Coq for natural language semantics day 1: Intro to Coq

Stergios Chatzikyriakidis

November 22, 2017

centre for linguistic theory and studies in probability

What this tutorial is all about

- The use of proof-assistants in the study of natural language semantics
  - More specifically, the use of the proof-assistant Coq
- Lecture 1
  - Historical notes on proof-assistants
  - Current state-of-the-art in proof-assistant technology
  - Introduction to Coq
    - ⋆ Some words on the logical framework behind Coq and Dependent Type Theories in general
    - ⋆ Installing Coq
    - ⋆ Working with Coq: Typing, Induction, pattern matching, basic theorem proving

CLASP centre for linguistic theory and studies in probability

What this tutorial is all about

- Lecture 2
  - Dependent Type Theories as foundational languages for formal semantics
    - ★ Some history and the state-of-the-art in Modern Type-Theoretical Semantics
  - Coq as an ideal vehicle for this type of semantics
    - ★ But also: Coq as an expressive platform to express any formal semantics theory
- Two uses of Coq with respect to NL semantics that correspond roughly to theorem proving and program verification
  1. Reasoning with Natural Language using theorem proving (inference as theorem proving)
  2. Verifying formal semantics accounts

CLASP centre for linguistic theory and studies in probability

What this tutorial is all about

- Lecture 3
  - Coq as a natural language reasoner
    - Doing simple natural language proofs
    - More complicated proofs based on the FraCaS test suite
    - The FraCoq system: Grammatical Framework meets Coq
    - The issue of automation
    - The way forward: automation, coverage and related work
  - Extra goodies
    - Some Type Theory with Records (TTR)
    - Some Montagovian Generative Lexicon
    - Some neo-Davidsonian semantics
    - Co-predication and individuation

CLASP centre for linguistic theory and studies in probability

Interactive theorem provers

- Roughly: piece of software that produces (or assists in the development of) formal proofs in collaboration (or alternatively under the guidance) with a human-agent.
- The idea goes back to the early 60s
  - The need for formally verified proofs
  - The AUTOMATH project (De Bruijn 1983, 1967 onwards)
    - ★ Aim: a system for the mechanic verification of mathematics
    - ★ Several AUTOMATH systems have been implemented
    - ★ The first system to practically exploit the Curry-Howard isomorphism

CLASP centre for linguistic theory and studies in probability

Interactive theorem provers

- Proof-assistant technology has gone a long way since then
  - Proliferation of proof-assistants implementing various logical frameworks
    - Classical logics/set theory (Mizar, Isabelle)
    - Constructive Type Theories (MTTs, Coq, Lego, Plastic, Agda among other things)
  - Important verified proofs
    - Four Colour Theorem (Gonthier 2004, Coq)
    - Jordan curve theorem (Kornilowicz 2005, Hales 2007, Mizar and HOL respectively)
    - The prime number theorem (Avigad et al 2007, Isabelle)
    - Feit-Thompson theorem (Gonthier et al. 2012, Coq (170.000 lines of code!))
  - Other uses: Software verification
    - CompCert: an optimized, formally verified compiler for C (Leroy 2013, Coq)
    - Coq in Coq (Barras 1997): Construct a model of Coq in Coq and show all tactics are sound w.r.t this model (verify the correctness of a system using the system itself)

The Coq proof-assistant

- INRIA project
  - Started in 1984 as an implementation of Coquand's Calculus of Constructions (CoC)
  - Extension to the Calculus of Inductive Constructions (CiC) in 1991
  - Coq offers a program specification and mathematical higher-level language called *Gallina* based on CiC
  - CiC combines both expressive higher-order logic as well as a richly typed functional programming language
- Winner of the 2013 ACM software system award
- A collection of 100 mathematical theorems proven in Coq: http://perso.ens-lyon.fr/jeanmarie.madiot/coq100/

CLASP centre for linguistic theory and studies in probability

The Coq proof-assistant

- An ideal tool for formal verification
  - Powerful and expressive logical language
  - Consistent embedded logic
  - Built-in proof tactics that help in the development of proofs
  - Equipped with libraries for efficient arithmetics in N, Z and Q, libraries about lists, finite sets and finite maps, libraries on abstract sets, relations and classical analysis among others
  - Built-in automated tactics that can help in the automation of all or part of the proof process
  - Allows the definition of new proof-tactics by the user
    - ★ The user can develop automated tactics by using this feature

