Interactive Theorem Proving

Lecture 1: Introduction to Coq

Elias Castegren and David Broman

15 April 2024

Who am I?

- Assistant professor at Uppsala University, Sweden
- Type systems, formal semantics, concurrency

- Learned Coq for OPLSS 2013
- I use Coq for mechanising semantics and their proofs
    - The concurrent object calculus OOlong
    - Delegation and atomicity in actor systems
    - Viktor's system for Statically Resolvable Ambiguity

UPPSALA
UNIVERSITET

What is Coq?

- Coq is an *interactive theorem prover*
  - Compare to *automated theorem provers* such as SAT/SMT-solvers

- Coq is a *dependently typed* programming language

  Technically this is Coq's specification language Gallina

- Coq allows writing proof scripts, using *tactics*

  Technically this is Coq's tactic language Ltac

UPPSALA UNIVERSITET

When is Coq?

- First version developed by Thierry Coquand and Gérard Huet in 1984
  - Calculus of Constructions

Congratulations on 40 years🎉

- Extended by Christine Paulin in 1991
  - Calculus of Inductive Constructions

- Four color theorem by Georges Gonthier in 2002

- Currently developed and maintained by ~40 people

…did you really have to name it that?

- Coq means “Rooster” in French
    - Compare to OCaml, Yacc, Bison, GNU…
- Coq is based on (a derivative of) the Calculus of Constructions (CoC)
- Coq was developed by Thierry Coquand (among others)

- There has been a decision to rename Coq into “The Rocq prover”

Practicalities

- Coq itself can be installed via https://coq.inria.fr
  or your favourite package manager (including opam and Homebrew)
- In order to use Coq meaningfully, you need IDE support!
  - VSCode with the VSCoq extension (recommended by the book)
    - Also requires installing `vscoq-language-server` from opam!
  - Emacs with Proof General (recommended if you use Emacs)

    This is what I will be using for live coding!
  - CoqIDE, maintained by Inria
  - For tinkering with small examples: https://coq.vercel.app

Coq, the Programming Language

- Coq is a purely functional language with *dependent types*
  - Terms can depend on terms (regular functions)

    $$(\lambda x\,.\,\lambda y\,.\,x\ \ y) : (\tau_1 \to \tau_2) \to \tau_1 \to \tau_2$$

  - Terms can depend on types (polymorphic terms)

    $$(\Lambda X\,.\,\lambda x : X\,.\,x) : \forall X\,.\,X \to X$$

  - Types can depend on types (type constructors)

    $$LIST :: \star \to \star$$

  - Types can depend on terms(!)

    $$VECTOR :: \Pi X :: \star\,.\,\Pi n : \mathbb{N}\,.\,[\ldots]$$

Aside: Barendregt's Lambda Cube

Simply Typed
Lambda Calculus $\lambda{\rightarrow}$

Graphics: wikipedia

Aside: Barendregt's Lambda Cube

Polymorphic
Lambda Calculus
(System F)

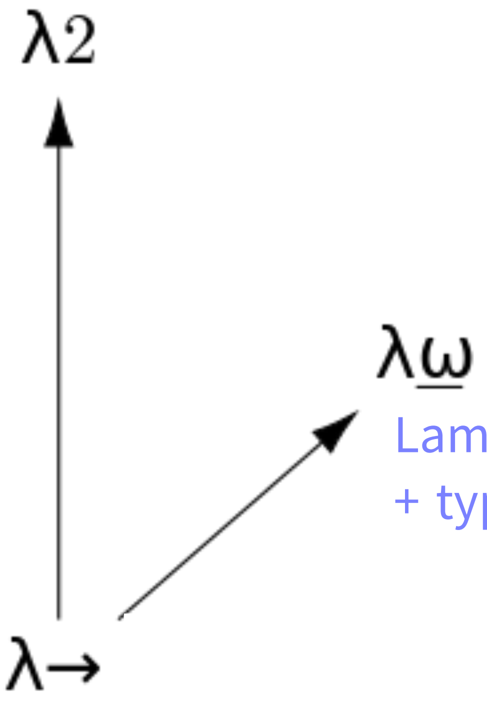
Simply Typed
Lambda Calculus

Graphics: wikipedia

UPPSALA
UNIVERSITET

Aside: Barendregt's Lambda Cube

Polymorphic
Lambda Calculus
(System F)



