Modern Type Theoretical semantics: Reasoning using proof-assistants

Stergios Chatzikyriakidis (based on joint work with Zhaohui Luo)

March 9, 2016

Interactive theorem provers

- Started in 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

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/

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

Installing Coq

- Easy to install (http://coq.inria.fr/download)
- Use the installer
  - The code in this tak is compatible with the earlier version of Coq 8.3 rather than 8.4
    - 8.4 version has some minor improvements that lead to minor incompatibilities with the earlier version
    - Download the earlier version if you want to directly use the code (version 8.3) (http://coq.inria.fr/coq-8.3)
    - If you feel adventurous, read the differences pertaining to the new version, and revise code accordingly
    - You can get Coq via Macports or HomeBrew
    - There is a for emacs, Proof General (provides support for a number of proof-assistants incl. Coq, Isabelle, HOL among others)

Theorem proving in Coq

- Coq is an interactive theorem prover: the user drives the prover to the proof
  - How it works: A theorem is declared with the command *Theorem*, followed by the name of the theorem we want to prove, followed by the theorem itself
    - $Theorem\ x : a \Rightarrow b$
    - The goal is to reach a complete proof using the proof tactics provided by the assistant

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 $nat \to nat \to nat$, 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
```

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

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.
      ```

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)

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
    ```

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.

```
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.
```

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
```

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
```

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
```

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
```

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.
```

Other useful 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*

Other useful 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
    - *elim*: Elimination rule

      ```
      Theorem NAT: exists x: nat, le 0 x.
      ```
  - 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

Other useful 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

Other useful 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

Other useful 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*.

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)

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)

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.
```

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 <->
    ```
  - *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.
      ```

MTT semantics in Coq

- Encoding MTT semantics based on theoretical work using Type Theory with Coercive Subtyping in Coq
  - Coq is a natural toolkit to perform such a task
    - The type theory implemented in Coq is quite close to Type Theory with Coercive Subtyping
    - Thus, the TT does not need to be implemented!
    - What we need, is a way to encode the various assumptions as regards linguistic semantics and then reason about them

The CN universe

- Common nous in MTTs are seen as types rather than predicates
- Zhaohui Luo proposed the introduction of a universe of CN interpretations (Luo 2011, 2012 among others)
  - A collection of the names of types that interpret common nouns
  - Coq does not support universe construction
    - Only the pre-defined universes can be used
    - In this sense, we define *CN* to be Coq's pre-defined *Set* universe

```
Definition CN:= Set.
Parameters Man Human Animal Object:CN
```

Subtyping relations

- In order for type many-sortedness to have any advantages over more coarse grained typing (like the e typing in MG), a subtyping mechanism is needed
  - We have already seen the use of coercive subtyping as an adequate subtyping mechanism
  - Coq uses a similar mechanism (albeit with minor formal differences)
  - Subtyping in Coq is also based on the notion of coercion.
    - An example is shown below:

```
Axiom MH: Man->Human. Coercion MH: Man>->Human.
Axiom HA: Human->Animal. Coercion HA: Human>->Animal.
```

Types for verbs

- The type of propositions is identified with *Prop*.
  - Verbs are function types returning a *Prop* type once one or more (depending on valency) arguments have been provided
    - However, given type many sortedness the arguments needed for individual verbs will be dependent on the specific verb in each case
    - Thus, Walk will be specified as $Animal \rightarrow Prop$ while fall as $Object \rightarrow Prop$

```
Parameter walk: Animal-> Prop.
Parameter fall: Object-> Prop.
```

Quantifiers, adjectives, adverbs

- Following work by Luo (2011, 2012) and Chatzikyriakidis and Luo (2013a,b,2014), quantifiers are given an inductive type, taking an $A : CN$ argument and returns the type $(A \rightarrow Prop) \rightarrow Prop$
- Adjectives are defined as simple predicates.
- VP adverbs are defined as predicate modifiers extending over the universe CN, while sentence adverbs as functions from propositions to propositions

```
Parameter some: forall A:CN, (A->Prop)->Prop
Parameter handsome: Human -> Prop
Parameter slowly: forall A:CN, (A->Prop)->(A->Prop).
Parameter fortunately: Prop ->Prop.
```

Quantifiers, adjectives, adverbs

- More must be said about the lexical semantics of all these categories.
- For example, in the case of *some* the following will be assumed
    - Same typing but has further information on the lexical semantics of some (i.e. existential quantification)

      ```
      Definition some:= fun A:CN, fun P:(A->Prop)=> exists x: A,
      P(x).
      ```

    - More will be said about the lexical semantics as we proceed

Adjectival modification using dependent record types

- Intersective and subsective adjectival modification have been treated as involving Σ types.
- This is the analysis we follow here
  - We however follow Luo (2012) and use dependent record types instead of Σ types (which are equivalent)
    - The first projection is declared as a coercion
    - Thus, for *handsome man*, we get the inference *man*

```
Record handsomeman:CN:=mkhandsomeman{ c :>Man;_: handsome c }.
```

Reasoning with NL

- As soon as NL categories are defined, Coq can be used to reason about them
  - In effect, we can view a valid NLI as a theorem
    - ★ Thus, we formulate NLIs as theorems
    - ★ The antecedent and consequent must be of type *Prop* in order to be used in proof mode
    - ★ Thus, the first can be formulated as a theorem, but not the second:

