Temporal Logic in Coq

Nuno Paiva

Nº 38994

Diploma thesis

Supervised by Amílcar Sernadas and Carlos Caleiro

Instituto Superior Técnico

Licenciatura em Matemática Aplicada e Computação

July 1998

Abstract

The aim of this work is to implement temporal logic in the Coq proof assistant system. This implementation uses the logical language of Coq as meta-language for temporal logic representation. The work starts with a crash introduction to Coq devoted to introduce the Coq system. The implementation of linear temporal logic and two branching temporal logics is discussed. In both linear and branching temporal logic soundness verification of proposed axiomatizations is made. Some application examples are shown.

Acknowledgments

To Prof. Amílcar Sernadas, whitout whom this work would not have been possible, for his ideas and guidance.

To Carlos for his guidance, constant help, presence and friendship.

To Jaime and Paulo for their suggestions.

To Sara and Alexandra for their good humor, support and fellowship.

To all section 84.

This work was partially supported by the PRAXIS XXI Program and FCT, as well as by PRAXIS XXI Projects 2/2.1/MAT/262/94 SitCalc, PCEX/P/MAT/46/96 ACL plus 2/2.1/TIT/1658/95 LogComp, and ESPRIT IV Working Groups 22704 ASPIRE and 23531 FIREworks.

Contents

Chapter 1

Introduction

The aim of this work is to implement temporal logic in the Coq proof assistant system. The implementation uses the logical language of Coq as meta-language for temporal logic representation.

Temporal logic is designed to reason about how truth values of assertions vary with time [Eme90]. It is useful, among other applications, to specify and verify correctness of computer programs, especially appropriate for reasoning about nonterminating or continuously operating concurrent programs, such as operating systems and network communication protocols. In spite of this, this work only focuses on the formal aspects of temporal logics and its implementation in the Coq system.

The following gives an overview of the chapters that the work is divided in. The work starts with a crash introduction to Coq devoted to introduce the Coq system and its specification language *Gallina*. Propositional logic is used as a working example. Co-inductive types used for the interpretation structures in further chapters, as well as the *modus operandi* of the work done with Coq for the temporal logics considered, are also introduced.

In chapter 3 the implementation of linear temporal logic in Coq is discussed. A special type of streams is used for the interpretation structures. The considered implementation is used for soundness verification. Some examples of application are given.

Analogously, in chapter 4, the implementation of two branching temporal logics in Coq – UB and CTL – and their use for soundness verification is discussed. Co-inductive trees of infinite depth and at least one branch are used as interpretation structures. Finally, in chapter 5, some conclusions of this work are drawn.

This work was developed in Coq V6.10. The implementation of the semantics, interpretation structures and complete proofs of soundness verification of the referred logics are given in Appendix.

Chapter 2

A Crash introduction to Coq

In this chapter a short presentation of the specification language *Gallina* and the proof system of Coq is given. Propositional Logic (PL) as described in [Ser93] is the chosen example. The representations of both its syntax and semantics are used to help describe both the specification language (used as a meta-language for the representation of PL) and the proof system of Coq. Moreover, the style of presentation is also similar to those adopted in the subsequent chapters, the core of this work.

2.1 Syntax of PL

Given a set of propositional symbols P, the set *Pform* of formulae of propositional logic is inductively defined as follows:

- Each element of P is a formula;
- If a is a formula then $(\neg a)$ is a formula;
- If a and b are formulae then $(a \Rightarrow b)$ is a formula.

Usual abbreviations of propositional connectives are $(a \wedge b) \equiv_{abv} (\neg(a \Rightarrow (\neg b)))$ and $(a \vee b) \equiv_{abv} ((\neg a) \Rightarrow b)$.

The corresponding definition in *Gallina* is based on a variable representing the set of propositional symbols `P` and an inductive type `Pform` (the type of propositional formulae).

```
Section PL.

  Variable P : Set.

  Inductive Type Pform :=
      id   :  P → Pform
    | no   :  Pform → Pform
    | imp  :  Pform → Pform → Pform
    | andp :  Pform → Pform → Pform
    | orp  :  Pform → Pform → Pform.
```

The command `Section PL` is used to open the section named PL. This mechanism allows to organize a proof in structured sections. A section opened with `Section` *ident* command must be closed with a corresponding `End` *ident* command. When a section is closed all global objects defined inside are *closed* with as many abstractions (in the sense of λ-calculus) as there were local declarations in the section explicitly occurring in the term.

The sort `Set` is linked to the name `P` by the command `Variable` in the context of the current section. This means that `P` will be unknown when the section is closed and the variable is said to be *discharged*.

