \psi \quad \Gamma \vdash \phi}{\Gamma \vdash \psi}$$

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

$$\Rightarrow\!\textbf{E} \; \frac{\Gamma, \phi \vdash \psi}{\Gamma \vdash \phi \Rightarrow \psi}$$

A family of logics / type systems

The Curry-Howard Isomorphism is a principle we can use to interpret either type systems or constructive logics

- (*Always* constructive logics because structural type systems **are** fully-materialized, structured proofs)

IPL is a boring logic—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?)

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)

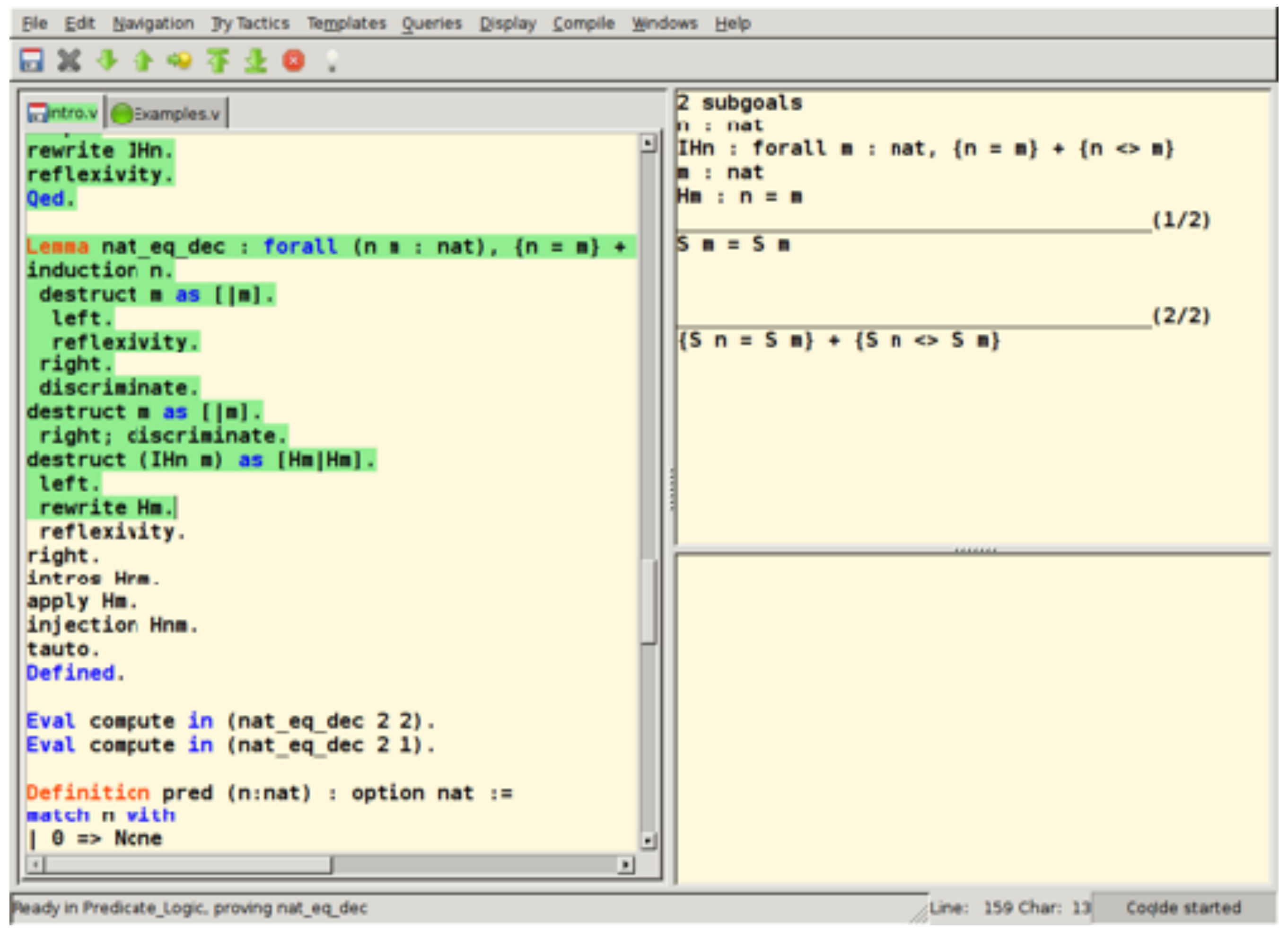
Precise formalization of these systems is beyond the scope of this class

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)
```

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

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

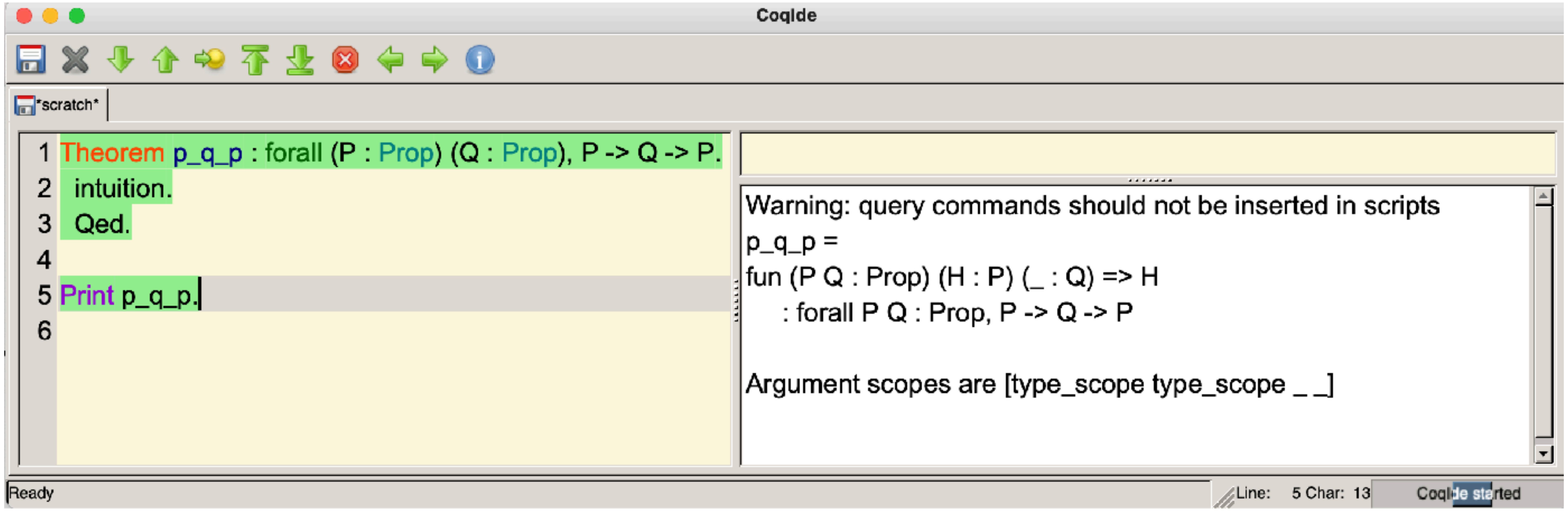
Agda will tell me what I need to fill in, allows me to use “holes” and then helps me hunt for a working proof.

```
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)

Other systems for dependent type syntehsis

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…)