# Interactive Theorem Proving

## Lecture 3: Tactics, Locally Nameless Representation

**Elias Castegren** and David Broman

23 May 2024

UPPSALA
UNIVERSITET

# Tactics in Coq

- Il y a plusieurs façons de plumer un canard

  "There are many ways to pluck a duck"

```
Theorem P_if_P:
  forall P,
    P -> P.
Proof.
  intros P HP.
  apply HP.
(* Any of the following tactics will work
  exact HP.
  assumption.
  trivial.
  auto.
  eauto.
  intuition.
*) Qed.
```

# Goals and Subgoals

- In general a tactic is applied to a goal and either
  - Produces zero or more subgoals
  - Fails
- Some tactics *never* fail, e.g., `simpl`, `auto`, `idtac`…
- Some tactics fail if more than zero subgoals are produced, e.g. `reflexivity`
- Some tactics fail if an identical subgoal is produced, e.g. `rewrite`
- One tactic *always* fails, namely `fail`
  - Useful when writing tactics of your own

# Focusing with Bullets

- When there are one or more subgoals, you can *focus* the proof with bullets

```
Theorem n_plus_Z:
  forall n,
    n + 0 = n.
Proof.
  intros n.
  induction n.
  (* Base case *)
    reflexivity.
  (* Inductive case *)
    simpl. rewrite IHn.
    reflexivity.
Qed.
```

```
Theorem n_plus_Z:
  forall n,
    n + 0 = n.
Proof.
  intros n.
  induction n.
  - (* Base case *)
    reflexivity.
  - (* Inductive case *)
    simpl. rewrite IHn.
    reflexivity.
Qed.
```

Subgoals on the same level must begin with the same kind of bullet

- Available bullets are `-`, `+`, `*`, `--`, `++`, `**`, etc.

# Focusing with Braces

- You can focus on a single goal with braces

```
Theorem n_plus_Z:
  forall n,
    n + 0 = n.
Proof.
  intros n.
  induction n.
  { (* Base case *)
    reflexivity.
  }
  (* Inductive case *)
    simpl. rewrite IHn.
    reflexivity.
Qed.
```

```
Theorem n_plus_Z:
  forall n,
    n + 0 = n.
Proof.
  intros n.
  induction n.
  2: { (* Inductive case *)
    simpl. rewrite IHn.
    reflexivity.
  }
  (* Base case *)
  reflexivity.
Qed.
```

# Advice Regarding Focus

- Use bullets whenever you have more than one goal
- Use braces when you are side-stepping the “main story” of the proof

```
Theorem wf_expr_eval:
  forall n,
    wf_expr (length env) e ->
    exists n, eval_expr env e = Some n.
Proof.
  (* ... *)
  - (* Case Var *)
    assert (Hex: exists n, nth_error env x = Some n).
    { destruct (nth_error env x).
      - exists n. reflexivity.
      - exfalso. apply H. reflexivity.
    }
  (* ... *)
Qed.
```

“Here, we know that x has some value n (which we can show by case analysis on…)”

# Running Example: An SSA Language

```
r0 := 1 + 2;
r1 := r0 * 3;
r2 := isZero r1? 2: 3;
r2 * 2;
```

$$e ::= n \mid r_i \mid e + e \mid e * e \mid \textbf{isZero}\ e ?\ e : e$$

$$p ::= e; p \mid e$$

Register numbering is implicit

$$env : \mathbb{N} \hookrightarrow \mathbb{N}$$
$$eval\_expr : env \to e \hookrightarrow \mathbb{N}$$
$$eval\_program : p \hookrightarrow \mathbb{N}$$