CLASP centre for linguistic theory and studies in probability

Installing Coq

- Easy to install (http://coq.inria.fr/download)
- Use the installer or can get Coq via Macports or HomeBrew
- There is an interface for emacs, Proof General (provides support for a number of proof-assistants incl. Coq, Isabelle, HOL among others)
  - Get Proof-general here: `https://proofgeneral.github.io/`
  - Customize your emacs .init file according to the instructions in there

The Logical Language Behind Coq

- But first a bit of history on type theory by Thierry Coquand
  - Russell type theory 1903
  - Hilbert formulation of primitive recursion at higher types 1926
  - Brouwer intuitionistic logic, Kolmogorov's calculus of problems 1932
  - Gentzen natural deduction 1934
  - Church simplification of type theory, λ-calculus, 1940
  - Gödel system T and Dialectica Interpretation, 1941, 1958
  - Curry's discovery of the propositions-as-types principle 1958

The Logical Language Behind Coq

- But first a bit of history on type theory by Thierry Coquand
  - Prawitz natural deduction 1965
  - Tait normalization proof for system T 1967
  - de Bruijn Automath 1967
  - Howard general formulation of proposition-as-types 1968
  - Scott Constructive Validity 1968 (strongly inspired by Automath)
  - Lawvere equality in hyperdocrtrines 1970
  - Martin Lf predicative system 1972, 1973 (formulation of the rules for identity)
  - Girard system F and normalization proof 1970
  - Girard's paradox 1971
  - Martin Löf predicative system 1972, 1973 (formulation of the rules for identity)
  - Martin Löf extensional" type theory 1979. Bibliopolis book, 1984 (available on-line) still extensional" version

CLASP centre for linguistic theory and studies in probability

The Logical Language Behind Coq

- Calculus of Constructions (CoC) (Coquand and Huet 1984)
- Calculus of Inductive Constructions (CiC) (Coquand and Paulin-Mohrings 1988)
- Extended Calculus od Constructions (Luo 1992)
- ...
- Homotopy Type Theory (Voevodsky 2009 (RIP))
- Cubical Type Theory (Coquand et al. 2016)

CLASP centre for linguistic theory and studies in probability

The Logical Language Behind Coq

- Calculus of Inductive Types
  - A type theory with dependent typing and inductive types
  - Can be seen as a programming language and/or a foundational language for constructive mathematics
  - Proof-theoretically specified
  - Includes three universes: Prop, Set and Type (propositions, the universe of specifications and a bigger universe including the previous two)
    - Prop is impredicative, the other two universes predicative
    - Subtyping is supported
- Let us see all these features in action!

One important last diversion: Installing Coq

- Easy to install (http://coq.inria.fr/download)
- Use the installer or can get Coq via Macports or HomeBrew
- There is an interface for emacs, Proof General (provides support for a number of proof-assistants incl. Coq, Isabelle, HOL among others)
  - Get Proof-general here: `https://proofgeneral.github.io/`
  - Customize your emacs .init file according to the instructions in there

Basics of Coq

- Typing
  - All objects have a type in Coq
    - ★ All the pre-defined objects in Coq can be checked for type using the command *check*
    - ★ For example the type *nat* of natural numbers has type *Set* (*nat* : *Set*), while natural numbers like 1,2,3 and so on, type nat (1 : *nat*).

```
Coq < Check nat.
nat:Set
Coq <Check 1.
1:nat
```

- Function application
  - Applying a function to an argument
    - ★ The addition function is of type $nat \rightarrow nat \rightarrow nat$, takes two nat arguments and also returns a *nat* argument

```
Coq < Check plus.
plus:nat -> nat -> nat
Coq < Check plus 3 4.
3 + 4:nat
```

CLASP centre for linguistic theory and studies in probability

Basics of Coq

- Declarations
  - Associating a name with a specification
  - Specifications classify the object declared
    - ★ Well-founded typing hierarchy of sorts: *Prop*, *Set* and *Type*, logical propositions, mathematical collections of objects and abstract types
    - ★ We can declare new types either by *Parameter* or via *Variable*
    - ★ We can restrict the scope by using local contexts, using *section.*

```
Coq < Variable H:Set.
H is assumed
Warning: H is declared as a parameter because it is at
a global level
Coq < Parameter H:Set.
H is assumed
Coq < Section section.
Coq < Variable H1:Set.
H1 is assumed
```

