What's left in the course

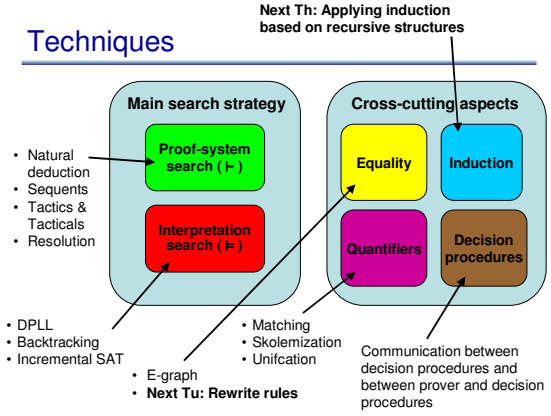
The course in a nutshell

- Logics
- Techniques
- Applications

Logics

- Propositional, first-order, higher-order
- Expressiveness, level of automation, human friendliness
- **Constructive logic: Today!**

Techniques



Applications

- Rhodium
- ESC/Java
- Proof Carrying Code
- **Later: Translation Validation (Zach's project)**

Curry-Howard Isomorphism

But before: Type systems 101

Simply typed lambda calculus

- Consider the simply typed lambda calculus:

e ::=	n	(integers)
|	x	(variables)
|	λ x:τ . e	(function definition)
|	e_1 e_2	(function application)
τ ::=	*int*	(integer type)
|	$\tau_1 \rightarrow \tau_2$	(function type)

Typing rules

- Typing judgment: $\Gamma \vdash e:\tau$
- Is read as: in context Γ, e has type τ
- The context tells us what the type of the free variables in e are
- Example: x:*int* , f:*int* $\rightarrow$ *int* $\vdash$ (f x):int
- Typing rules:

$$\frac{\text{Judgment}_1}{\text{Judgment}_2}$$

Rules for lambda terms

$$\frac{\Gamma \vdash f : \tau_1 \rightarrow \tau_2 \quad \Gamma \vdash x : \tau_1}{\Gamma \vdash (f\ x) : \tau_2}$$

Rules for lambda terms

$$\frac{\Gamma, x:\tau_1 \vdash b : \tau_2}{\Gamma \vdash (\lambda\ x:\tau_1\ .\ b) : \tau_1 \rightarrow \tau_2}$$

What other rules do we need?

$$\frac{}{i : int} \qquad \frac{x:\tau \in \Gamma}{\Gamma \vdash x : \tau}$$

What other rules do we need?

$$\frac{x{:}\tau \in \Gamma}{\Gamma \vdash x{:}\tau} \qquad \frac{}{\Gamma \vdash n{:}\ \text{int}}$$

Summary so far

$$\frac{x{:}\tau \in \Gamma}{\Gamma \vdash x{:}\tau} \qquad \frac{}{\Gamma \vdash n{:}\ \text{int}}$$

$$\frac{\Gamma, x{:}\tau_1 \vdash b{:}\tau_2}{\Gamma \vdash (\lambda\, x{:}\tau_1 .\ b){:}\tau_1 \to \tau_2} \qquad \frac{\Gamma \vdash f{:}\tau_1 \to \tau_2 \quad \Gamma \vdash x{:}\tau_1}{\Gamma \vdash (f\ x){:}\tau_2}$$

Adding pairs

e ::= n | x | λ x:τ . e | e_1 e_2

τ ::= *int* | $\tau_1 \to \tau_2$

Adding pairs

e ::= n | x | λ x:τ . e | e_1 e_2
| (e_1,e_2) (pair construction)
| **fst** e (select first element of a pair)
| **snd** e (select second element of a pair)

τ ::= *int* | $\tau_1 \to \tau_2$
| τ_1 * τ_2 (pair type)

Rules for pairs

$$\frac{\Gamma \vdash x{:}\tau_1 \quad \Gamma \vdash y{:}\tau_2}{\Gamma \vdash (x,y){:}\tau_1 * \tau_2}$$

Rules for pairs

$$\frac{\Gamma \vdash x\ {:}\tau_1 * \tau_2}{\Gamma \vdash \textbf{fst}\ x{:}\tau_1} \qquad \frac{}{\Gamma \vdash \textbf{snd}\ x{:}\tau_2}$$