```
Theorem EX:(walk) John-> some Man (walk).
Theorem WA:walk -> drive.
```

Reasoning with NL

- The same tactics that can be used in proving mathematical theorems are used for NL reasoning
  - The aim is to predict correct NLIs while avoiding unwanted inferences
    - ★ For example, given the semantics for quantifier *some*, one can formulate the following theorem and further try to prove it

```
Theorem EX:  (walk) John-> some Man (walk).
```

An NLI example

- One of the inferences we should be able to get when a proper name acts as an argument of the verb is one where an element of the same type as the proper name acts as the argument of the same verb
  - Basically, from a sentence like *John walks*, we should infer that *a man walks*
  - We formulate the theorem

    ```
    Theorem EX:  (walk) John-> some Man (walk).
    ```

  - We unfold the definition for *some* and use *intro*

    ```
    EX < intro.
    1 subgoal
    H : walk John
    ============================
    exists x : Man, walk x
    ```

  - We use the exists tactic to substitute x for *John*. Using *assumption* the theorem is proven

An NLI example

- To the contrary, we should not be able to prove the opposite

```
Theorem EX: some Man (walk) -> (walk) John.
```

- Indeed, no proof can be found in this case.
  - We unfold *some* and use *intro*

```
EX < intro.
1 subgoal
H : exists x : Man, walk x
============================
walk John
```

  - From this point on, we can use any of the *elim, induction, case* tactics but at the end we reach a dead end

```
EX < intro.
1 subgoal
H : exists x : Man, walk x
x : Man
H0 : walk x
============================
walk John
```

Automation?

- From a theoretical point a view, having a system that can reason about NL semantics in such a straightforward way is already something
  - From the practical side however, in order to develop something like this into a more practical device, automation needs to be possible
    - ★ For the simple case we have been discussing, automation is possible once we unfold the definition for *some*
    - ★ The tactic *eauto* will solve the theorem in one step

```
EX< unfold some.
1 subgoal
============================
walk John -> exists x : Man, walk x
EX < eauto.
Proof completed.
```

    - ★ Still, this is not yet full automation. What can we do?

The tactic language Ltac

- Besides the predefined tactics offered by Coq or these imported by various Coq packages, Coq offers a way for the user to define his own proof-tactics
  - This is achieved by Ltac
    - A programming language inside Coq that can be used to build new user-defined tactics
    - Using Ltac we can define the following tactic that will fully automate the example we are interested in

      ```
      Ltac EXTAC:= cbv; eauto.
      ```

    - The *cbv* tactic performs all possible reductions using δ, β, ζ and ι
    - In our case, δ reduction is applied first unfolding the definition and then β reduction
    - The tactic *compute* that embodies *cbv* can also be used

The tactic language Ltac

- What we need is automated tactics that work for a range of examples and not tactics that work on a case by case basis
  - For example, the tactic *EXTAC*, though simple enough, has the power to automate quite a few inferences
    - One can further prove:

      ```
      all Man (walk)->walk John.
      all Man (walk)->walk John->some Man (walk).
      ```

    - Also cases where subtyping is involved, like the following:

      ```
      all Animal (walk)->walk John.
      ```

The tactic language Ltac

- However, the following cannot be proven with *EXTAC*:

  `all Man (walk)-> some Man (walk).`

  - We get the following error:

    ```
    Coq < Theorem EX2: all Man (walk) -> some Man (walk).
    1 subgoal
    ============================
    all Man (fun x : Man => walk x) -> some Man (fun x : Man =>
     walk x)
    EX2< EXTAC.
    No more subgoals but non-instantiated existential variables
    Existential 1 = ?463 : [H : forall x : Man, walk x |- Man]
    ```

  - This means that *eauto* did not manage to instantiate an existential, which was then eliminated by a computation
  - The solution is to instantiate the value "manually"

The tactic language Ltac

- For example, we can substitute x for *John* using the *exists* tactic
  - We can define a similar tactic that instantiates the variable using *exists* and then calls *EXTAC*

    ```
    Ltac EXTAC1 x:= cbv; try exists x;EXTAC.
    ```
  - The command *try* + *tactic*, tries to perform the tactic, and if it fails, it moves on
  - This will suffice to prove automatically all the NL examples we have considered so far

NL Inference

- The task of determining whether an NL hypothesis H can be deduced from an NL premise P
- A central task in both theoretical and computational semantics
  - As Cooper et al. (1996) aptly put it: "inferential ability is not only a central manifestation of semantic competence but is in fact centrally constitutive of it"
    - Inferential ability as the best way to test the semantic adequacy of NLP systems
    - An adequate NLP system should be able to predict correct inferences like (1)-(3) without further generating unwanted inferences like (4) or (5)

(1) John walks and Mary talks $\Rightarrow$ Mary talks

(2) Some men run fast $\Rightarrow$ Some men run

(3) John walks $\Rightarrow$ Some one walks

(4) John walks and Mary talks $\not\Rightarrow$ If John walks, Mary talks

(5) No men run fast $\not\Rightarrow$ No men run

NL inference platforms: FraCas

- Platforms for NLI - The Fracas test suite
  - Came out of the FraCas consortium, a large collaboration in the 90's to create resources for computational semantics
  - Contains 349 NLIs, with one or more premises
    - Categorized by semantic section: e.g. Quantifiers, adjectives temporal reference etc.
    - A number of premises (usually single premised), followed by the hypothesis in the form of a question

The FraCas test suite

- Typical examples

(6) No delegate finished the report.
Did any delegate finished the report on time? [No] (quantifier section)

(7) Either Smith, Jones or Anderson signed the contract. Did John sign the contract? [UNK] (plurals)

(8) Dumbo is a large animal. Is Dumbo a small animal? [NO] (adjectives)

(9) Smith believed that ITEL had won the contract in 1992. Did ITEL win the contract in 1992? [UNK] (Attitudes)

Formulating the examples

- As already said, the examples involve a number of premises, followed by a question (h).
  - We reformulate the examples as involving declarative forms in Coq (this is a usual approach, at least with deep approaches)
    - In cases of *yes* in the FraCas test suite, we formulate a declarative hypothesis as following from the premise
    - In cases of *no*, we formulate the negation of a declarative hypothesis as following from the premise
    - In cases of *UNK*, for both the positive and the negated h, no proof should be found. If it is, we overgenerate inferences we do no want

Formulating the examples

- A *YES* example

(10) A Swede won the Nobel Prize.
Every Swede is Scandinavian.
Did a Scandinavian win the Nobel prize? [Yes, FraCas ex. 3.49]

