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

Main search strategy
- Proof-system search (+)
- Interpretation search (#)

Cross-cutting aspects
- Equality
- Induction
- Quantifiers
- Decision procedures

Main search strategy
- Natural deduction
- Sequents
- Tactics & Tactics
- Resolution

Cross-cutting aspects
- DPLL
- Backtracking
- Incremental SAT
- E-graph
- Next Tu: Rewrite rules

Next Th: Applying induction based on recursive structures

Communication between decision procedures and between prover and decision procedures

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:

  \[ e ::= n \quad \text{(integers)} \]
  \[ | x \quad \text{(variables)} \]
  \[ | \lambda x : \tau . e \quad \text{(function definition)} \]
  \[ | e_1 e_2 \quad \text{(function application)} \]

  \[ \tau ::= \text{int} \quad \text{(integer type)} \]
  \[ | \tau_1 \rightarrow \tau_2 \quad \text{(function type)} \]

Typing rules

- Typing judgment: \( \Gamma \vdash e : \tau \)
  - Is read as: in context \( \Gamma \), \( e \) has type \( \tau \)
  - The context tells us what the type of the free variables in \( e \) are
  - Example: \( x : \text{int} \), \( f : \text{int} \rightarrow \text{int} \)
    \( \vdash (f \; x) : \text{int} \)

- Typing rules:

  \[
  \begin{align*}
  & \text{Judgment}_1 \\
  & \text{Judgment}_2
  \end{align*}
  \]

Rules for lambda terms

\[
\begin{align*}
\Gamma \vdash f : \tau_1 \rightarrow \tau_2, \quad \Gamma \vdash x : \tau_1 \\
\hline
\Gamma \vdash (f \; x) : \tau_2
\end{align*}
\]

What other rules do we need?

\[
\begin{align*}
\Gamma, x : \tau_1 \vdash b : \tau_2 \\
\hline
\Gamma \vdash (\lambda x : \tau_1 . b) : \tau_1 \rightarrow \tau_2
\end{align*}
\]
What other rules do we need?

\[
\frac{\text{x} \in \Gamma}{\Gamma \vdash \text{x}}
\]
\[
\Gamma \vdash \text{int}
\]

Summary so far

\[
\frac{\text{x} \in \Gamma}{\Gamma \vdash \text{x}}
\]
\[
\Gamma \vdash \text{int}
\]

\[
\frac{\Gamma, \text{x}_1 : \tau_1 \vdash \text{b}_2}{\Gamma \vdash (\lambda \text{x}_1 : \tau_1 . \text{b}) : \tau_1 \rightarrow \tau_2}
\]
\[
\frac{\Gamma \vdash \text{f} : \tau_1 \rightarrow \tau_2 \quad \Gamma \vdash \text{x} : \tau_1}{\Gamma \vdash (\text{f} \text{x}) : \tau_2}
\]

Adding pairs

\[
e ::= \text{n} \mid \text{x} \mid \lambda \text{x} : \tau . e \mid e_1 e_2
\]

\[
\tau ::= \text{int} \mid \tau_1 \rightarrow \tau_2 \mid \tau_1 \times \tau_2
\]

Rules for pairs

\[
\frac{\Gamma \vdash \text{x} : \tau \quad \Gamma \vdash \text{y} : \tau}{\Gamma \vdash (\text{x}, \text{y}) : \tau \times \tau}
\]

Adding pairs

\[
e ::= \text{n} \mid \text{x} \mid \lambda \text{x} : \tau . e \mid e_1 e_2
\mid (e_1, e_2) \quad \text{(pair construction)}
\mid \text{fst} e \quad \text{(select first element of a pair)}
\mid \text{snd} e \quad \text{(select second element of a pair)}
\mid \tau_1 \times \tau_2 \quad \text{(pair type)}
\]

Rules for pairs

\[
\frac{\Gamma \vdash (\text{x}, \text{y}) : \tau \times \tau}{\Gamma \vdash \text{fst} \text{x} : \tau_1}
\]
\[
\frac{\Gamma \vdash (\text{x}, \text{y}) : \tau \times \tau}{\Gamma \vdash \text{snd} \text{x} : \tau_2}
\]
Rules for pairs (summary)

\[
\begin{align*}
\Gamma \vdash x : \tau_1 \quad \Gamma \vdash y : \tau_2 \\
\Gamma \vdash (x,y) : \tau_1 \ast \tau_2 \\
\Gamma \vdash \text{fst} \, x : \tau_1 \\
\Gamma \vdash \text{snd} \, x : \tau_2 
\end{align*}
\]

Adding unions

\[
e ::= n \mid x \mid \lambda x. \, e \mid e_1 \, e_2 \mid \text{fst} \, e \mid \text{snd} \, e \\
\begin{align*}
inl \, e & \quad \text{(create a union of the left case)} \\
inr \, e & \quad \text{(create a union of the right case)} \\
\text{case} \, e \, \text{of inl} \, x \Rightarrow b_1 \mid \text{inr} \, y \Rightarrow b_2 & \quad \text{(perform case analysis on union)}
\end{align*}
\]

\[
\begin{align*}
\tau ::= \text{int} & \mid \tau_1 \to \tau_2 \\
& \mid \tau_1 \ast \tau_2 \\
& \mid \tau_1 + \tau_2 \quad \text{(sum aka union type)}
\end{align*}
\]

Rules for unions (summary)

\[
\begin{align*}
\Gamma \vdash x : \tau_1 \quad \Gamma \vdash y : \tau_2 \\
\Gamma \vdash \text{inl} \, x : \tau_1 + \tau_2 \\
\Gamma \vdash \text{inr} \, x : \tau_1 + \tau_2 \\
\Gamma \vdash \text{case} \, z \, \text{of inl} \, x \Rightarrow b_1 \mid \text{inr} \, y \Rightarrow b_2 : \tau
\end{align*}
\]
Curry-Howard Isomorphism

Typing rules for lambda terms

Where have we seen these rules before?