Rules for pairs (summary)

$$\frac{\Gamma \vdash x:\tau_1 \quad \Gamma \vdash y:\tau_2}{\Gamma \vdash (x,y):\tau_1 * \tau_2} \qquad \frac{\Gamma \vdash x:\tau_1 * \tau_2}{\Gamma \vdash \mathbf{fst}\ x:\tau_1} \qquad \frac{\Gamma \vdash x:\tau_1 * \tau_2}{\Gamma \vdash \mathbf{snd}\ x:\tau_2}$$

Adding unions

e ::= n | x | λ x:τ . e | e_1 e_2 | (e_1,e_2) | **fst** e | **snd** e

inl e

inr e

ccase e of inl x → e inr y → e

τ ::= *int* | $\tau_1 \rightarrow \tau_2$ | τ_1 * τ_2

$\tau_1 | \tau_2$

Adding unions

e ::= n | x | λ x:τ . e | e_1 e_2 | (e_1,e_2) | **fst** e | **snd** e
| **inl** e (create a union of the left case)
| **inr** e (create a union of the right case)
| **case** e **of** **inl** x ⇒ e_1 | **inr** y ⇒ e_2
(perform case analysis on union)

τ ::= *int* | $\tau_1 \rightarrow \tau_2$ | τ_1 * τ_2
| τ_1 + τ_2 (sum (aka union) type)

Rules for unions

$$\frac{\Gamma \vdash x:\tau_1}{\Gamma \vdash \mathbf{inl}\ x:\tau_1 + \tau_2} \qquad \frac{\Gamma \vdash x:\tau_2}{\Gamma \vdash \mathbf{inr}\ x:\tau_1 + \tau_2}$$

Rules for unions

$$\frac{\Gamma \vdash z:\tau_1 + \tau_2 \quad \Gamma, x:\tau_1 \vdash b_1 : \tau \quad \Gamma, y:\tau_2 \vdash b_2 : \tau}{\Gamma \vdash (\mathbf{case}\ z\ \mathbf{of}\ \mathbf{inl}\ x \Rightarrow b_1 \mid \mathbf{inr}\ y \Rightarrow b_2) : \tau}$$

Rules for unions (summary)

$$\frac{\Gamma \vdash x:\tau_1}{\Gamma \vdash \mathbf{inl}\ x:\tau_1 + \tau_2} \qquad \frac{\Gamma \vdash y:\tau_2}{\Gamma \vdash \mathbf{inr}\ x:\tau_1 + \tau_2}$$

$$\frac{\Gamma \vdash z:\tau_1 + \tau_2 \quad \Gamma, x:\tau_1 \vdash b_1 : \tau \quad \Gamma, y:\tau_2 \vdash b_2 : \tau}{\Gamma \vdash (\mathbf{case}\ z\ \mathbf{of}\ \mathbf{inl}\ x \Rightarrow b_1 \mid \mathbf{inr}\ y \Rightarrow b_2) : \tau}$$

Curry-Howard Isomorphism

Typing rules for lambda terms

$$\frac{\Gamma, x{:}\tau_1 \vdash b{:}\tau_2}{\Gamma \vdash (\lambda\, x{:}\tau_1 \,.\, b){:}\tau_1 \to \tau_2} \qquad \frac{\Gamma \vdash f{:}\tau_1 \to \tau_2 \quad \Gamma \vdash x{:}\tau_1}{\Gamma \vdash (f\ x){:}\tau_2}$$

Where have we seen these rules before?

Typing rules for lambda terms

$$\frac{\Gamma, x{:}\tau_1 \vdash b{:}\tau_2}{\Gamma \vdash (\lambda\, x{:}\tau_1 \,.\, b){:}\tau_1 \to \tau_2} \qquad \frac{\Gamma \vdash f{:}\tau_1 \to \tau_2 \quad \Gamma \vdash x{:}\tau_1}{\Gamma \vdash (f\ x){:}\tau_2}$$

Erase terms

$$\frac{\Gamma, \tau_1 \vdash \tau_2}{\Gamma \vdash \tau_1 \to \tau_2} \qquad \frac{\Gamma \vdash \tau_1 \to \tau_2 \quad \Gamma \vdash \tau_1}{\Gamma \vdash \tau_2}$$

Convert to logic

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

Typing rules for pairs

$$\frac{\Gamma \vdash x{:}\tau_1 \quad \Gamma \vdash y{:}\tau_2}{\Gamma \vdash (x,y){:}\tau_1 * \tau_2} \qquad \frac{\Gamma \vdash x{:}\tau_1 * \tau_2}{\Gamma \vdash \mathbf{fst}\ x{:}\tau_1} \qquad \frac{\Gamma \vdash x{:}\tau_1 * \tau_2}{\Gamma \vdash \mathbf{snd}\ x{:}\tau_2}$$

Where have we seen these rules before?