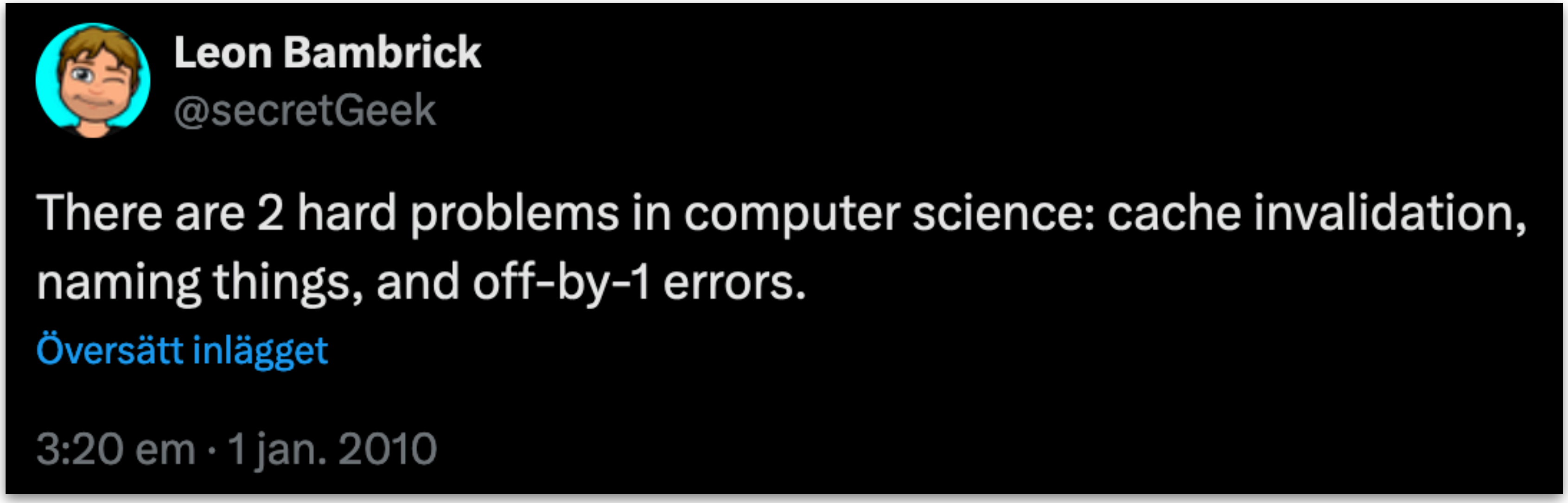
Well-formedness:

$\vdash p \iff p$ only mentions assigned variables

Main theorem:

$$\vdash p \implies \exists n \,.\, eval\_program\ p = n$$

# Naming Things




- Naming things is a hassle, but relying on generated names is worse
  - Adding a hypothesis to your proof can offset your H0, H1…
  - Updating your Coq version can change the numbering scheme(!)



UPPSALA UNIVERSITET

# How to Name Things

- Most tactics that introduce new terms and hypothesis support naming
  - `destruct n as [| n']`
  - `induction l as [| x xs]`
  - `assert (Hlen: length l < 10)`
- The way you declare data types matters

```
Inductive expr :=
| ENat : nat -> expr
| EVar : var -> expr
| EAdd : expr -> expr -> expr.
```

Running `destruct` will give the names `n`, `v` and `e`/`e0`

```
Inductive expr :=
| ENat : (n : nat)
| EVar : (x : var)
| EAdd : (e1 e2 : expr).
```

Running `destruct` will give the names `n`, `x` and `e1`/`e2`

# How to Avoid Naming Things

- Sometimes you introduce something just to immediately forget it

```
destruct IHHwf as [n Heq].
rewrite Heq. (* Heq is no longer used *)
```

- Wherever an introduction pattern is used, you can immediately rewrite!

```
destruct IHHwf as [n ->].
(* Implicitly does rewrite -> Heq. clear Heq. *)
```

- With enough automation, you can find hypothesis automatically
  - A heavy alternative is `match goal with` (see later slides)

# Tacticals

- Tacticals are tactics taking other tactics as arguments
  - `tactic1; tactic2` — run `tactic2` on all subgoals generated by `tactic1`
  - `tactic1 || tactic2` — run `tactic1`, and if it fails run `tactic2`
  - **`try`** `tactic` — run `tactic` but ignore if it fails
  - **`repeat`** `tactic` — run `tactic` until it fails or generates an identical subgoal
  - **`do`** *n* `tactic` — run `tactic` *n* times
  - **`progress`** `tactic` — run `tactic` and require that it produces a new subgoal
  - **`solve`** `tactic` — run `tactic` and fail if it generates more than zero subgoals
  - *n*`: tactic` — run `tactic` on the *n*th goal in focus (1-indexed, can have ranges)
  - `all: tactic` — run `tactic` on all goals in focus