```
Theorem SWE:(a Swede)(Won(a Nobel_Prize))->(a
Scandinavian)(Won(a Nobel_Prize)).
```

Formulating the examples

- A *NO* example

(11) A Swede did not win the Nobel Prize.
Every Swede is Scandinavian.
Did a Scandinavian win the Nobel prize? [No]

```
Theorem SWE:not((a Swede)(Won(a Nobel_Prize)))->not
(a Scandinavian)(Won(a Nobel_Prize)).
```

Formulating the examples

- An *UNK* example

(12) A Scandinavian won the Nobel Prize.
Every Swede is Scandinavian.
Did a Swede win the Nobel prize? [UNK, 3.65]

```
Theorem SWE:(a Scandinavian)(Won(a
Nobel_Prize))->(a Scandinavian)(Won(a Nobel_Prize)).

Theorem SWE:(a Scandinavian)(Won(a Nobel_Prize))->not((a
Scandinavian)(Won(a Nobel_Prize))).
```

Evaluating against the FraCas test suite - Quantifier monotonicity

- This section involves inferences due to quantifier monotonicity
  - Upwards monotonicity on the first argument

(13) Some Irish delegates finished the survey on time
Did any delegates finish the survey on time? [YES]

★ Standard semantics for indefinites *some* and *any* (no presuppositions encoded)

```
Definition some:= fun A:CN, fun P:A->Prop=> exists x:A, P(x).
```

Modification

- The examples we are dealing involve instance of adjectival modification
  - *Irishdelegate* in this case says that something is a *delegate* and furthermore *Irish*
  - We follow the Σ type treatment of adjectives. The first projection, $\pi 1$ is a coercion
  - We formulate it in Coq via means of dependent records

```
Record Irishdelegate:CN:=mkIrishdelegate{c:> Man;_:Irish c}
```

    - ⋆ With $Delegate : CN$ and $Irish : Object \rightarrow Prop$

Modification

- With these assumptions, nothing more is needed
  - The inference can be proven given the coercion of $\pi 1$
  - We formulate the theorem:

    ```
    Theorem IRISH: (some Irishdelegate(On_time(finish(the
    survey)))->(some Delegate)(On_time (finish(the survey))).

    compute.intro. elim H.intro.intro.exists x.auto.
    ```

  - Easy to prove. Subtyping does the work. Eliminating H and using intro we get an $x : Irishdelegate$ that $On\_time(finish(thesurvey))(x))$ holds. Then, given subtyping, $Irishdelegate < Delegate$ via the first projection $\pi 1$, we also have that $On\_time(finish(thesurvey))(x))$ with $x : Delegate$

Subtyping again

- Other similar examples involve more direct cases of subtyping

(14) A Swede won a Nobel prize
Every Swede is a Scandinavian
Did a Scandinavian win a Nobel Prize? [YES, 3.49]

- The above is multi-premised, i.e. more than one premise
  - We first define *Swede* and *Scandinavian* as being of type *CN*
    - ⋆ This is a case of nominalized adjectives. At least in this guise they function as *CNs*. One can give a Unit type capturing both guises (more on Unit types later)
  - Note that both arguments of the verb are quantifiers
  - In order to accommodate this, we have two options
    - ⋆ The first option is to define *won* as a regular transitive (leaving tense aside for the moment since it does not play a role in proving the inference). Then, in order to perform functional application, given the higher types for quantifiers, one must directly translate to something like the following: $\exists x : Man, \exists y : Object, win(x)(y)$ (scope issues are not going to be discussed here)

Subtyping again

- Alternatively, one can follow the strategy employed by Montague and type shift the verb, thus lifting to type $((Object \to Prop) \to Prop) \to (Human \to Prop)$
  - We exemplify with both alternatives
  - The most important part in proving the inference is the declaration of a subtyping relation between *Swede* and *Scandinavian*, i.e. $Swede < Scandinavian$