  - The type Type is of type Type (but of a higher universe, $Type_n : Type_{n+1})$ Girard's paradox is avoided, there is no impredicativity

CLASP centre for linguistic theory and studies in probability

Basics of Coq: A note on the universes of Coq

- Terms have types and these types are themselves terms and so on
  - CiC has an infinite array of sorts, called *universes* (sort: type of a type)
    - ⋆ Universes are stratified, predicative, so no self-referential vicious circles arise
    - ⋆ The exception is the universe of propositions, which is impredicative (you can have a *Prop* quantifying over other any *Prop*)
    - ⋆ One impredicative universe of propositions and infinitely many predicative ones

CLASP centre for linguistic theory and studies in probability

Basics of Coq

- Definitions
  - *Definition* ident : *term*1 := *term*2
  - It checks that the type of *term*2 is definitionally equal to *term*1, and registers *ident* as being of type *term*1, and bound to value *term*2.
  - We can define a constant *three* to be the successor of the successor of the successor of 0 (the successor is pre-defined).

    ```
    Definition three:nat:= S (S(S((0))).
    ```

  - Coq can infer the type in these cases, so it can be dropped:

    ```
    Definition three:= S (S(S((0))).
    ```

  - Defining functions
    - ★ Square number function
    - ★ Uses λ abstraction. Takes a *nat* to return a *nat*

      ```
      Definition square:= fun n:nat=> n*n.
      ```

CLASP centre for linguistic theory and studies in probability

Basics of Coq

- Inductive types
  - Inductive types without recursion
    - ★ The inductive type for booleans
    - ★ Pre-defined in Coq in the following manner:

```
Coq < Inductive bool : Set := true | false.
bool is defined
bool_rect is defined
bool_ind is defined
bool_rec is defined
```

    - ★ The above introduces a new *Set* type, *bool*. Then the constructors of this *Set* *true* and *false* are declared, and three elimination rules are provided, allowing to reason with this type of types
    - ★ The bool_ ind combinator for example allows us to prove that every $b : bool$ is either *true* or *false* (more on this later)

CLASP centre for linguistic theory and studies in probability

Basics of Coq

- Inductive types
  - Inductive types with recursion: Natural numbers

```
Coq < Inductive nat : Set :=
| O : nat
| S : nat -> nat.
nat is defined
nat_rect is defined
nat_ind is defined
nat_rec is defined
```

  - Recursive types are closed types
    - Their constructors define all the elements of that type
    - Peano's induction axiom (*nat_ind*) as well as general recursion is defined (*nat_rec*)

An example of a simple proof

- Transitivity of implication: $(P \to Q) \to (Q \to R) \to (P \to R)$
- What is needed before we get into proof-mode
  - Declaring P, Q, R as propositional variables (only elements of type Prop can be the arguments of logical connectives)

    ```
    Variables P Q R:Prop.
    ```

  - With this declaration at hand, we can get into proof-mode:

    ```
    Theorem trans: (P->Q)->(Q->R)->(P->R)
    ```

  - *intro* tactic: introduction of $(P \to Q)$, $(Q \to R)$ and P as assumptions

    ```
    1 subgoal

    H : P -> Q
    H0 : Q -> R
    H1 : P
    ============================
    R
    ```

CLASP centre for linguistic theory and studies in probability

An example of a simple proof in Coq

- The *apply* tactic: It takes an argument which can be decomposed into a premise and a conclusion (e.g. $Q \rightarrow R$), with the conclusion matching the goal to be proven (R), and creates a new goal for the premise (basicall *modus ponens*)

```
H : P -> Q
H0 : Q -> R
H1 : P
============================
Q
```

- We now use *apply* for H

```
H : P -> Q
H0 : Q -> R
H1 : P
============================
P
```

An example of a simple proof in Coq

- The tactic *assumption*: matches a goal with an already existing hypothesis. Applying *assumption* completes the proof

```
1 subgoal

H : P -> Q
H0 : Q -> R
H1 : P
============================
P
trans < assumption.
Proof completed.
```

- Alternatively one can use the *exact* tactic.