The `Inductive` command is used to define inductive types and inductive families such as inductively defined relations. The name `Pform` is the name of the inductively defined object and is of sort `Type`. The names `id`, `no`, ..., `orp` are the names of its constructors and `P` $\rightarrow$ `Pform`, `Pform` $\rightarrow$ `Pform`, ..., `Pform` $\rightarrow$ `Pform` $\rightarrow$ `Pform` are their corresponding types.

The term `id` introduces propositional symbols as formulae, `no` means that the negation of a formula is a formula, `imp` stands for formulae implication, `andp` stands for formulae conjunction and finally `orp` stands for formulae disjunction. It is not possible to use `not` instead of `no` because `not` is a reserved word of `Coq`. The same happens with `and` and `or`.

The constructors must satisfy a well-foundedness condition called the *positivity condition* which is better explained in Section 6.5.3 of [PM96]. Roughly, this means that the basis of induction must be well defined. In the present it corresponds to the constructor `id`.

The `Coq` system provides three destructors for `Pform` named `Pform_ind`, `Pform_rec` and `Pform_rect`. These are elimination principles for, respectively, `Prop`, `Set` and `Type`. Refer to Section 2.6.1 of [PM96] for a detailed explanation of the `Inductive` command.

Note that there are five constructors for the type of formulae of propositional logic `Pform`, instead of the expected three: identity, negation and implication. This is due to the fact that implication, negation, disjunction and conjunction are all primitive connectives for intuitionistic logic, in which the `Coq` system is based. Thus it is not possible to make use of the usual abbreviations of classical propositional logic and therefore the five constructors are needed, in order to have full expressiveness.

2.2 Semantics of PL

The interpretation structures of PL are valuations. A valuation is a map $V : P \rightarrow \{0, 1\}$.

The satisfaction relation $\Vdash$ between valuations and formulae is inductively defined as follows:

- $V \Vdash p$ iff $V(p) = 1$, if $p \in P$;
- $V \Vdash (\neg a)$ iff not $V \Vdash a$, if $a \in \mathit{Pform}$;
- $V \Vdash (a \Rightarrow b)$ iff not $V \Vdash a$ or $V \Vdash b$, if $a, b \in \mathit{Pform}$.

The implementation of these two definitions help proceed with the explanation of *Gallina*.

```
Definition Valuation := P → Prop.

Fixpoint Sat[v:Valuation; f:Pform] :  Prop :=
  Case f of
    [a:P]       (v a)
    [a:Pform]   ¬(Sat v a)
    [a,b:Pform] (Sat v a) → (Sat v b)
    [a,b:Pform] (Sat v a) ∧ (Sat v b)
    [a,b:Pform] (Sat v a) ∨ (Sat v b)
  end.
```

The `Definition` command above binds the value `P → Prop` to the name `Valuation` in the environment. Definitions differ from declarations, such as `Variable P` in the former example, since they allow to assign a name to a term, whereas declarations just link a type to a name. In this way, the name of the defined object can be replaced at any time by its definition, and it is said to be *constant in the environment*.

`Fixpoint` is used to define `Sat` as an inductive object using a fixed point construction. `Sat` is defined as a recursive function with two arguments, such that `(Sat v f)` has type `Prop` if `v` has type `Valuation` and `f` has type `Pform`. Note that at least one of the arguments of a `Fixpoint` definition must be an inductively defined type, in this case `f`, and is called the *recursive variable* of `Sat`. This restriction is needed to ensure that the `Fixpoint` definition always terminates. Refer to Section 2.6.3 of [PM96].

The `Case` operator matches a value (here `f`) with the various constructors of its inductive type. The remaining arguments give the respective values to be returned, as functions of the parameters of the corresponding constructor. Thus we return `(v p)` when `f` equals `(id p)`, `¬(Sat v a)` when `f` equals `(no a)`, ..., `(Sat v a) ∨ (Sat v b)` when `f` equals `(orp a b)`. The system recognizes that in the recursive calls the second argument actually decreases because it is a *pattern variable* coming from `Case f of`. Refer to Section 6.5.4 of [PM96] for more details.

The overview continues with a few more examples, the definitions of valid formula, subset of propositional formulae and entailment relation.

A formula $f \in \mathit{Pform}$ is said to be valid iff it is satisfied by all valuations, i.e., $V \Vdash f$ for all valuations V.

A subset of propositional formulae is a map from *Pform* to $\{0, 1\}$.

Given a subset of propositional formulae Φ and a propositional formula f, then Φ entails f, which is written $\Phi \models f$, iff for all valuations V, $V \Vdash f$ whenever $V \Vdash a$ for each $a \in \Phi$.