```
Parameter Swede Scandinavian:CN
Won: Object->Human->Prop.
Won: ((Object->Prop)->Prop)->(Human->Prop)
Axiom ss: Swede->Scandinavian.
Coercion ss: Swede >-> Scandinavian.
Theorem SWEDE1: (a Swede)(won (a Nobel_Prize))->(a
Scandinavian (won(a Nobel_Prize)).
Theorem SWEDE2: exists x:Swede, exists y:Nobel_Prize, won(x)(y
```

The Swede example

- Formulation with the verb type-lifted

```
SWEDE22<Theorem SWEDE22: (a Swede)(Won2(a Nobel_Prize))->
(a Scandinavian)(won(a Nobel_Prize)).
```

- We first use *cbv* to unfold the definitions. Then *intros*:

```
SWEDE22 < intros.
1 subgoal
H : exists x : Swede,
Won2 (fun P : Object -> Prop => exists x0: Nobel_Prize,
P x0) x
============================
exists x : Scandinavian,Won2 (fun P : Object -> Prop =>
exists x0 : Nobel_Prize, P x0) x
```

- Elimination (*elim*) can now be used followed by *eauto*. This suffices to prove the goal.

The Swede example

- Formulation with the verb regularly typed

```
Theorem SWEDE2: exists x:Swede, exists y:Nobel_Prize,
Won(y)(x)->exists x:Scandinavian, exists y:Nobel_Prize,
won(y)(x).
```

- There are no definitions to unfold and *intro* cannot apply.
- The natural solution is to be use *eauto*. However, this will give us the following error:

```
SWED < eauto.
No more subgoals but non-instantiated existential
variables:
Existential 1 = ?535 : [ |- Swede]
Existential 2 = ?536 : [ |- Nobel_Prize]
```

- This basically says that non-instantiated variables generated by *eapply* have been lost prior to instantiation

The Swede example

- The solution is to instantiate these variables
  - In this sense, we can introduce a number of variables (parameters) of type *Human* and a number of variables (parameters) of type *Object*
  - We can use one of these variables to instantiate the existentials
  - Starting with the proof, we basically instantiate both existentials
    - ★ We then apply *eauto*, and the proof is completed (with $d : Swede$ and $n : Scandinavian$.

```
SWED < exists d.
1 subgoal
============================
exists y : Nobel_Prize,
Won1 y d -> exists x : Scandinavian, exists y0 : Nobel_Prize,
Won1 y0 x
SWED < exists n.
1 subgoal
============================
 Won1 n d -> exists x : Scandinavian, exists y : Nobel_Prize,
Won1 y x
SWED < eauto.
Proof completed.
```

A NO example

- Monotonicity on the first argument

(15) No delegate finished the report on time
Did any Scandinavian delegate finish the report on time?
[NO, FraCas 3.70]

- We try to prove the negation of the hypothesis

```
Theorem SCAN: (no delegate)(On_time Human(finish(the
report)))->not((some Scandinaviandelegate)(On_time Human
(finish(the report)))).
```

- We apply *cbv* to unfold the definitions followed by *intros*
- Then, the tactic jauto can be used to complete the proof
  - *jauto* is similar to *eauto* but can further open up conjunctions and existentials (what we need here)

An UNK example

- Again from the monotonicity on the first argument part of the suite

(16) Some delegates finished the survey on time
Did any Irish delegates finish the survey on time? [UNK, FraCas 3.71]

- Indeed the above cannot be proven given that the subtyping relation is from *Irishdelegate* $<$ *delegate* and not the other way around
  - Basically, we end up with something like the following, and the proof cannot further continue

```
IRISH < AUTO.
H0 : exists x : delegate, On_time (finish (the survey)) x
x : delegate
H1 : On_time (finish (the survey)) x
============================
exists x0 : Irishdelegate,
On_time (finish (the survey)) (let (c0, _) := x0 in c0)
```

  - Trying to substitute x for x_0 fails since the terms are of different types!

Monotonicity on the second argument

- In this section we find examples like the following:

  (17) Some delegates finished the survey on time
  Did some delegates finish the survey? [UNK, FraCas 3.71]

- The inference in these cases comes from the veridicality of VP-adverbials like *ontime*
  - In order to capture this, we will have to see how VP veridical adverbials can be defined.
    - In order to do this we first introduce the auxiliary object ADV_{ver}, for veridical VP-adverbials

      (18) $ADV_{ver} : \Pi A : \textsc{cn}.\Pi v : A \rightarrow Prop.\ \Sigma p : A \rightarrow Prop.\forall x : A.p(x) \supset v(x)$

Monotonicity on the second argument

- Continued

(19) $ADV_{ver} : \Pi A\colon \textsc{cn}.\Pi v\colon A \rightarrow Prop.\ \Sigma p\colon A \rightarrow Prop.\forall x : A.p(x) \supset v(x)$

- Note that this is minimally different from $\forall A : CN, (A \rightarrow Prop) \rightarrow (A \rightarrow Prop)$, the only addition is the second part of the Σ specifying that in case $p(x)$ holds (the clause with the adverbial), then $V(x)$ also holds (the same sentence without the adverbial)
- Now, we define on time to be the first projection of this auxiliary object

(20) $on\ time = \lambda A\colon \textsc{cn}.\lambda v\colon A \rightarrow Prop.\ \pi_1(ADV(A, v))$

Monotonicity on the second argument

- We formulate these assumptions in Coq

```
Parameter ADV: forall (A : CN) (v : A -> Prop),sigT
(fun p : A -> Prop => forall x : A, p x -> v x).
Definition on_time:= fun A:CN, fun v:A->Prop=> projT1
(ADV(v)).
```

- Let us see whether this definition suffices to prove the inference in (19).