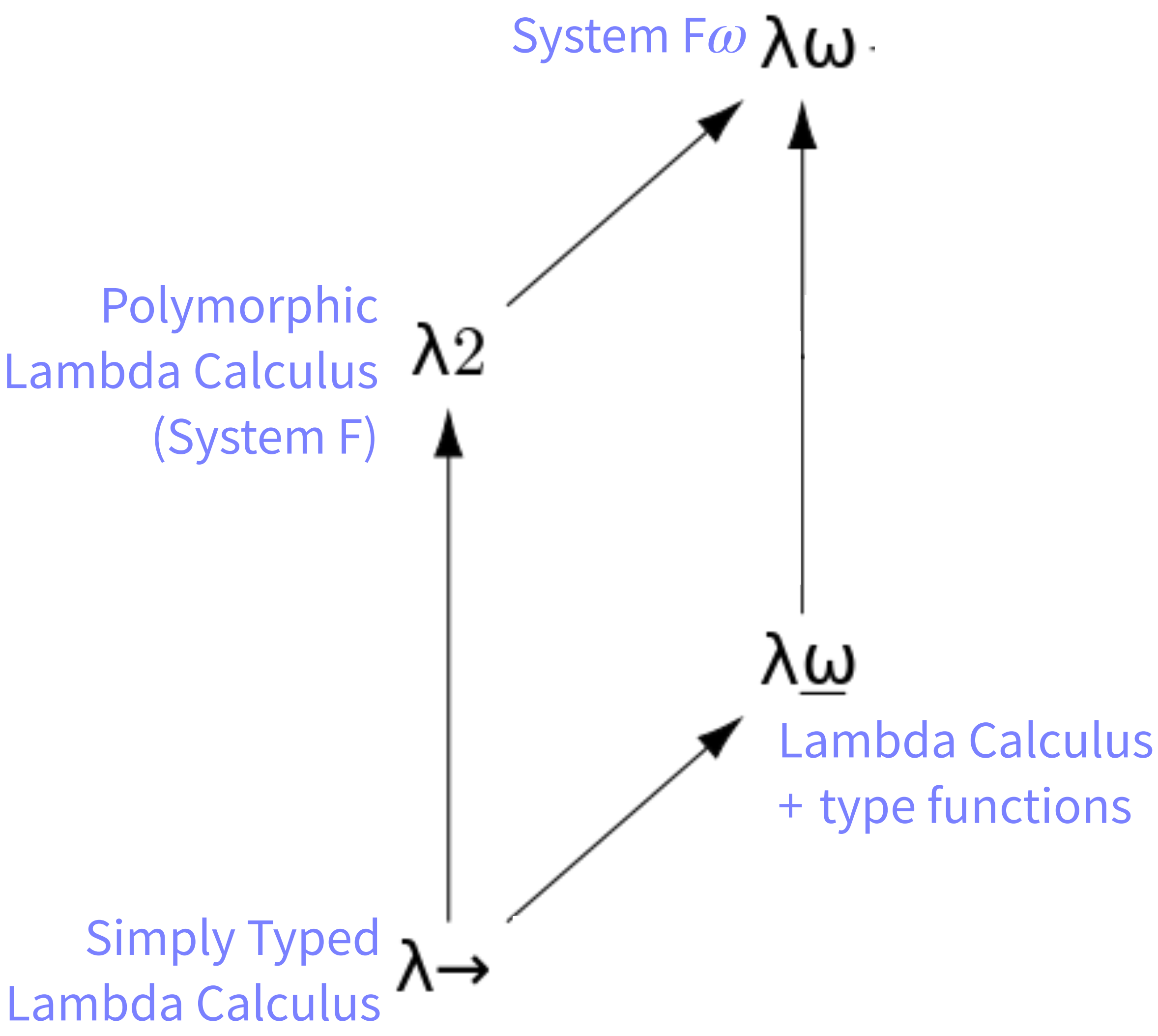
Lambda Calculus
+ type functions

Simply Typed
Lambda Calculus

Graphics: wikipedia

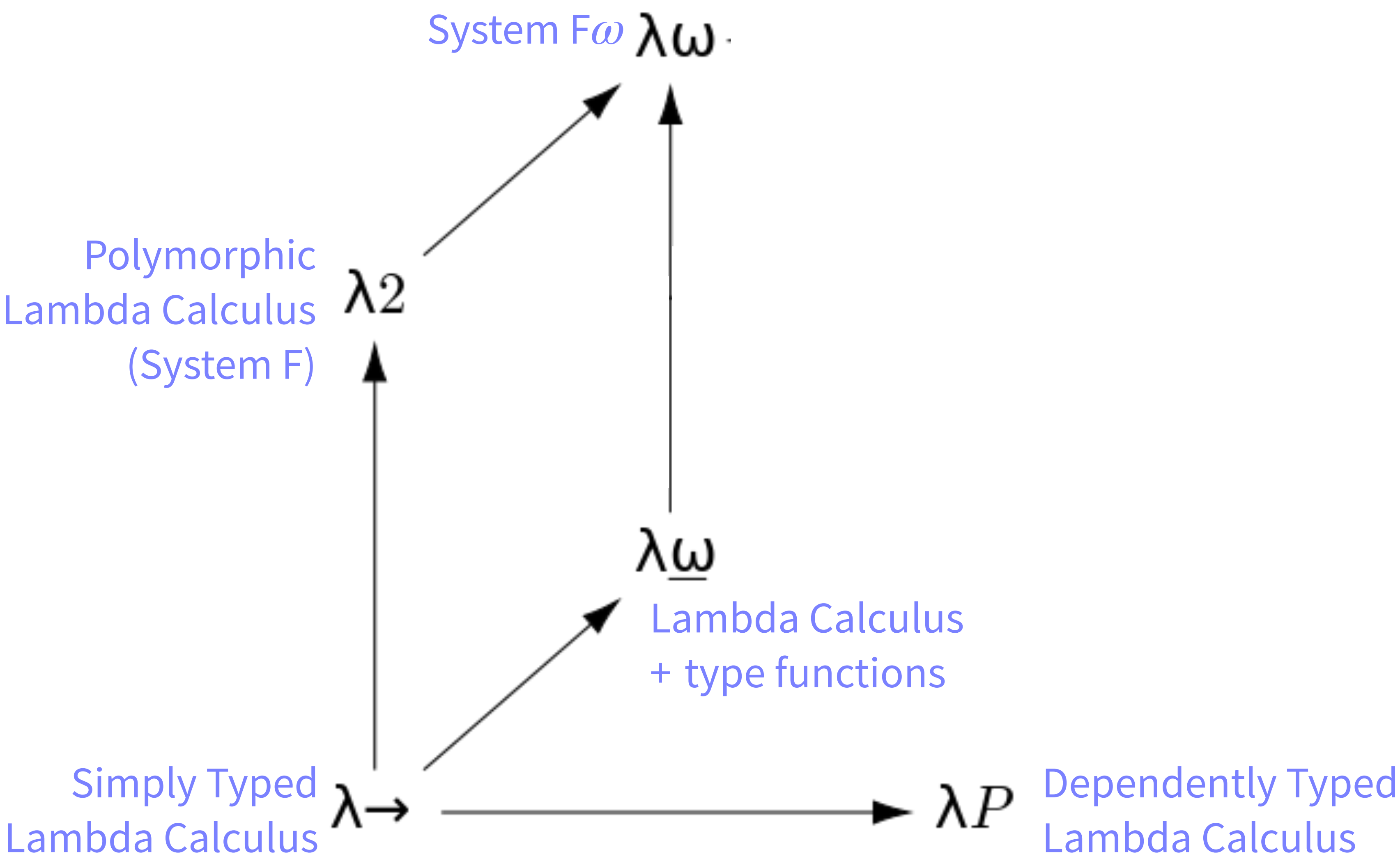
UPPSALA
UNIVERSITET