The previous definitions are straightforward in Coq.

```
Definition Valid :  Pform → Prop :=
  [f:Pform] (v:Valuation) (Sat v f).

Definition Subset_Pform := Pform→Prop.

Definition Entails :  Subset_Pform → Pform → Prop :=
  [Phi:Subset_Pform] [f:Pform] (v:Valuation)
```

```
(((f':Pform)((Phi f')→(Sat v f')))→(Sat v f)).

End PL.
```

In the definition of `Valid` the term `Pform` $\rightarrow$ `Prop` is used for type checking and is definitionally equal to `[f:Pform] (v:Valuation) (Sat v f)`. The term in square brackets means that `Valid` has an argument `f` of type `Pform`. The first term in parentheses `(v:Valuation)` is an universal quantification over the type `Valuation`. The second one `(Sat v f)` is the universally quantified proposition. Furthermore `(Valid f)` can be replaced at any time by `(v:Valuation) (Sat v f)`.

The definition of `Subset_Pform` does not offer any difficulty after `Valuation` has been explained. In the definition of `Entails`, notice that there are two arguments of different types, `Phi` of type `Subset_Pform` and `f` of type `Pform`. Notice also that the proposition is an universal quantification with an implication. The antecedent is `((a:Pform)((Phi a)→(Sat v a)))` which is also an universal quantification with an implication. The consequent of the outer implication is just `(Sat v f)`.

Note that the section PL is closed with `End PL`.

2.3 Soundness verification of PL

The axiomatization of PL that is considered is usual, with the axioms:

$Ax1$. $(a \Rightarrow (b \Rightarrow a))$

$Ax2$. $((a \Rightarrow (b \Rightarrow c)) \Rightarrow ((a \Rightarrow b) \Rightarrow (a \Rightarrow c)))$

$Ax3$. $(((\neg b) \Rightarrow (\neg a)) \Rightarrow (a \Rightarrow b))$

and the inference rule:

MP: From a and $(a \Rightarrow b)$ it is derived b.

The proof system of Coq is now used to prove the soundness of the axiomatization over the defined semantics. For further details refer to Appendix A.

```
Require Export PL.

Section PL_Soundness.

  Variable P : Set.

  Theorem Ax1 :  (a,b :  (Pform P))
    (Valid P (imp P a (imp P b a))).
  Proof.
    Intros.
    Unfold Valid.
```

```
  Intro.
  Simpl.
  Auto.
Qed.
```

Note that the variable P stands for the set of propositional atomic formulae and is only used for parameter instantiation. The proof of Theorem Ax1 is illustrated as it appears on the screen during its realization. To start with, the goal of the proof is written.

```
Coq < Theorem Ax1 :  (a,b :  (Pform P))(Valid P (imp P a (imp P b a))).

1 subgoal

  P : Set
  ============================
   (a,b :  (Pform P))(Valid P (imp P a (imp P b a)))
```

The Intros command introduces all possible hypotheses and universal quantifications that are in the goal.

```
Ax1 < Intros.

1 subgoal

  P : Set
  a :  (Pform P)
  b :  (Pform P)
  ============================
   (Valid P (imp P a (imp P b a)))
```

The command Unfold Valid replaces Valid by its definition

```
Ax1 < Unfold Valid.

1 subgoal

  P : Set
  a :  (Pform P)
  b :  (Pform P)
  ============================
   (v:(Valuation P))(Sat P v (imp P a (imp P b a)))
```

Intro now introduces the quantification over v:(Valuation P).

```
Ax1 < Intro.

1 subgoal
```

```
P : Set
a :  (Pform P)
b :  (Pform P)
v :  (Valuation P)
============================
 (Sat P v (imp P a (imp P b a)))
```

The tactic `Simpl` simplifies the `Sat` relation according to its definition.

```
Ax1 < Simpl.

1 subgoal

  P : Set
  a :  (Pform P)
  b :  (Pform P)
  v :  (Valuation P)
  ============================
   (Sat P v a)→(Sat P v b)→(Sat P v a)
```

The goal is now a simple tautology of intuitionistic logic, thus, the tactic `Auto` is able to prove it.

```
Ax1 < Auto.

Subtree proved!
```

This concludes the proof of `Theorem Ax1`. The proof of $Ax2$ is similar to the previous one, and therefore, not presented here.

```
Theorem Ax2 :  (a,b,c:(Pform P))
  (Valid P
    (imp P
      (imp P a (imp P b c))
      (imp P (imp P a b) (imp P a c)))).
```