An example of a more complicated proof in Coq

- Peirce's law: If the law of the excluded middle holds, then so is the following: $((A \to B) \to A) \to A$
  - We formulate in Coq notation:

    ```
    Definition lem:= A \/ ~ A.
    Definition Peirce:= ((A->B)->A)->A.
    Theorem lemP: lem -> Peirce.
    ```

  - We first use unfold to unfold the definitions. So *lem* and *Peirce* will be substituted by their definition

    ```
    lemP < unfold lem.
    1 subgoal
    ============================
    A \/ ~ A -> Peirce
    lemP < unfold Peirce.
    1 subgoal
    ============================
    A \/ ~ A -> ((A -> B) -> A) -> A
    ```

CLASP centre for linguistic theory and studies in probability

An example of a more complicated proof in Coq

- Applying intro twice (we can use intros to apply intro as many times possible)

```
lemP < intros.
1 subgoal
H : A \/ ~ A
H0 : (A -> B) -> A
============================
A
```

- We can now use the *elim* tactic on *H*, basically using the elimination rules for disjunction:

```
H : A \/ ~ A
H0 : (A -> B) -> A
============================
A -> A
subgoal 2 is: ~A -> A
```

CLASP centre for linguistic theory and studies in probability

An example of a more complicated proof in Coq

- We use intro and assumption and the first subgoal is proven

```
lemP < intro. assumption.
H : A \/ ~ A
H0 : (A -> B) -> A
============================
~A -> A
```

- Intro and apply H0

```
lemP < intro. apply H0.
H : A \/ ~ A
H0 : (A -> B) -> A
H1 : ~ A
============================
A -> B
```

centre for linguistic theory and studies in probability

An example of a more complicated proof in Coq

- *Intro* and *absurd A*:

```
lemP < absurd A.
2 subgoals
H : A \/ ~ A
H0 : (A -> B) -> A
H1 : ~ A
H2 : A
============================
~ A
subgoal 2 is:
A
```

CLASP centre for linguistic theory and studies in probability

An example of a more complicated proof in Coq

- *Absurd A* proves the goal from *False* and generates to subgoals, *A* and *not A*
- Using assumption twice, the proof is completed

```
lemP < assumption. assumption.
1 subgoal
H : A \/ ~ A
H0 : (A -> B) -> A
H1 : ~ A
H2 : A
============================
A
Proof completed.
```

CLASP centre for linguistic theory and studies in probability

Proof tactics

- We discuss some of the basic predefined Coq tactics
- Following Chipalla (2014) we categorize these according to the connective involved in each case
  - Conjunction
    - *elim*: Use of the elimination rule
    - *split*: Splits the conjunction into two subgoals
    - Examples:

      ```
      Theorem conj: A/\B->A.
      Theorem conj: B/\(A/\C)->A/\B.
      ```
  - Disjunction
    - *Elim:* elimination rule
    - *Left,Right:* deals with one of the two disjuncts

      ```
      Theorem disj: (B\/(B\/C))/\(A\/B)->A\/B.
      ```
  - *Implication* ($\Rightarrow$) and *Forall*
    - *intro(s)*
    - *apply*

Proof tactics

- We discuss some of the basic predefined Coq tactics
- Following Chipalla (2014) we categorize these according to the connective involved in each case
  - Existential
    - *exists t*: instantiates an existential variable
  - Equality (=)
    - *reflexivity, symmetry, transitivity*: the usual properties of equality
    - *congruence*: used when a goal is solvable after a series of rewrites
    - *rewrite, subst*: rewrites an element of the equation with the other element of the equation. *Subst* is used when one of the terms is a variable

CLASP centre for linguistic theory and studies in probability

Proof tactics - exists, elim

- Imagine we want to prove the following:

```
Parameter P: nat -> Prop.
Theorem EXISTS: P 5-> exists n: nat, P n.
```

- We can use the tactic *exists* to substitute 5 for *n* and prove the goal

CLASP centre for linguistic theory and studies in probability

Proof tactics - reflexivity,symmetry,transitivity

- Reflexivity: Any time a goal of the form A=A needs to be proven for A: Type.
- Symmetry. Any time a goal of the form A=B needs to be proven from a hypothesis B=A, for A B: Type
- Transitivity. Any time a goal A=C needs to be proven from hypotheses A=B and B=C for A,B,C:Type

```
Theorem SRT: forall n m n1: nat,  n=m/\m=n1-> n=n /\ n=n1
```

Proof tactics:

- *idtac*: does nothing
  - Useful as part of composite tactics where we want to apply some tactic to some parts and leave some others untouched
- Simiarly the *fail* tactic can be used after a tactic, in case we want the tactic to leave no further goal and to abort if it does
- *assert*: adding consequences explicitly
- *generalize*: generalizes the conclusion with respect to some term.