Aside: Barendregt's Lambda Cube



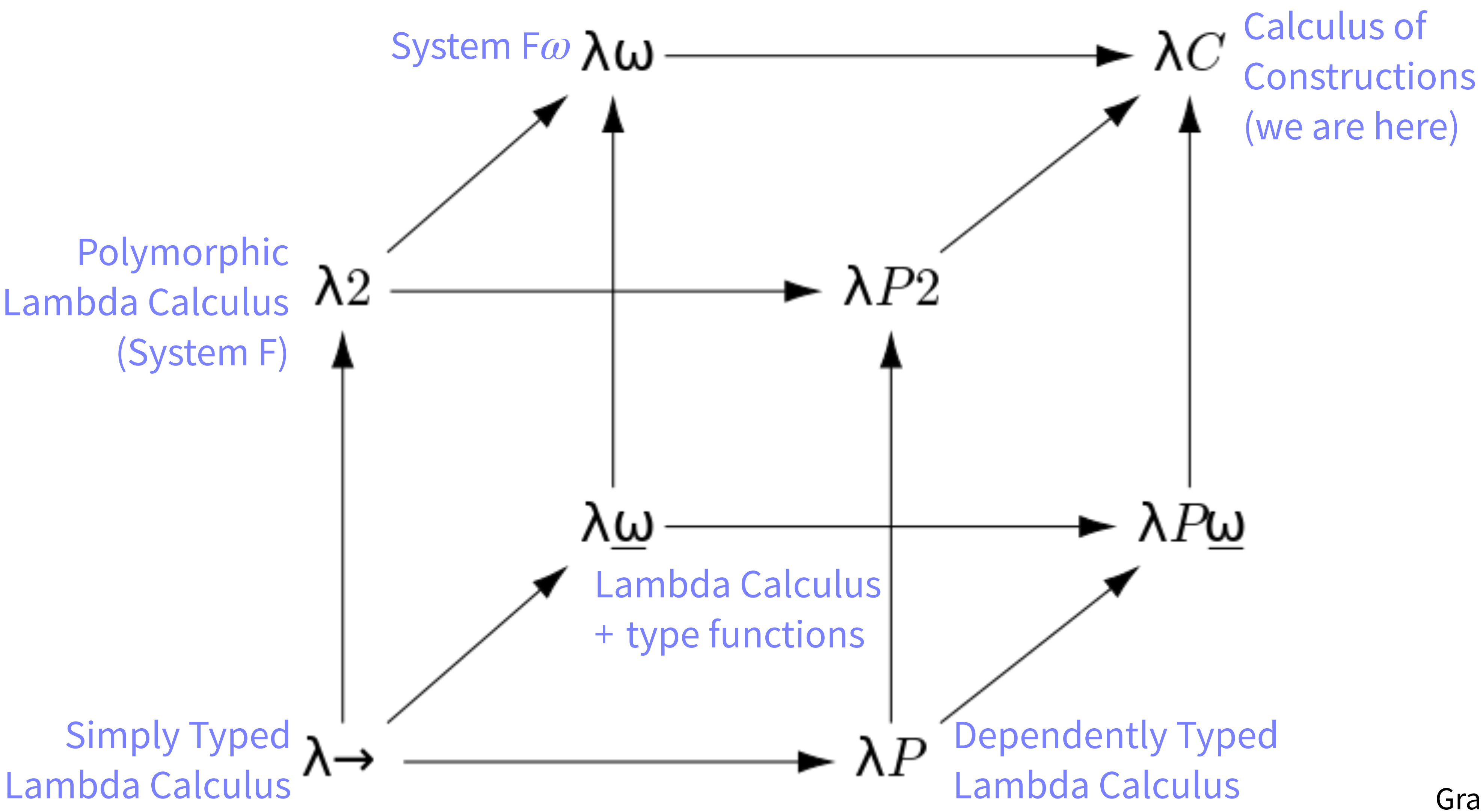
Graphics: wikipedia

Aside: Barendregt's Lambda Cube



Graphics: wikipedia

Aside: Barendregt's Lambda Cube



Graphics: wikipedia

Dependent Types (Example)

- What is the type of `sprintf`?
  - `sprintf "foo"` : string
  - `sprintf "x = %d"` : int → string
  - `sprintf "%s = %d"` : string → int → string
  - `sprintf` : ???