# Tacticals (cont.)

- Some tacticals work with lists of tactics
  - `tactic1; [tac1 | … | tacn]` — run `taci` on the *i*th subgoal of `tactic1`
  - **`solve`** `[tac1 | … | tacn]` — try solving with all listed tactics (in order)
  - **`first`** `[tac1 | … | tacn]` – run the first listed tactic that does not fail

- Two useful patterns:
  - Solve an assert without focusing: `assert (H: ...); [ apply foo; auto|].`
  - Solving similar goals without accidentally touching other goals:

```
induction e;
  try solve [assumption
            |apply foo; auto
            |some_other_tactic].
```

# Questions so far?

# Automation

- The automation tactics search by applying hypotheses recursively
  - Is the current goal a hypotheses? If so, apply it and be done
  - Is the current goal the *conclusion* of a hypothesis? If so, apply it

    ```
    Hint Resolve my_pretty_lemma : hint_db.
    ```

  - Will also `unfold` (when told), `intros` and `simpl` when possible

    ```
    Hint Unfold my_pretty_fixpoint : hint_db.
    ```

- `auto [with hint_db]` — proof search with `simple apply`
- `eauto [with hint_db]` — proof search with `simple eapply`
- `intuition [tactic]` — specialised for logic (can use `tactic`) NB: will update
  - Can `destruct` branching hypotheses: $P \vee Q \implies Q \vee P$ the goal!

# Matching on the Goal

- Tactics can inspect the current proof state

```
match goal with
| [hypotheses] |- [conclusion] => tactic
end.
```

```
match goal with
| H : wf_expr _ _ |- _ => apply wf_expr_eval in H as [n ->]
end.
```

- Can express “something containing $e$” with $\mathtt{context}[e]$

```
match goal with
| _ : _ |- context[match ?e with | _ => _ end] => destruct e
end.
```

# Extending Automation with Custom Tactics

- Matching on the goal is a great way of writing custom tactics

```
Ltac destruct_ex :=
  let n := fresh "n" in
  let Heq := fresh "Heq" in
  match goal with
  | H : exists n, _ = _ |- _ =>
      destruct H as [n Heq];
      try ((rewrite -> Heq in * || rewrite <- Heq in *);
           clear Heq)
  end.
```

Generate fresh names to avoid collisions

- Let *auto* (and friends) use your new tactic with some cost:

```
Hint Extern 1 => destruct_ex : hint_db.
```

# Extending Automation with Custom Tactics

- A manual version together with an automatic

```
Ltac destruct_ex H :=
  let n := fresh "n" in
  let Heq := fresh "Heq" in
  match type of H with
  | exists n, _ = _ =>
      destruct H as [n Heq];
      try ((rewrite -> Heq in * || rewrite <- Heq in *);
           clear Heq)
  | _ => fail "Not an existential equality"
  end.
```

```
Ltac auto_destruct_ex :=
 match goal with
  | exists n, _ = _ |- _ => destruct_ex H
  | _ |- _ => fail "No existential equalities in context"
 end.
```

# Using External Tactic Libraries

- The LibTactics chapter of Software Foundations has custom tactics
- These are from an external library called TLC by Arthur Charguéraud
  - Custom tactics that removes boilerplate
  - Alternative standard library (lists, sets, maps…)
  - Uses type classes heavily to reuse notations
  - NB: Assumes classical logic!
- One of my favourite tactics is `introv`:

```
Theorem transitivity:
  forall P Q R, (P -> Q) -> (Q -> R) -> P -> R.
Proof.
  introv PtoQ QtoR. (* same as intros P Q R PtoQ QtoR *)
  ...
```

# Syntactic Conventions in TLC

- Most new tactics end with an “s” to distinguish them from originals
  - `simpls` — like `simpl`, but better at unfolding
  - *`substs`* — like *`subst`*, but handles circular equalities
  - *`inverts`* `H` — like *`inversion`* `H`, but does *`substs`* and `clear H`
  - *`applys`* `H` — like *`apply`* `H`, but better at handling quantifiers
- Automation makes it easy to say “and then do proof search”
  - `tactic~` — run tactic and then use “auto” on subgoals
  - `tactic*` — run tactic and then use “eauto” on subgoals

```
induction~ n.
simpls. rewrites~ IHn.
```