CLASP centre for linguistic theory and studies in probability

Proof tactics: Cut

- This is a very useful tactic
  - Say we want to prove P
  - We have two solutions:
    - $t_1 for Q \rightarrow P$
    - $t_2 for Q$
    - the application of $t1$ $t2$ proves P
    - the *cut* tactic provides this combined reasoning step

CLASP centre for linguistic theory and studies in probability

Proof tactics - Induction tactics

- *induction*: *induction x* decomposes the goal statement to a property applying to x and then applies *elim x*
- *elim*: Similar tactic, does not add hypotheses in the context
- An example using inductive types. We define the inductive type season, consisting of four members, corresponding to each season:

```
month1 < Inductive season:Set:= Winter|Spring|Summer|
Autumn.
season is defined
season_rect is defined
season_ind is defined
season_rec is defined
```

- Coq automatically adds several theorems that make reasoning about the type possible. In the case above these are season_ rect season_ ind and season_ rec

Proof tactics - Induction tactics

- season_ ind provides the induction principle associated with an inductive definition. In this case this amounts to:

```
month1 < Check season_ind.
season_ind
:forall P : season -> Prop,
P Winter -> P Spring -> P Summer ->
P Autumn -> forall s : season, P s
```

- Universal quantification on a property P of seasons, followed by a succession of implications, each premise being P applied to each of the seasons. The conclusion says that P holds for all seasons

Proof tactics - Induction tactics

- Let us say we want to prove the following:

  ```
  SEASONEQUAL < Theorem SEASONEQUAL: forall s: season,
  s=Autumn\/s=Winter\/s=Spring\/s=Summer.
  ```

- We apply *intro* and call *elim*

  ```
  s : season
  ============================
  Autumn = Autumn \/ Autumn = Winter \/ Autumn = Spring \/
  Autumn = Summer
  Winter = Autumn \/ Winter = Winter \/ Winter = Spring
  \/ Winter = Summer
  Spring = Autumn \/ Spring = Winter \/ Spring = Spring
  \/ Spring = Summer
  Summer = Autumn \/ Summer = Winter \/ Summer = Spring
  \/ Summer = Summer
  ```

- Can be easily proven using *left*, *right* and *reflexivity* or using *auto*.

CLASP centre for linguistic theory and studies in probability

Automation tactics

- Tactics that are a combination of more simple tactics, in effect a macro of tactics
  - Used to automate parts or the whole proof
  - Examples of such tactics
    - ★ The *auto* tactic: Provides automation in case a proof can be found by using any of the tactics:*intros, apply, split, left, right and reflexivity*
    - ★ The *eauto* tactic: A variant of *auto*. Uses tactics that are variants of the tactics used in *auto*, the only difference being that they can deal with conclusions involving existentials (for example *eapply*, functions like *apply* but further introduces existential variables)

centre for linguistic theory and studies in probability

Automation tactics

- An example exemplifying the difference between *auto* and *eauto*
  - We define a predicate nat_ predicate and then create a theorem:

    ```
    Parameter nat_predicate: nat->Prop.
    Theorem NATPR: nat_predicate(9) -> exists n: nat,
    nat_predicate(n).
    ```

  - Due to the existential, *auto* cannot prove the above, while *eauto* can
- However, the following can be proven by *auto* as well:

```
Variable j:nat.
Let h:= j.
Theorem NATPR: nat_predicate(j) -> nat_predicate(h)\/
exists n:nat, nat_predicate(n).
```

  - In effect, the existential does not have to be dealt with, only the left disjunct is used
    - *Eauto* cannot however open up existentials or conjunctions from context. This is made possible with another tactic called *jauto* (see next lecture)

CLASP centre for linguistic theory and studies in probability

Automation tactics

- The tactics *tauto*, *intuition*
  - The first is used for propositional intuitionistic tautologies
  - The latter for first-order intuitionistic logic tautologies

```
Coq < Theorem TAUTO: A\/B->B\/A.
1 subgoal

============================
A \/ B -> B \/ A

TAUTO < tauto.
Proof completed.
```