```
IRISH2 < Theorem IRISH2: (some delegate)(on_time
(finish(the survey)))->(some delegate)((finish
(the survey))).
```

Monotonicity on the second argument

- Continued
- We unfold the definitions and use destruct for *ADV* (basically it unfolds the definition for *ADV*)

```
IRISH2 < cbv. intro. destruct ADV in H.
1 subgoal
x : Human -> Prop
f0 : forall x0 : Human, x x0 -> finish (the survey) x0
H : exists x0 : delegate, x x0
============================
exists x0 : delegate, finish (the survey) x0
```

- We apply *induction* or *elim* to *H*
  - The difference between the two is that *induction* will add the inductive hypotheses into the context while *elim* will not
  - Applying *eauto* after this, will complete the proof

A note on veridical adverbs/adverbials

- The way proposed to capture veridicality can be generalized to all VP adverbs/adverbials.
  - For example if one is interested in getting the veridicality inferences right, ignoring other issues pertaining to the lexical semantics of each adverbial, then the auxiliary object can be used in all these cases
  - Thus, adverbs like slowly, fast etc. can given a similar definition to *on_time*

(21) $adv_{ver} = \lambda A \colon \textsc{cn}.\lambda v \colon A \rightarrow Prop.\ \pi_1(ADV_{ver}(A, v))$

  - A similar strategy can be used for veridical sentence adverbs. We first define an auxiliary object:

(22) $ADV_{Sver} \colon \Pi v \colon Prop.\ \Sigma p \colon Prop.p \supset V$

  - Then veridical sentence adverbs/adverbials like *fortunately, ironically* can be defined as:

(23) $adv_{Sver} = \lambda v \colon Prop.\ \pi_1(ADV_{Sver}(v))$

A note on veridical adverbs/adverbials

- We can check this in Coq

```
Coq < Theorem FORT: fortunately (walk John)-> walk John
1 subgoal
============================
fortunately (walk John) -> walk John
```

- We unfold the definitions and apply destruct to *ADVS*.

```
FORT < cbv. destruct ADVS.
x : Prop
w : x -> walk John
============================
x -> walk John
```

- Using *assumption* will complete the proof

Cases with more premises

- Example cases involving more than one premise

(24) Each European has the right to live in Europe
Every European is a person
Every person who has the right to live in Europe can travel freely within Europe
Can each European travel freely within Europe? [Yes, FraCas 3.20]

- For reasons of brevity some elements will be treated non-compositionally
  - But: only those that do not play any role in inference
  - Thus, to leave in Europe will be assumed as a single lexical item, since its treatment does not play any role in the specific inference
    - ★ Interesting case: Does *each european hat the right to live in Europe* imply *that each European has the right to live*?

Cases with more premises

- We assume *to_ live* to be a regular predicate. Then, we further assume that in_ Europe, freely and within_ Europe to be predicate modifiers
  - It is not difficult to give entries for prepositions *in* and *within* separately, but we will keep it simple in this case

```
Parameter in_Europe: forall A:CN, (A->Prop)->(A->Prop).
Parameter can: forall A:CN, (A->Prop)->(A->Prop).
Parameter travel: Object->Prop.
Parameter freely: forall A:CN, (A->Prop)->(A->Prop).
Parameter within_Europe: forall A:CN, (A->Prop)->(A->Prop).
```

Cases with more premises

- Let us formulate the example:
  - The first premise is straightforward
  - The second premise is encoded as a coercion and thus does not have to be present in the proof explicitly
  - The third premise is an implication relation (if a person... then)
  - Careful with the parentheses: the above two premises must imply the conclusion

    ```
    Theorem EUROPEAN: ((each European)(have
    (the righttoliveinEurope))/\forall x:person, ((have
    (the righttoliveinEurope)x)->Can (within_Europe(freely
    (travel)))x))->(each European)(Can (within_Europe(freely
    (travel)))).
    ```

  - Once formulated correctly, it is to prove
  - Using *cbv* to unfold the definitions, we can use *intuition* and complete the proof

One further example - at least two

- We define *at least two* as follows:

```
Definition at_least_two:= fun A:CN, fun P:A->Prop=>exists
exists y: A, P(x)/\(P(y))/\ not(x=y).
```

- With this one can deal with inferences like the following:

(25) At least two female commissioners spent time at home
At least two commissioners spent time at home [Yes, FraCas 3.63]

Adjectives section

- The adjectival class is notoriously non-homogeneous and a rather problematic class
  - Behaviour in terms of inference depends on the specific adjective
    - ★ The FraCas test suite uses a somehow different terminology than that usually found in the literature
    - ★ Affirmative/non-affirmative distinction: This is basically the subsective, non-subsective distinction in mainstream terminology.

    a. Affirmative: $Adj(N)(x) \Rightarrow N(x)$

    b. Non-affirmative: $Adj(N)(x) \Rightarrow \neg N(x)$ or undefined

Affirmative adjectives

- The Σ type account for adjectives suffices

(26) John has a genuine diamond
Does John have a diamond? [Yes, FraCas 3.197]

- Let us formulate the theorem

```
Theorem GENUINE: (a genuine_diamond)(has John)->
(a diamond)(has John).
```

- We unfold the definitions, use *intros*, *elim H* and *eauto*. The proof is completed