Actually slightly more involved

# Introduction Patterns in TLC

- There are introduction patterns that handle nesting better

```
Theorem foo:
  forall A B (P Q : A -> B -> Prop),
    (exists x y, P x y /\ Q x y) ->
    ~(forall x y, ~(P x y /\ Q x y)).
Proof.
  introv Hex.
  destruct* Hex as (x & y & HP & HQ).
Qed.
```

Instead of
`destruct Hex as [x [y [HP HQ]]]`

Could also do `destructs* 4 Hex`

# Forward Reasoning in TLC

- Forward reasoning made simpler through two tactics
  - *lets* `H: my_lemma arg1 … argn` — instantiate `my_lemma` as `H`
  - *forwards* `H: my_lemma [arg1 … argn]` — like `lets`, but introduce existentials
- Together with automation they get rid of a lot of boilerplate

```
Lemma wf_program'_eval:
  forall env p,
    wf_program' (dom env) p ->
    exists n, eval_program' env p = Some n.
Proof.
  introv Hwf.
  inductions Hwf; simpls*.
  forwards~ (n & ->): wf_expr_eval.
Qed.
```


UPPSALA
UNIVERSITET

# Other Brands Exist

- TLC is one alternative to the Coq standard library and tactics
  - https://www.chargueraud.org/softs/tlc
- The Iris project includes std++
  - https://gitlab.mpi-sws.org/iris/stdpp
- The SSReflect proof language uses a completely different style
  - https://inria.hal.science/inria-00407778/document

# Time for a break!

UPPSALA
UNIVERSITET

# The Locally Nameless Representation

# Representing Variables

- We typically represent variables by using their names

$$\lambda f \,.\, \lambda x \,.\, f\ x \qquad \lambda fst \,.\, \lambda snd \,.\, fst$$

$$\lambda x \,.\, \lambda y \,.\, y \qquad \lambda fst \,.\, \lambda snd \,.\, snd$$

- This is natural when using pen and paper, but brings formal issues:
  - Names do not mean anything, $\alpha$-equivalence instead of syntactic equality
  - The same name can be bound multiple times (shadowing)

$$\lambda x \,.\, \lambda y \,.\, x\ (\lambda x \,.\, x\ \ y)$$

$$x_1 \qquad x_1 \quad x_2\ x_2$$

# The Barendregt Convention

- In informal proofs we usually assume that all variables are distinct or can be renamed to avoid problems
- This is after Henk Barendregt who (quite literally) wrote the book on the lambda calculus
  - Barendregt in turn attributes this to Thomas Ottman
- When working in proof assistants, we are not this lucky…

# Capture-Avoiding Substitution

- When substituting names, $tm[x \mapsto u]$ we need to check for shadowing:

$$
\begin{aligned}
(x)[x \mapsto u] &= u \\
(y)[x \mapsto u] &= y, \text{ if } x \neq y \\
(\lambda x \,.\, tm)[x \mapsto u] &= \lambda x \,.\, tm \\
(\lambda y \,.\, tm)[x \mapsto u] &= \lambda x \,.\, (tm[x \mapsto u]), \text{ if } x \neq y \\
(tm_1 \ tm_2)[x \mapsto u] &= (tm_1[x \mapsto u]) \ (tm_2[x \mapsto u])
\end{aligned}
$$

- $\beta$-reduction can use substitution

$$(\lambda x \,.\, tm) \ v \to tm[x \mapsto v]$$

# Typing Environments

- A typing environment maps variables to types

$$\frac{x : T \in \Gamma}{\Gamma \vdash x : T} \qquad \frac{\Gamma, x : T_1 \vdash tm : T_2}{\Gamma \vdash \lambda x \,.\, tm : T_1 \to T_2}$$