Write in the chat!

The type of `sprintf` depends on its argument!

- `sprintf (s : string) : sprintfType s`

UPPSALA UNIVERSITET

Dependent Types (Example)

```
Definition string := list ascii.  Definition of a type (or term)

Inductive format :=   Definition of an inductive data type
| Fmt_d (* %d *)
| Fmt_c (* %c *)
| Fmt_s (* %s *)
| Fmt__ (c : ascii). (* any other character c *)

Definition format_string := list format.

Fixpoint to_format (s: string): format_string :=   Recursive function
  match s with  Pattern matching
  | nil => nil
  | "%" :: "d" :: s' => Fmt_d :: to_format s'
  | "%" :: "c" :: s' => Fmt_c :: to_format s'
  | "%" :: "s" :: s' => Fmt_s :: to_format s'  to_format ["f"; "o"; "o"; "%"; "d"] =
  | c          :: s' => Fmt__ c :: to_format s'  [Fmt__ "f"; Fmt__ "o"; Fmt__ "o"; Fmt__d]
  end.
```

UPPSALA UNIVERSITET

Dependent Types (Example)

```
Fixpoint sprintfType' (fmt: format_string): Type :=
  match fmt with
  | nil               => string
  | Fmt_d   :: fmt' => nat -> sprintfType' fmt'
  | Fmt_c   :: fmt' => ascii -> sprintfType' fmt'
  | Fmt_s   :: fmt' => string -> sprintfType' fmt'
  | Fmt__ c :: fmt' => sprintfType' fmt'
  end.
```

Function calculating a type(!)

the type of natural numbers

```
Definition sprintfType (s: string): Type := sprintfType' (to_format s).
```

```
sprintfType "%s = %d" =
sprintfType' [Fmt_s; Fmt__ " "; Fmt__ "="; Fmt__ " "; Fmt_d] =
string -> nat -> string
```

Dependent Types (Example)

```
Fixpoint sprintf' (fmt: format_string) (a: string): sprintfType' fmt :=
  match fmt with
  | nil             => a
  | Fmt_d   :: fmt' => fun n => sprintf' fmt' (a ++ (ascii_of_nat n :: nil))
  | Fmt_c   :: fmt' => fun c => sprintf' fmt' (a ++ (c :: nil))
  | Fmt_s   :: fmt' => fun s => sprintf' fmt' (a ++ s)
  | Fmt__ c :: fmt' => sprintf' fmt' (a ++ (c :: nil))
  end.

Definition sprintf (s: string): sprintfType s :=
  sprintf' (to_format s) nil.
```

The type depends on the parameter!

```
                                  : string -> nat -> string
 sprintf "%s = %d" "foo" 42 =
 sprintf' [Fmt_s; Fmt__ " "; Fmt__ "="; Fmt__ " "; Fmt_d] nil "foo" 42 =
 (fun s => fun n => nil ++ s ++ " = " ++ ascii_of_nat n :: nil) "foo" 42 =
 nil ++ "foo" ++ " = " ++ "42" =
 "foo = 42"
```

UPPSALA UNIVERSITET

Recursion in Coq

```
Fixpoint loop (n: nat) := loop n.
```

"Cannot guess decreasing argument of **fix**"

```
Definition hmm (n: nat): loop n := ...
```

All functions in Coq must be total (i.e. must provably terminate)!

```
Fixpoint merge (xs ys: list nat) :=
  match xs, ys with
  | [], ys' => ys'
  | xs', [] => xs'
  | x::xs', y::ys' =>
      if x <? y then x :: merge xs' ys
                else y :: merge xs ys'
  end.
```

"Cannot guess decreasing argument of **fix**"

Recursion in Coq

```
Fixpoint loop (n: nat) := loop n.
```

"Cannot guess decreasing argument of **fix**"

```
Definition hmm (n: nat): loop n := ...
```

All functions in Coq must be total (i.e. must provably terminate)!

```
Fixpoint merge (xs ys: list nat) (fuel: nat) :=
  match fuel with
  | Z => None
  | S fuel' =>
    match xs, ys with
    | [], ys' => Some ys'
    | xs', [] => Some xs'
    | x::xs', y::ys' =>
        if x <? y then Option.map (cons x) (merge xs' ys) fuel'
                  else Option.map (cons y) (merge xs ys') fuel'
    end
  end.
```

Given enough fuel, merge will be correct

UPPSALA UNIVERSITET

PSA: Dependent Types in Coq

Friends don't let friends

program with dependent types

in Coq

PSA: Dependent Types in Coq

- Dependent types are extremely powerful

- The ergonomics of dependent types is not great, especially not in Coq
  - Try to avoid it as much as possible!