```
GENUINE < cbv. intros. elim H. eauto.
Proof completed.
```

- In a more economical way, *cbv* and *jauto* will also suffice to complete the proof

Opposites

- In this category, we find opposite adjectives like *small/large* in the FraCas test suite
  - What we want to get are the following inferences

    (27) $\text{Small(N)} \Rightarrow \neg\ \text{Large(N)}.$
    $\text{Large(N)} \Rightarrow \neg\ \text{Small(N)}$
    $\neg\ \text{Small(N)} \nRightarrow \text{Large(N)}.$
    $\neg\ \text{Large(N)} \nRightarrow \text{Small(N)}$

  - These are a little bit tricky to get
    - ★ The problem is that there are other sizes than a binary opposition *small-large*, e.g. normalsized items
    - ★ We can use this intuition to find a way out of the problem
    - ★ First define the element that its negation is implied by the other, i.e. large in our case
    - ★ We just give a regular predicate type for large
    - ★ Now, small is going to be defined as not being large AND not being normalsized (in fact additional sizes can be introduced, depends on the sizing granularity one assumes)

Opposites

- The definition for *small*

```
Definition small:= fun A:CN, fun a:A=> not (large (a)
/\ not (normalsized (a)).
```

- Checking against the examples

(28) Mickey is a small animal
Is Mickey a large animal? [No, FraCas 3.204)

- This is easily proven. We want to prove its negation

```
Theorem MICKEY: (Small Animal Mickey) ->not( Large Animal
Mickey).
```

- Unfolding the definitions, *intros, elim and eauto* or just *cbv* and *jauto* will complete the proof
  - Note how powerful *jauto* is. We are pretty much able to complete the proof in two steps (almost automation (we will exploit *jauto* when developing automated tactics))

Opposites

- The next example from FraCas shows an inference that we should not get, i.e. $\neg$ large $\Rightarrow$ small

(29) Fido is not a large animal
Is Fido a small animal? [UNK, FraCas 3.207)

- We formulate the theorem

```
Theorem FIDO: not(Large Animal Fido) ->Small Animal Fido.
```

- We cannot complete the proof
- The same goes for the same theorem with the implicatum negated

Comparison classes

- Adjectives that assume a comparison class like for example *small/big* (small for an *N*, big for an *N*) and adjectives that do not like *four-legged*
  - Let us see cases that do not assume a comparison class like *four-legged*
    - ⋆ We assume a simple predicate type $Animal \rightarrow Prop$
    - ⋆ Let us see a FraCas example

      (30) Dumbo is a four-legged animal
      Is Dumbo four-legged? [Yes, FraCas 3.203)

    - ⋆ We formulate the theorem (avoiding a discussion on how the copula should be treated here if at all)

      ```
      Theorem dfdss:exists x:Animal, four_legged(x)/\Dumbo=x->
      four_legged(Dumbo).
      ```

    - ⋆ We substitute *Dumbo* for *x* and use *jauto*. This suffices to complete the proof

Comparison classes

- Adjectives like big/small assume a comparison class
  - The idea is that something like *big elephant*, means big for an elephant but not big in general
    - ★ This is basically the subsective class of adjectives where the adjective noun combination implies the noun only (e.g. skilful surgeon(x) $\Rightarrow$ surgeon(x))
    - ★ Chatzikyriakidis and Luo (2013) deal with these types of adjectives by introducing a polymorphic type extending over the CNuniverse

      (31) $\Pi A : CN.A \rightarrow Prop$

    - ★ The idea is that typing is dependent on the choice of A. If A is of type *Animal* then the type will be $Animal \rightarrow Prop$, if A is of type *Human*, the typing would be $Human \rightarrow Prop$ and so on

Comparison classes

- This polymorphic type along with the lexical semantics given for *small* will predict the correct inferences
  - Consider the following example

    (32) All mice are small animals
    Mickey is a large mouse
    Is Mickey a large animal? [No, FraCas 3.210)

  - We formulate the theorem

    ```
    Theorem MICKEY2: (all Mouse (Small Animal)/\ Large Mouse
    Mickey)->not( Large Animal Mickey)
    ```

  - We unfold the definitions and apply *intro*, followed by two applications of *induction* or *destruction* of *H*
  - In the second use, we have to introduce the value for *x* ourselves, *Mickey* in our case. Otherwise we can use *edestruct* or *einduction*

Comparison classes

- Continued

```
H : forall x : Mouse,(Large Animal x -> False) /\
(Normalsized Animal x -> False)
H0 : Large Mouse Mickey
H1 : Large Animal Mickey -> False
H2 : Normalsized Animal Mickey -> False
============================
Large Animal Mickey -> False
```

- Applying *assumption* completes the proof
- The other examples in the section can be proven in a similar way

Comparatives

- Two ways to deal with comparatives: One without measures, one with measures
  - Both proposals were put forth in Chatzikyriakidis and Luo (2014)
  - The examples in the test suite do not need the explicit introduction of measures so we will concentrate on the approach without measures
    - ★ The same of idea of using an auxiliary object first is used. Thus, in the case of *SMALLER_THAN* one can define the following:

(33) $SHORTER\_THAN : \Sigma p : Human \rightarrow Human \rightarrow Prop.\ \forall h_1, h_2, h_3 : Human.\ p(h_1, h_2) \wedge p(h_2, h_3) \supset p(h_1, h_3) \wedge \forall h_1, h_2 : Human.p(h_1, h_2) \supset short(h_2) \supset short(h_1)$

(34) $shorter\ than = \pi_1(SHORTER\_THAN)$

    - ★ This basically captures the transitive properties of comparatives as well as the fact that an x being *A_er* than something does not mean that this x is also A (being shorter than something does not guarantee shortness)
    - ★ It does however just in case the y that x is in a *A_er* relation with, is A

Comparatives

- Let us see an example

(35) The PC-6082 is faster than the ITEL-XZ
The ITEL-xz is fast
Is the PC-6082 fast? [Yes, FraCas 3.220)

- We define *faster_than* in the sense described