- Assuming no duplicates, order in the environment doesn't matter

$$\Gamma_1, \Gamma_2 \vdash tm : T \implies \Gamma_2, \Gamma_1 \vdash tm : T$$

- Weakening is straightforward

$$\mathbf{dom}(\Gamma_1) \cap \mathbf{dom}(\Gamma_2) = \varnothing \implies$$
$$\Gamma_1 \vdash tm : T \implies \Gamma_2, \Gamma_1 \vdash tm : T$$

# Nameless Representation

- An alternative representation comes from Nicolaas de Bruijn
- A variable is an *index*, the distance to its $\lambda$-binder

$$\lambda f\,.\,\lambda x\,.\,f\ x \qquad\qquad \lambda fst\,.\,\lambda snd\,.\,fst$$
$$\lambda\,.\,\lambda.1\ 0 \qquad\qquad \lambda\,.\,\lambda.1$$

$$\boxed{\begin{array}{cc} \lambda x\,.\,\lambda y\,.\,y & \lambda fst\,.\,\lambda snd\,.\,snd \\ \lambda\,.\,\lambda.0 & \lambda\,.\,\lambda.0 \end{array}}$$

- Equivalent terms are syntactically equivalent! Shadowing not an issue:

$$\lambda x\,.\,\lambda y\,.\,x\ (\lambda x\,.\,x\ y)$$
$$\lambda\,.\,\lambda.1\ (\lambda.0\ 1)$$

# Substitution and Shifting

- When manipulating terms, we need to handle indices with care

$$
\begin{aligned}
(n)[n \mapsto u] &= u \\
(m)[n \mapsto u] &= m, \text{if } m \neq n \\
(\lambda\,.\,tm)[n \mapsto u] &= \lambda x\,.\,(tm[n+1 \mapsto \uparrow_0^1 u]) \\
(tm_1\ tm_2)[n \mapsto u] &= (tm_1[n \mapsto u])\ (tm_2[n \mapsto u])
\end{aligned}
$$

$$
\begin{aligned}
\uparrow_c^i (n) &= n + i, \text{if } n \geq c \\
\uparrow_c^i (n) &= n, \text{if } n < c \\
\uparrow_c^i (n)(\lambda\,.\,tm) &= \lambda x\,.\,(\uparrow_{c+1}^i tm) \\
\uparrow_c^i (n)(tm_1\ tm_2) &= (\uparrow_c^i tm_1)\ (\uparrow_c^i tm_2)
\end{aligned}
$$

- $\beta$-reduction becomes a lot trickier!

$$(\lambda\,.\,tm)\ v \rightarrow \uparrow_0^{-1} (tm[0 \mapsto \uparrow_0^1 v])$$

# Typing Environments

- A typing environment is a list of types

$$\frac{\Gamma[n] = T}{\Gamma \vdash n : T} \qquad \frac{T_1, \Gamma \vdash tm : T_2}{\Gamma \vdash \lambda\,.\,tm : T_1 \to T_2}$$

- The order in the environment very much matters!
- Weakening needs to update all indices

$$\Gamma_1 \vdash tm : T \implies \Gamma_2, \Gamma_1 \vdash \uparrow_0^{|\Gamma_2|} tm : T$$

- Our jobs as theoreticians gets harder!

# Middle Ground: Locally Nameless

- Good introduction in paper by Arthur Charguéraud (but concept older)
- Idea:
    - Use *nameless* representation for *bound* variables
    - Use *named* representation for *free* variables
- As soon as we "look under a $\lambda$", introduce a fresh variable

$$\{\} \vdash \lambda\,.\,\lambda.1\ 0$$

$$\{f\} \vdash \lambda\,.f\ 0$$

$$\{f, x\} \vdash f\ x$$

# Opening and Substitution

- Substitution of free variables no longer needs to care about shadowing!

$$
\begin{aligned}
(x)[x \mapsto u] &= u \\
(y)[x \mapsto u] &= y, \text{ if } x \neq y \\
(n)[x \mapsto u] &= n \\
(\lambda\,.\,tm)[x \mapsto u] &= \lambda\,.\,(tm[x \mapsto u]) \\
(tm_1\ tm_2)[x \mapsto u] &= (tm_1[x \mapsto u])\ (tm_2[x \mapsto u])
\end{aligned}
$$