- Dependently typed languages that are nicer to *program* in:
  - Agda
  - Idris
  - Lean?

…but not necessarily *do proofs* in

Any Questions so far?

Coq, the Theorem Prover

- Write formal definitions
  - Using data types and functions over these
- State theorems about these definitions
  - Specifications for Coq functions
  - Properties regarding inductive definitions
- Prove these theorems
  - Each step of the proof is checked by Coq
  - It’s enough to read the specifications and theorems (and check for **Admitted** proofs)

```
Inductive evaluates_to :
  program -> value -> Prop := ...
```

```
Definition halts (p : program) :=
  exists v, evaluates_to p v.
```

```
Fixpoint check_halts (p : program) :=
  ...
```

```
Theorem halting_problem :
  forall p,
    check_halts p = true ->
    halts p.
```

```
Proof.
  (* Hmm... *)
Admitted.
```

UPPSALA UNIVERSITET

Theorem Proving (example)

```
Inductive nat :=
| Z
| S (n: nat).

Definition one := S Z.
Definition two := S one.
Definition three := S two.

Fixpoint plus(a b: nat) :=
  match a with
  | Z => b
  | S a' => S (plus a' b)
  end.

Example one_plus_two:
  plus one two = three.
Proof.
  unfold one. unfold plus. fold three. reflexivity.
Qed.
```

$$n ::= 0 \mid S\ n$$

$$1 \equiv S\ 0$$
$$2 \equiv S\ 1$$
$$3 \equiv S\ 2$$

$$a + b = \begin{cases} b & \text{if } a = 0 \\ S\ (a' + b) & \text{if } a = S\ a' \end{cases}$$

Show that $1 + 2 = 3$

$$1 + 2 = (S\ 0) + 2 = S\ (0 + 2) = S\ 2 = 3$$

Theorem Proving (example)

```
Inductive nat :=
| Z
| S (n: nat).

Fixpoint plus(a b: nat) :=
  match a with
  | Z => b
  | S a' => S (plus a' b)
  end.
```

$$n ::= 0 \mid S\ n$$

$$a + b = \begin{cases} b & \text{if } a = 0 \\ S\ (a' + b) & \text{if } a = S\ a' \end{cases}$$

$$\forall n\,.\, n + 0 = n$$

Audience Participation

Theorem Proving (example)

```
Inductive nat :=
| Z
| S (n: nat).
```

$$n ::= 0 \mid S\ n$$

```
Fixpoint plus(a b: nat) :=
  match a with
  | Z => b
  | S a' => S (plus a' b)
  end.
```

$$a + b = \begin{cases} b & \text{if } a = 0 \\ S\ (a' + b) & \text{if } a = S\ a' \end{cases}$$

```
Theorem plus_Z_r:
  forall n, plus n Z = n.
Proof.
  intros n. induction n.
  - simpl. reflexivity.
  - simpl. rewrite IHn. reflexivity.
Qed.
```

$$\forall n\,.\, n + 0 = n$$

Assume that we have some natural number n.

We proceed by induction over n.

Base case $(n = 0)$: $0 + 0 = 0$, by the definition of $+$.

Inductive case $(n = S\ m)$:

1. $(S\ m) + 0 = S(m + 0)$, by the definition of $+$.

2. $m + 0 = m$ by the induction hypothesis.