CLASP centre for linguistic theory and studies in probability

Imported modules

- A number of other more advanced tactics can be used by importing different Coq packages
  - E.g. the *Classical* module can be imported, which includes classical tautologies rather than intuitionistic

    ```
    Theorem CLASSICAL: not (not A)-> A.
    ```
  - The *Omega* module can be used in order to deal with goals that need Presburger arithmetic in order to be solved

    ```
    Theorem neq_equiv : forall x:nat, forall y:nat, x <> y <->
    x < y  \/ y < x.
    ```
    - ★ Includes tactics to deal with basic algebraic structures like *rings*, *fields* etc.

CLASP centre for linguistic theory and studies in probability

Imported modules: LibTactics (Lib.v in your files)

- *Libtactics* is a collection of advanced tactics, basically advanced variations of the standard tactics
  - For example, the *destructs* tactic is the recursive application of the *destruct* tactic

    ```
    Theorem DESTRUCTS: (A/\B/\C/\D)->B.
    ```
  - jauto: helps in automation but more nuanced than the built-in auto and euato tactic
    - Able to deal with cojunctions, disjunctions and existentials in both goal and assumptions

Imported modules: LibTactics (Lib.v in your files)

- The tactic false: like *exfalso*, i.e. substituting goals with False and trying to prove False
  - Does more than that: it will prove the goal in case it involves absurd assumptions or contradictory assumptions

CLASP centre for linguistic theory and studies in probability

Interim Note on Proof Handling

- Local sections: *Section x*
- *Qed* in case no more subgoals exist (declared as a theorem)
- *Abort* to abort the proof process
- *Restart* to start over
- *Admitted* giving up and declaring the goal as an axiom (there is some uses of this for NL!)

Pattern Matching

- Let us imagine an inductive data type month with 12 members (similar case to the *seasons* example):

  ```
  Inductive month:Set:= January|February|March|April|May|
  June|July|August|September|October|November|December.
  ```

- In such cases, one can compute values according to the element we are interested in in the type
  - Pattern matching on the enumerated type month
  - For example, let us define the days individual months have
  - For simple pattern matching, we use *Definition*

    ```
    Definition nbdays (m:month) :=
    match m with
    |April => 30| June => 30|September=> 30|November=> 30
    |February=> 28| _ => 31 end.
    ```

CLASP centre for linguistic theory and studies in probability

Pattern Matching

- Let us check the type, it is a function *month* $\rightarrow$ *nat*
- It takes a *month* argument and returns a natural number according to the what we have defined
- Checking that it works (Compute nbdays $<$ *argument* $>$)

```
Compute nbdays July.
  = 30
: nat
```

CLASP centre for linguistic theory and studies in probability

Pattern Matching

- Let us define a function that returns true in case a month is a winter month, false otherwise
  - Think about it for a minute

CLASP centre for linguistic theory and studies in probability

Pattern Matching

- Let us define a function that returns true in case a month is a winter month, false otherwise
  - Think about it for a minute

```
Definition is_winter_month (m:month) :=
match m with
|December => True
| January => True
|February => True
|_ => False
end.
```

CLASP centre for linguistic theory and studies in probability

Useful tactic for pattern matching: case

- Takes as argument a term that is part of an inductive type
- Replaces all instances of the term with all possible cases

```
Theorem monthcase: forall m: month, le nbdays m  28.
```

- This can be proven using *case* on m
  - Another useful tactic is *simpl* for this example
    - Variety of uses: e.g. performs ι reduction, one is to require computation of a case (for example *nbdays July* with simpl returns *31*)

Another useful tactic for pattern matching: discriminate

- Reasoning with contradictive equalities
  - A goal of the form $a_1 = a_2$ where a_1, a_2 are both constructors of the same inductive type
    - Discriminate solves goals of this kind
  - An example to illustrate the point: the inductive type *next_month*

```
Definition next_month (m:month) :=
match m with
| January => February | February => March
| March => April | April => May
| May => June  | June => July
| July => August | August => September
| September => October | October => November
| November => December | December => January
end.
```

  - the *discriminate* tactic solves a goal when a false equality exists in the context (a special case of *exfalso quodlibet*)