Typing rules for pairs

$$\frac{\Gamma \vdash x{:}\tau_1 \quad \Gamma \vdash y{:}\tau_2}{\Gamma \vdash (x,y){:}\tau_1 * \tau_2} \qquad \frac{\Gamma \vdash x{:}\tau_1 * \tau_2}{\Gamma \vdash \mathbf{fst}\ x{:}\tau_1} \qquad \frac{\Gamma \vdash x{:}\tau_1 * \tau_2}{\Gamma \vdash \mathbf{snd}\ x{:}\tau_2}$$

Erase terms

$$\frac{\Gamma \vdash \tau_1 \quad \Gamma \vdash \tau_2}{\Gamma \vdash \tau_1 * \tau_2} \qquad \frac{\Gamma \vdash \tau_1 * \tau_2}{\Gamma \vdash \tau_1} \qquad \frac{\Gamma \vdash \tau_1 * \tau_2}{\Gamma \vdash \tau_2}$$

Convert to logic

$$\frac{\Gamma \vdash A \quad \Gamma \vdash B}{\Gamma \vdash A \wedge B}\ \wedge\!I \qquad \frac{\Gamma \vdash A \wedge B}{\Gamma \vdash A}\ \wedge\!E_1 \qquad \frac{\Gamma \vdash A \wedge B}{\Gamma \vdash B}\ \wedge\!E_2$$

Typing rules for unions

$$\frac{\Gamma \vdash x{:}\tau_1}{\Gamma \vdash \mathtt{inl}\ x{:}\ \tau_1 + \tau_2} \qquad \frac{\Gamma \vdash y{:}\tau_2}{\Gamma \vdash \mathtt{inr}\ x{:}\ \tau_1 + \tau_2}$$

Where have we seen these rules before?

Typing rules for unions

$$\frac{\Gamma \vdash x{:}\tau_1}{\Gamma \vdash \texttt{inl}\, x{:}\, \tau_1 + \tau_2} \qquad \frac{\Gamma \vdash y{:}\tau_2}{\Gamma \vdash \texttt{inr}\, x{:}\, \tau_1 + \tau_2}$$

Erase terms

$$\frac{\Gamma \vdash \tau_1}{\Gamma \vdash \tau_1 + \tau_2} \qquad \frac{\Gamma \vdash \tau_2}{\Gamma \vdash \tau_1 + \tau_2}$$

Convert to logic

$$\frac{\Gamma \vdash A}{\Gamma \vdash A \vee B}\ \vee I_1 \qquad \frac{\Gamma \vdash B}{\Gamma \vdash A \vee B}\ \vee I_2$$

Typing rules for unions (cont'd)

$$\frac{\Gamma \vdash z{:}\tau_1 + \tau_2 \quad \Gamma, x{:}\tau_1 \vdash b_1 : \tau \quad \Gamma, y{:}\tau_2 \vdash b_2 : \tau}{\Gamma \vdash (\texttt{case}\ z\ \texttt{of}\ \texttt{inl}\ x \Rightarrow b_1 \mid \texttt{inr}\ y \Rightarrow b_2) : \tau}$$

Where have we seen this rule before?

Typing rules for unions (cont'd)

$$\frac{\Gamma \vdash z{:}\tau_1 + \tau_2 \quad \Gamma, x{:}\tau_1 \vdash b_1 : \tau \quad \Gamma, y{:}\tau_2 \vdash b_2 : \tau}{\Gamma \vdash (\texttt{case}\ z\ \texttt{of}\ \texttt{inl}\ x \Rightarrow b_1 \mid \texttt{inr}\ y \Rightarrow b_2) : \tau}$$

Erase terms

$$\frac{\Gamma \vdash \tau_1 + \tau_2 \quad \Gamma, \tau_1 \vdash \tau \quad \Gamma, \tau_2 \vdash \tau}{\Gamma \vdash \tau}$$

Convert to logic

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

Curry-Howard isomorphism

- Propositions-as-types :

  $\tau ::= \ \mathit{int} \mid \tau_1 \rightarrow \tau_2 \mid \tau_1 * \tau_2 \mid \tau_1 + \tau_2$

  $A ::= \ p \mid A_1 \Rightarrow A_2 \mid A_1 \wedge A_2 \mid A_1 \vee A_2$

- If types are propositions, then what are lambda terms?