Typing rules for pairs

Where have we seen these rules before?

Typing rules for unions

Where have we seen these rules before?
Typing rules for unions

\[ \Gamma \vdash x : \tau \]
\[ \Gamma \vdash y : \tau \]
\[ \Gamma \vdash \text{inl} \; x : \tau_1 + \tau_2 \]
\[ \Gamma \vdash \text{inr} \; y : \tau_1 + \tau_2 \]

Erase terms

\[ \Gamma \vdash \tau \]
\[ \Gamma \vdash \tau_1 + \tau_2 \]

Convert to logic

\[ \Gamma \vdash \text{A} \lor \text{B} \]
\[ \Gamma \vdash \text{C} \lor \text{E} \]

Typing rules using logic for types

\[ \Gamma \vdash x : \text{A} \]
\[ \Gamma \vdash y : \text{B} \]
\[ \Gamma \vdash (\text{x} \cdot \text{A} \land \text{B}) : \text{A} \land \text{B} \]
\[ \Gamma \vdash (\text{x} \cdot \text{A} \land \text{B}) \land \text{E}_1 \]
\[ \Gamma \vdash (\text{x} \cdot \text{A}) \land \text{E}_2 \]
\[ \Gamma \vdash (\text{x} \cdot \text{A} \lor \text{B}) : \text{A} \lor \text{B} \]
\[ \Gamma \vdash \text{inl} \; x : \text{A} \lor \text{B} \]
\[ \Gamma \vdash \text{inr} \; y : \text{A} \lor \text{B} \]
\[ \Gamma \vdash (\text{case z of inl x = e} \mid \text{inr y = e}) : \text{C} \]

Curry-Howard isomorphism

- Propositions-as-types:
  \[ \tau ::= \text{int} \mid \tau_1 \rightarrow \tau_2 \mid \tau_1 * \tau_2 \mid \tau_1 + \tau_2 \]
  \[ A ::= \text{p} \mid A_1 \Rightarrow A_2 \mid A_1 \land A_2 \mid A_1 \lor A_2 \]

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

  - Programs-as-proofs:
    \[ \Gamma \vdash e : A \text{ means that under assumptions } \Gamma, A \text{ holds and has proof e} \]
Example

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 $B$)

$$
\Gamma, x : A \vdash y : B \\
\Gamma \vdash \lambda x : A . y : A \Rightarrow B
$$

• A proof of $A \land B$ is just a pair containing the proof of $A$ and the proof of $B$

$$
\Gamma, x : A \vdash y : B \\
\Gamma \vdash (x, y) : A \land B
$$

Another example

• 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 $B$.

$$
\Gamma \vdash f : A \Rightarrow B \\
\Gamma \vdash x : A \\
\Gamma, x : A \vdash f(x) : B
$$

Another example

• A proof of $A \lor B$ is a union in the left case, which records that we attained the disjunction through the left of the $\lor$

$$
\Gamma, x : A \lor B \vdash \text{inl} x : A \lor B
$$

• There is a problem though…

Another example

• A proof of $A \lor B$ is a union in the left case, which records that we attained the disjunction through the left of the $\lor$

$$
\Gamma, x : A \lor B \vdash \text{inl} x : A \lor B
$$

Another example

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

$$
\Gamma \vdash x : A \\
\Gamma \vdash \text{inl} x : A \lor B
$$

$$
\Gamma \vdash y : B \\
\Gamma \vdash \text{inr} x : A \lor B
$$
The fix for $\lor$ 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 $\lor$ proof (union term):

$$\begin{array}{ll}
\Gamma \vdash x : A & \Gamma \vdash x : A \lor B \\
\Gamma \vdash y : B & \Gamma \vdash y : A \lor B \\
\end{array}$$

Intuition for quantifiers

- A proof of $\forall x : \tau. P(x)$ is a function that, given a parameter $x$ of type $\tau$, returns a proof of $P(x)$
- A proof of $\exists x : \tau. P(x)$ is a function that computes a value of type $\tau$ for which $P(x)$ holds
- Note that $\forall x : \tau. P(x)$ and $\exists x : \tau. P(x)$ are formulas, and so they are types. But they also contain a type $\tau$ 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 $\exists x. P(x)$
  - A constructive proof must show how to compute the $x$ that makes the formula valid
  - A proof by contradiction would assume $\forall x. \neg P(x)$, 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:

$$\begin{array}{ll}
\Gamma \vdash \neg A & \Gamma \vdash A \\
\Gamma \vdash \neg \neg A & \Gamma \vdash A \\
\end{array}$$

Constructive logic

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

$$\begin{array}{ll}
\Gamma \vdash \neg A & \Gamma \vdash F \\
\Gamma, A \vdash F & \Gamma, \neg \neg A \vdash A \\
\end{array}$$

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

$$\begin{array}{ll}
\Gamma \vdash A \lor \neg A \\
\end{array}$$
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?

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) | n ≥ 0 }
- Doesn’t hold in constructive logic, because the algorithm doesn’t exist

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 ∀ x. ∃ 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

Constructive logic

- 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

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

- 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