CLASP centre for linguistic theory and studies in probability

Another useful tactic for pattern matching: discriminate

- A clearer example

```
Inductive bool: Set :=
| true
| false.

Lemma incorrect_equality_implies_anything: forall a,
 false = true -> a.
```

CLASP centre for linguistic theory and studies in probability

Another useful tactic for pattern matching: injection

- " if H states an equality of two terms using the same constructor, then injection adds to the context the equalities implied by the fact that the constructors are injective functions" [From Chipalla's online Coq Tactics Quick Reference]

CLASP centre for linguistic theory and studies in probability

Pattern Matching on Recursive Functions

- Doing pattern matching on recursive functions, e.g. on nat, requires the use of *Fixpoint*
  - For example take a look at the way addition is defined in Coq

```
Fixpoint plus n m :=
match n with
| O => m
| S n' => S (plus n' m)
end.
```

  - Here is another one for subtraction (this is actually very neat)

```
Fixpoint minus (n m : nat) : nat :=
match n, m with
S p, S q => minus p q
| _, _ => n
end.
```

CLASP centre for linguistic theory and studies in probability

Pattern Matching: an example from morphology

- Let us see an example from moprhology
  - Define a function that will return the correct 3sg pronoun form depending on gender and case
  - First, let us define the pronouns as parameters of type e.

    ```
    Parameter he she it him her: e.
    ```

  - Now, we define the enumerarated types for gender and case.

    ```
    Inductive Gender:Set:= Fem|Masc|Neut.
    Inductive Case:Set:= Nom|Acc.
    ```

  - Now, we can pattern match:

    ```
    Definition  pronoun (g: Gender) (c:Case):=
    match g, c with
    |Fem, Nom=> she
    |Fem, Acc => her
    |Masc, Nom=> he
    |Masc, Acc=> him
    |Neut, _=> it
    end.
    ```

CLASP centre for linguistic theory and studies in probability

Record types

- Inductive types with one constructor
- Can bundle pieces of data together in a single type
- For example, the following record defines plane

```
Record plane: Set := point {abscissa: Z; ordinate: Z}.
```

- single constructor *point* with two fields *abscissa* and *ordinate*

Record types

- Records can be dependent
  - Fields may depend on other fields
  - For example, let us define a more structured Entity type, which has two fields, one having an element of type e and one where the predicate human holds of this element

    ```
    Record Entity : Type := mkentity {x: e; z: human x}.
    ```
  - People familiar with Cooper's TTR will recognize the potential for expressing this type of semantics in Coq
    - ★ More on this in the next days

CLASP centre for linguistic theory and studies in probability

Subtyping

- Coq supports subtyping
  - The subtyping used is coercive subtyping (quite close to the one used by Luo)
  - Here is an example of subtyping between types *Man* and *Human*

```
Parameters Man Human: CN.
Axiom mh: Man -> Human.
Coercion mh: Man >-> Human.
```

Subtyping

- Here is another example from the Coq manual

```
Definition bool_in_nat (b:bool) := if b then 0 else 1.
Coercion bool_in_nat : bool >-> nat.
```

- With this coercion one can put a bool and a nat in an equality without Coq complaining about it!

```
Check (0 = true).
0 = true
: Prop
```

CLASP centre for linguistic theory and studies in probability

Advanced topic for people interested: Co-inductive types

- This is not something that can be covered here
  - But, it deserves a mention
  - Co-inductive types are similar to inductive types
    - terms should be obtained by recursive uses of the constructors
  - But:
    - There is no induction principle
    - The branches in the types can be infinite
  - Use of co-recursive functions
    - "The terms these functions produce may be infinite, but as long as we require only to see a finite part of these terms, these functions only need to perform finite computations" (Bertot and Casteran 2004: 348)

Advanced topic for people interested: Co-inductive types

- An example: streams (infinite lists)

```
CoInductive Stream (A:Set): Set :=
Cons : A -> Stream A -> Stream A.

CoInductive LList (A:Set) : Set :=
LNil : LList A
| LCons : A -> LList A -> LList A.
```

- The first is an infinite list (there is no nil constructor)
- The second one (lazy list) is the output of a process that can be either finite or infinite

CLASP centre for linguistic theory and studies in probability

Advanced topic for people interested: Co-inductive types

- More on lazy lists, see:
  - Bertot and Casteran (2004)
  - The file bertot_co-recursion.pdf