Typing rules using logic for types

$$\frac{\Gamma, x : A \vdash y : B}{\Gamma \vdash (\lambda\, x : A\, .\, y) : A \Rightarrow B}\Rightarrow I \qquad \frac{\Gamma \vdash f : A \Rightarrow B \quad \Gamma \vdash x : A}{\Gamma \vdash (f\ x) : B}\ \Rightarrow E$$

$$\frac{\Gamma \vdash x : A \quad \Gamma \vdash y : B}{\Gamma \vdash (x,y) : A \wedge B}\ \wedge I \qquad \frac{\Gamma \vdash x : A \wedge B}{\Gamma \vdash \texttt{fst}\ x : A}\ \wedge E_1 \qquad \frac{\Gamma \vdash x : A \wedge B}{\Gamma \vdash \texttt{snd}\ x : B}\ \wedge E_2$$

$$\frac{\Gamma \vdash x : A}{\Gamma \vdash \texttt{inl}\, x : A \vee B}\ \vee I_1 \qquad \frac{\Gamma \vdash y : B}{\Gamma \vdash \texttt{inr}\, x : A \vee B}\ \vee I_2$$

$$\frac{\Gamma \vdash z : A \vee B \quad \Gamma, x : A \vdash e_1 : C \quad \Gamma, y : B \vdash e_2 : C}{\Gamma \vdash (\texttt{case}\ z\ \texttt{of}\ \texttt{inl}\ x \Rightarrow e_1 \mid \texttt{inr}\ y \Rightarrow e_2) : C}\ \vee E$$

Curry-Howard isomorphism

- If types are propositions, then what are lambda terms?
- Answer: terms are proofs
- Programs-as-proofs:

  $\Gamma \vdash e{:}A$ means that under assumptions Γ, A holds and has proof e

Example

$$\frac{\Gamma, x : A \vdash y : B}{\Gamma \vdash (\lambda\, x : A\, .\, y) : A \Rightarrow B} \Rightarrow I$$

- A proof of $A \Rightarrow B$ is a function that takes a parameter x of type A (that is to say, a proof of A), and returns something of type B (that is to say, a proof of B)

Another example

$$\frac{\Gamma \vdash f : A \Rightarrow B \quad \Gamma \vdash x : A}{\Gamma \vdash (f\ x) : B} \Rightarrow E$$

- Suppose we have a proof of $A \Rightarrow B$. This is a function f that, given a proof of A, returns a proof of B.
- Suppose also that we have a proof of A, call it x.
- Then applying f to x gives us a proof of B.

Another example

$$\frac{\Gamma \vdash x : A \quad \Gamma \vdash y : B}{\Gamma \vdash (x,y) : A \wedge B} \wedge I$$

- A proof of $A \wedge B$ is just a pair containing the proof of A and the proof of B

Another example

$$\frac{\Gamma \vdash x : A}{\Gamma \vdash \mathbf{inl}\, x : A \vee B} \vee I_1$$

- Given a proof of A, a proof of $A \vee B$ is a union in the left case, which records that we attained the disjunction through the left of the $\vee$
- There is a problem though…

Another example

$$\frac{\Gamma \vdash x : A}{\Gamma \vdash \mathbf{inl}\, x : A \vee B} \vee I_1$$

- Given a proof of A, a proof of $A \vee B$ is a union in the left case, which records that we attained the disjunction through the left of the $\vee$
- Unfortunately, the proof does not record what the right type of the union is.
  - Given that x is a proof of A, what is **inl** x a proof of?
- Ideally, we would like the proof (lambda term) to determine the formula (type). What's the fix?