```
Parameter FASTER_THAN : forall A : CN, {p : A -> A ->
Prop & forall h1 h2 h3 : A, (p h1 h2 /\ p h2 h3 ->
p h1 h3) /\ (forall h4 h5 : A, p h4 h5 -> Fast1 A h4 ->
Fast1 A h5)}.
Definition faster_than:= fun A:CN=>projT1 (FASTER_THAN A).
```

- With this, examples like (35) can be proven
- More on comparatives and inference in Chatzikyriakidis and Luo (2014)

Epistemic, Intensional and Reportive Attitudes

- Section on the FraCas dealing with verbs that presuppose the truth of their propositional complement (e.g. *know*) and verbs that do not (e.g. *believe*)
  - For verbs like *believe* just a typing with no additional semantics will do

    (36) $believe : Prop \rightarrow Human \rightarrow Prop$

    - For a treatment of belief intensionality in MTTs, see Ranta (1994), Chatzikyriakidis and Luo (2013), Chatzikyriakidis (2014)
  - For verbs that presuppose the truth of their complement, we can use a strategy similar to the one used for veridical adverbs
  - We define an auxiliary object first and then the lexical entry

    (37) $KNOW = \Sigma p : Human \rightarrow Prop \rightarrow Prop.\ \forall h : Human \forall P : Prop.\ p(h, P) \supset P$
    $know = \pi_1(KNOW)$

Epistemic, Intensional and Reportive Attitudes

- Examples like the following can be treated:

(38) John knows that Itel won the contract
Did Itel win the contract? [Yes, FraCas 3.334]

(39) Smith believed that Itel had won the contract Did Itel win the contract? [UNK, FraCas 3.335]

```
Theorem KNOW:know John((Won1 (the Contract) ITEL))->
(Won1 (the Contract) ITEL) .
```

- We unfold the definitions, *destruct* the auxiliary object and then use *eapply*

Plurals

- The section on plurals in the FraCas contains various subsections
- Conjoined plurals. Examples like the following

(40) Smith, Jones and Anderson signed the contract
Did Jones sign the contract? [Yes, FraCas 3.81]

- We can define conjunction using the same technique of using an auxiliary item
  - The following proposal was put forth in Chatzikyriakidis and Luo (2014) for the three place conjunction (see the paper on how to propose a generalized n-ary conjunction)

(41) $AND_3 : \Pi A : LType.\ \Pi x, y, z : A.\ \Sigma a : A.\ \forall p : A \to Prop.\ p(a) \supset p(x) \wedge p(y) \wedge p(z).$
$and_3 = \lambda A : LType.\lambda x, y, z : A.\ \pi_1(AND_3(A, x, y, z))$

Plurals

- We formulate these assumptions in Coq
  - We use *Type* instead of *Ltype* given that universe construction is not an option in Coq
  - We these assumptions we can deal with examples like (42)
  - We formulate the theorem

```
Theorem CONJ:(Signed(the Contract)(and3 Smith Jones
Anderson)-> (Signed(the Contract)Smith)).
```

  - We unfold the definitions and destruct *AND*3

```
x : Man
a : forall p:Man->Prop,p x->p Smith/\ p Jones /\ p Anderson
============================
Signed (the Contract)x->Signed(the Contract) Smith
```

  - We use *apply a* and then *eauto* to complete the proof
  - Similar entries can be assumed for disjunction

Plurals

- Dependent plurals

(42) All APCOM managers have company cars
John is an APCOM manager
Does John have a company car? [Yes, FraCas 3.2.4)

- Again, we introduce some form non-compositionality for APCOM managers and company cars, since compositionaliity of these expressions does not play any role in the proof
    - The semantics given for *all* guarantee the completion of the proof

Temporal reference

- We introduce a simple model of tense
  - We introduce first the parameter $Time : Type$
    - We have a precedence relation $\leq$ and a specific object $now : Time$, standing for 'the current time' or the 'default time'
    - We can define *Time* as an inductive with one of its constructors being the following:

      (43) $DATE : date \rightarrow Time$

    - Where date consists of the triples (y, m, d) ranging over integers for years, months and days respectively

Temporal reference

- Now, a present verb will say that the proposition expressed holds at the default time while a past tense verb at a time prior to the default time.
  - A number of inferences can be captured in this way. Let us see one:

    (44) ITEL has a factory in Birmingham
    Does ITEL currently have a factory in Birmingham? [Yes, FraCas 3.251]

  - We define currently to take an argument $P : \mathit{Time} \rightarrow \mathit{Prop}$ and specify that $P(\mathit{default}_t)$, P holds in the default time
  - The present tense of the verb will also specify that P holds at the default time.