- We substitute bound variables by *opening* terms, $tm^u = tm\{0 \mapsto u\}$

$$
\begin{aligned}
(n)\{n \mapsto u\} &= u \\
(m)\{n \mapsto u\} &= m, \text{ if } m \neq n \\
(x)\{n \mapsto u\} &= x \\
(\lambda\,.\,tm)\{n \mapsto u\} &= \lambda x\,.\,(tm\{n+1 \mapsto u\}) \\
(tm_1\ tm_2)\{n \mapsto u\} &= (tm_1\{n \mapsto u\})\ (tm_2\{n \mapsto u\})
\end{aligned}
$$

# Opening and Substitution

- Substitution of free variables no longer needs to care about shadowing!
- We substitute bound variables by *opening* terms, $tm^u = tm\{0 \mapsto u\}$

$$\begin{aligned}
(x)[x \mapsto u] &= u \\
(y)[x \mapsto u] &= y, \text{ if } x \neq y \\
(n)[x \mapsto u] &= n \\
(\lambda\,.\,tm)[x \mapsto u] &= \lambda\,.\,(tm[x \mapsto u]) \\
(tm_1\ tm_2)[x \mapsto u] &= (tm_1[x \mapsto u])\ (tm_2[x \mapsto u])
\end{aligned}$$

$$\begin{aligned}
(n)\{n \mapsto u\} &= u \\
(m)\{n \mapsto u\} &= m, \text{ if } m \neq n \\
(x)\{n \mapsto u\} &= x \\
(\lambda\,.\,tm)\{n \mapsto u\} &= \lambda x\,.\,(tm\{n+1 \mapsto u\}) \\
(tm_1\ tm_2)\{n \mapsto u\} &= (tm_1\{n \mapsto u\})\ (tm_2\{n \mapsto u\})
\end{aligned}$$

- $\beta$-reduction uses opening, but we can prove equivalence with substitution

$$(\lambda\,.\,tm)\ v \to tm^v$$

$$x \notin fv(tm) \implies tm^u = tm^x[x \mapsto u]$$

# Typing Environments

- Typing environments map variables to types, but we pick the names!

$$\frac{x : T \in \Gamma}{\Gamma \vdash x : T} \qquad \frac{\forall x \notin L \qquad \Gamma, x : T_1 \vdash tm^x : T_2}{\Gamma \vdash \lambda\,.\,tm : T_1 \to T_2}$$

- $\forall x \notin L$ is *cofinite quantification*: we show the rule for any $x$ not in some finite set $L$
- Since we can always pick fresh names, order in the environment doesn't matter

$$\Gamma_1, \Gamma_2 \vdash tm : T \implies \Gamma_2, \Gamma_1 \vdash tm : T$$

- Weakening is straightforward, as before

$$\mathbf{dom}(\Gamma_1) \cap \mathbf{dom}(\Gamma_2) = \varnothing \implies$$
$$\Gamma_1 \vdash tm : T \implies \Gamma_2, \Gamma_1 \vdash tm : T$$

# Locally Closed Terms

- Since we open terms by need, indices are never “out of bounds”
  - Terms like $\lambda.1$ are meaningless
- We can formalise terms being *locally closed* as the relation $lc$

$$\frac{}{lc\ x} \qquad \frac{lc\ tm_1 \quad lc\ tm_2}{lc(tm_1\ tm_2)} \qquad \frac{\forall x \notin L \quad lc\ tm^x}{lc(\lambda\,.\,tm)}$$

- With locally closed terms, we get a bunch of useful facts, for example

$$lc\ tm \implies tm^u = tm$$

$$lc\ u \wedge x \neq y \implies (tm^y)[x \mapsto u] = (tm[x \mapsto u])^y$$

$$lc\ u \implies (tm^v)[x \mapsto u] = (tm[x \mapsto u])^{v[x \mapsto u]}$$

# Questions before we go back to Coq?

# Conclusions

- Automation is what makes large proof developments feasible
  - Avoids spending time on trivial subgoals
  - Makes your development more robust to change
- Having a good standard library makes a big difference
  - Having to prove mundane things e.g. lists is surprisingly common
- Dealing with name binding and substitution is boring and technical
  - Locally nameless is **one** way to make definitions simpler
  - Having good library support helps a lot too!