The fix for $\vee$ proofs (union terms)

- Ideally, we would like the proof (lambda term) to determine the formula (type). What's the fix?

$$\frac{\Gamma \vdash x : A}{\Gamma \vdash \mathbf{inl}^{B}\, x : A \vee B} \vee I_1 \qquad \frac{\Gamma \vdash y : B}{\Gamma \vdash \mathbf{inr}^{A}\, x : A \vee B} \vee I_2$$

The fix for ∨ proofs (union terms)

- Ideally, we would like the proof (lambda term) to determine the formula (type). What's the fix?
- We add the other type to the ∨ proof (union term):

$$\frac{\Gamma \vdash x : A}{\Gamma \vdash \mathbf{inl}^{B}\, x : A \vee B}\ \vee I_1 \qquad \frac{\Gamma \vdash y : B}{\Gamma \vdash \mathbf{inr}^{A}\, x : A \vee B}\ \vee I_2$$

Intuition for quantifiers

- A proof of ∀ x:τ. P(x) is a function that, given a parameter x of type τ, returns a proof of P(x)
- A proof of ∃ x:τ. P(x) is a function that computes a value of type τ for which P(x) holds
- Note that ∀ x:τ.P(x) and ∃ x:τ. P(x) are formulas, and so they are types. But they also contain a type τ inside of them.

Programs-as-proofs

- The programs-as-proofs paradigm is operational: to prove something, we have to provide a program
- This program, when run, produces a computational artifact that represents a proof of the formula
  - the program itself is also a representation of the proof, but so is the final result computed by the program

Curry-Howard breaking down

- Because of the operational nature of the programs-as-proofs paradigm, the paradigm only works for proofs that are constructive
- Consider the formula ∃ x. P(x)
  - A constructive proof must show how to compute the x that makes the formula valid
  - A proof by contradiction would assume ∀ x. ¬ P(x), and then derive false.
  - But this does not give us a way to compute x, which means it doesn't give us a "program-as-proofs" proof.

Curry-Howard breaking down

- Curry-Howard isomorphism only holds for constructive logics
  - Like classical logic, except that we do not allow proofs by contradiction
- The rule that you remove depends on the calculus you're using
  - In our natural deduction calculus, remove the following rule:

$$\frac{\Gamma \vdash \neg\neg A}{\Gamma \vdash A}\ \neg E$$

Constructive logic

- In other calculii, it may be the following rule:

$$\frac{\Gamma, \neg A \vdash \mathbf{F}}{\Gamma \vdash A}$$

- Or it may be the law of the excluded middle:

$$\frac{}{\Gamma \vdash A \vee \neg A}$$

Constructive logic example

- Consider the task of constructing an algorithm that prints 0 if Riemann's Hypothesis holds and prints 1 otherwise.
  - Riemann's Hypothesis has not been proved or disproved (Fermat's last theorem was previously used, until it was proven…)
- Does such an algorithm exists?

Constructive logic example

- Consider the task of constructing an algorithm that prints 0 if Riemann's Hypothesis holds and prints 1 otherwise.
  - Riemann's Hypothesis has not been proved or disproved (Fermat's last theorem was previously used, until it was proven…)
- Does such an algorithm exists?
  - Classicists: yes
  - Constructivists: don't know yet. Need to wait until Riemann's Hypothesis is proven or disproven

Another example

- Consider the following theorem, which holds in classical logic:
  - Every infinite sequence of 0's and 1's has a least upper bound
- Translation to constructive logic:
  - There exists an algorithm that given an algorithm f from natural numbers of {0, 1}, computes the least upper bound of $\{ f(n) \mid n \geq 0 \}$
- Doesn't hold in constructive logic, because the algorithm doesn't exist

Constructive logic

- It may seem that using constructive logic is like tying your hands behind your back before starting a proof. So why use it?
- Several arguments for it, including philosophical ones (for example?)
- One of the concrete arguments in favor of constructive logic is the Curry-Howard isomorphism, which leads to the so-called specifications-as-programs paradigm.

Specifications as programs

- Suppose we want to program an algorithm that given a natural number x produces a natural number y so that a decidable condition P(x,y) is satisfied
- A proof of $\forall$ x. $\exists$ y. P(x,y) in constructive logic yields a program for computing y from x, which is a provably correct implementation of the specification.
- Programs and specifications are the same!

Specifications as programs

- This idea has been used in various contexts, including three widely know theorem provers
  - Coq
  - NuPRL
  - Twelf
- Main challenge
  - extracting ***efficient*** programs from the proofs