```
Definition currently:=fun P : Time -> Prop=> P default_t
Definition Has:=fun (x : Object)(y : Human) (t : Time)=>
Have x y t /\ t = default_t.
```

Temporal reference

- We formulate the theorem (we ignore the adverbial for the moment)

  ```
  Theorem sCURRENTLY: (Has (a_factory))ITEL t-> currently ((
  ```

- We unfold the definitions and use *intros*, then we split the goal and destruct the hypothesis

  ```
  H : Have a_factory ITEL t
  H0 : t = DATE default_y default_m default_d
  ============================
  Have a_factory ITEL (DATE default_y default_m default_d)
  subgoal 2 is:
  DATE default_y default_m default_d = DATE default_y
  default_m default_d
  ```

- See Chatzikyriakidis and Luo (2014) for more examples
- We stop here as regards the phenomena to look at
  - See Chatzikyriakidis and Luo (2014) for more semantic phenomena e.g. bare plurals, elementary aspect and collective predication among others

Automation

- We have seen that Coq is a powerful tool to reason about NL semantics
  - We have seen that using more composite tactics can shorten the proofs, e.g. using the *jauto* instead of the *eauto* tactic in cases of existentials.
    - ★ The question is whether we can fully automate our proofs
    - ★ It seems that we can, at least for the examples we are dealing
    - ★ We have seen that a number of examples can be proven using *jauto* or *intuition* after their definitions are unfolded. We have also seen in the end that *congruence* is also a very useful tactic to deal with equalities
    - ★ We can define a new composite tactic called *AUTO* that will basically formed out of the tactics just mentioned

      ```
      Ltac AUTO:= cbv delta;intuition;try repeat congruence;
      jauto;intuition.
      ```

Automation

- Using the AUTO tactic
  - It turns out that *AUTO* is quite a powerful tactic
    - ★ It can actually automate many of the examples we were dealing with (and most importantly a lot more similar examples)

```
Theorem EX1:some Man (walk)->(some Human) walk
Theorem EX2: (walk) John-> some Man (walk).
Theorem IRISH: (some Irishdelegate)(On_time(finish(the
survey)))->(some delegate)(On_time (finish(the survey))).
Theorem SWEDE22: (a Swede) (Won2(a Nobel_Prize))->(a
Scandinavian)(Won2(a Nobel_Prize)).#
Theorem SCAN: (no delegate)(On_time Human(finish(the report))
->not((a Scandinaviandelegate)(On_time Human (finish(the
report)))).
Theorem EUROPE:((each European)(have(the righttoliveinEurope)
/\forall x:person, ((have(the righttoliveinEurope)x)->Can
(within_Europe(freely (travel)))x))->(each European)(Can
(within_Europe(freely(travel)))).
Theorem GENUINE: (a genuine_diamond)(has John)->(a diamond)
(has John).
Theorem MICKEY: (Small Animal Mickey) ->not( Large Animal
Mickey).
```

Automation

- *AUTO* will fail in cases where destruct is needed, e.g. in the cases for factive complements, comparatives, conjunction etc.
  - We can remedy this by introducing a tactic which trie destruct before calling *AUTO* (the tactic is a little bit more complex but the details are not needed here)

    ```
    Ltac AUTOa  x i:= cbv;try destruct x;try intro;
    try ecase i;  AUTO;  try eapply i; try omega; AUTO;
    intuition; try repeat congruence; jauto;intuition.
    ```

  - Let us say we want to prove something which needs destruct

    ```
    Theorem KNOW:know John((Won1 (the Contract) ITEL))->(Won1 (
    ITEL) .
    ```

  - This can be automated with the new tactic now
  - Now, we can combine the two tactics into one generalized *GAUTO* tactic that tries to solve the goal via using one of the two automated tactics discussed

    ```
    Ltac GAUTO:= solve[AUTO|AUTOa].
    ```

Automation

- *GAUTO* automates most of the proofs
  - There are some further cases like collective predication that need additional steps
    - Extra AUTO tactics are defined in Chatzikyriakidis and Luo (2014) for these cases and are then added to *GAUTO*.
    - All the examples discussed in the paper are given automated proofs
    - How far can one go with automation?
    - Is automation possible when NLIs are longer?

How can we use Coq for CLASP?

- What we have is a powerful reasoner implementing a rich type theory
  - There is a rather straightforward way to encode NL semantics in Coq
  - One of the options is to output Coq TTR record represenations
  - This will then can be reasoned about in Coq
- Let us see a simple example

A simple TTR example

- TTR record types using Coq's record type mechanism
  - A simple non-compositional example, taken from Robin's draft

```
Definition Ind:=Set.
Parameter man: Ind->Prop.
Parameter donkey: Ind->Prop.
Parameter own: Ind->Ind->Prop.
Record amanownsadonkey : Type := mkamanownsadonkey{ x : In
c1 : man x;  y : Ind;  c2 : donkey y; c3 : own x y }.
```

- In terms of inference, one can infer any of the fields in case an object $e : amanownsadonkey$ exists
- One can for example infer that there is an x of type *Ind* that is a *man* and similarly that there are x and y of type *Ind* that stand in an *own* relation to each other
  - If we have records as output or equivalent translations to some logic, then we can very well use Coq to reason about the semantics

Other semantic frameworks in Coq

- Simple neo-Davidsonian Brutus Semantics