3. $S\ (m + 0) = S\ m$, by **2.**

$\square$

Theorem Proving (example)

```
Inductive nat :=
| Z
| S (n: nat).

Fixpoint plus(a b: nat) :=
  match a with
  | Z => b
  | S a' => S (plus a' b)
  end.

Theorem plus_Z_r:
  forall n, plus n Z = n.
Proof.
  intros n. induction n.
  - simpl. reflexivity.
  - simpl. rewrite IHn. reflexivity.
Qed.
```

Don’t do video game proving!
— Robert Harper, CMU

Theorem Proving (hands-on)

If you don't have Coq on your own machine:
https://coq.vercel.app

- Formulate and prove commutativity of addition

  $\forall a\ b\,.\, a + b = b + a$

- You should be able to get by with the following tactics:
  - `intros x1 … xn` — introduce universally quantified variables
  - `induction x` — proceed by induction over `x`
  - `simpl` — simplify the current expression in the goal
  - `rewrite H` — rewrite using the equality `H` (can be other theorems!)
  - `reflexivity` — solve an equality where both sides are syntactically equal
- You will most likely need to prove one or two lemmas!
- You can start from the file `nat_basic.v`

UPPSALA UNIVERSITET

Theorem Proving (hands-on)

```
Lemma plus_S:
  forall a b,
    plus a (S b) = S (plus a b).
Proof.
  intros a b. induction a.
  - reflexivity.
  - simpl. rewrite IHa. reflexivity.
Qed.

Theorem plus_comm:
  forall a b,
    plus a b = plus b a.
Proof.
  intros a b. induction a.
  - rewrite plus_Z_r. reflexivity. (* plus_Z_r defined previously *)
  - simpl. rewrite IHa. rewrite plus_S. reflexivity.
Qed.
```

Theorem Proving in Coq

- Warning: proving things in Coq is highly addictive!
- Prove helper lemmas separately whenever you get stuck
  - It’s better to have 100 simple lemmas than 10 complex theorems
  - Compare to how helper functions improve readability of code
- Always think before you prove. Avoid video game proving!

Question Time Again

Coq, the Tactic Language

- Proofs in Coq are (typically) written using *tactics*
  - Tactics actually build (dependently typed) values
- While programming in Coq is *functional*, tactics are *imperative*
- There is a huge number of built-in tactics in Coq (too many?)
  - Try to be consistent in your own style!
- The `auto` tactic provides *proof automation* through proof search

Interacting with Tactics

- Guiding automation

```
Local Hint Resolve plus_Z_r : nat_db.


Local Hint Extern 1 => myTactic : nat_db.
```

In the current scope,
add the theorem `plus_Z_r`
to the hint database `nat_db`

In the current scope,
allow `auto` to use `myTactic`
with a cost of 1 with `nat_db`

- Writing new tactics

```
Ltac myInduction x := intros x; induction x; simpl.
```

Tactics (example)

```
Local Hint Resolve plus_Z_r : nat_db.
Local Hint Resolve plus_S_r : nat_db.

Ltac perform_rewrite :=
  match goal with
  | H: ?x = _ |- context[?x] => rewrite H
  end.

Local Hint Extern 1 => perform_rewrite : nat_db.

Lemma plus_Z_r: forall n, plus n Z = n.
Proof. intros n. induction n; auto with nat_db. Qed.

Lemma plus_S_r: forall a b, plus a (S b) = S (plus a b).
Proof. intros a b. induction a; auto with nat_db. Qed.

Lemma plus_comm: forall a b, plus a b = plus b a.
Proof. intros a b. induction a; simpl; auto with nat_db. Qed.
```

Proof Automation or Not?

- When learning Coq, avoid automation to learn what is going on!
  - Start automating once you get annoyed with tiny details
- Just adding lemmas to a hint database will get you far! (+ `auto`)
- Software engineering $\Longleftrightarrow$ Proof engineering
  - Is this proof maintainable?
  - Is it resilient to change?
  - Is it using the the right abstractions?
  - …

QED at Large: a Survey of Engineering of Formally Verified Software
Talia Ringer et al.

UPPSALA
UNIVERSITET

Final Words

- Focus of this course is Coq as a *theorem prover*
  - We will connect to dependent types next lecture!
- For now, don't worry about fancy tactics or automation
  - Focus on learning the *craft* of mechanised proofs
- Go have fun with Software Foundations. It's a great book!

Reach out if you get stuck!

UPPSALA
UNIVERSITET