In order to prove the $Ax3$ it is necessary to consider the Excluded Middle (EM) axiom of classical propositional logic. Once more, note that `Coq` is based on intuitionistic logic where the EM axiom is not a tautology. It is necessary to introduce the EM axiom, which is done by declaring it as an `Axiom`. The proof is then similar to those of $Ax1$ and $Ax2$.

```
Axiom EM : (X:Prop) X ∨ ¬X.

Theorem Ax3 :  (a,b:(Pform P))
  (Valid P (imp P (imp P (no P b) (no P a)) (imp P a b))).
```

The next theorems do not offer any special difficulty. Note that a few hints are given to the proof system of Coq, so that they can be used with the `Auto` tactic. This ends the definition of propositional logic and corresponding soundness verification in Coq.

```
Theorem MP :
  (a,b:(Pform P))
    (v:(Valuation P))
      (Sat P v a) ∧ (Sat P v (imp P a b)) → (Sat P v b).
```

It is possible to prove the abbreviations $(a \vee b) \equiv_{abv} ((\neg a) \Rightarrow b)$ which is `orp_abbv_no_imp` and $(a \wedge b) \equiv_{abv} \neg(a \Rightarrow (\neg b))$ which is `andp_abbv_no_imp`, using the `EM` axiom.

```
Theorem orp_abbv_no_imp :  (a,b:  (Pform P))
  (Valid P (orp P a b)) ↔ (Valid P (imp P (no P a) b)).

Theorem andp_abbv_no_imp :  (a,b:(Pform P))
    (Valid P (andp P a b)) ↔ (Valid P (no P (imp P a (no P b)))).

End PL_Soundness.

Hint Unfold Valid.
Hint Unfold Entails.
Hint Ax1 Ax2 Ax3 MP.
```

2.4 An application example

An application example is now presented as in the following chapter of the dissertation. Although very simple it can be seen as a typical application of the implemented semantics. Plus, it can be used as a test of expressiveness.

It is intended to prove the following lemma of PL, usually known as "syllogism":

$$\{(a \Rightarrow b), (b \Rightarrow c)\} \models (a \Rightarrow c)$$

which is done with the following implementation in the Coq system:

```
Section Application_pl.

  Variable P : Set.

  Variables a, b, c :  (Pform P).

  Definition Gamma :  (Subset_Pform P) := [t:(Pform P)]
    Cases t of
       (imp a b) ⇒ True
     | (imp b c) ⇒ True
```

```
  |    _     ⇒ False
  end.

Lemma syllogism :  (Entails P Gamma (imp P a c)).
Proof.
  Unfold Entails.
  Intros.
  Simpl.
  Intros.
  Cut (Sat P v (imp P b c)).
  Simpl.
  Intro.
  Apply H1.
  Cut (Sat P v (imp P a b)).
  Simpl.
  Intro.
  Apply H2.
  Assumption.

  Apply H.
  Unfold Gamma.
  Auto.

  Apply H.
  Unfold Gamma.
  Auto.
Qed.

End Application_pl.
```

Again, the variable P stands for the set of propositional atomic formulae and is only used for parameter instantiation. Variables a, b and c stand for meta-formulae. Gamma is an instantiated subset of formulae which contains only (a⇒b) and (b⇒c). The proof of the lemma, although a bit long, is straightforward. Refer to Appendix A for more details.

2.5 Co-inductive Types

As was said before, the goal of this dissertation is to analyze temporal logic within Coq. In the previous sections the considered interpretation structures were just valuations. There was no temporal reasoning present nor any time structure. However, in temporal logic one must be able to make assertions over time, interpretation structures must allow to understand temporal assertions and time structures must be able to represent them.

Two basic infinite structures of time shall be considered. The first is linear, in which time has a sequential nature and at each moment there is exactly one possible future. The other is branching, in which time has a tree-like nature and at each moment time may split into alternative courses representing different possible futures.

These time structures are infinite, and to be able to represent them in Coq co-inductive types are needed. One of the main characteristics of co-inductive types is that there is no induction principle available for them, once the constructors may be applied an infinite number of times. Thus, elimination must be made in a more primitive way, by case analysis, i.e., by considering through which constructor a term could have been introduced.

This section introduces the co-inductive types used to support the time structures needed in the following chapters.

2.5.1 Streams

In the linear case time can be seen as an infinite line. The underlying structure of time can be described by the following properties:

- Time is discrete;
- Time has an initial moment (with no predecessors);
- Each moment has exactly one successor.

Therefore, a timeline is isomorphic to $(\mathbb{N}_o, <)$. A labeling function $\lambda : \mathbb{N}_o \rightarrow L$, where L is a set of labels, is also considered.

Coq provides a type of infinite sequences, called streams, which is a co-inductive type. Streams fulfill the properties of time enumerated above. However the streams of Coq are defined to be instantiated with the sets of Coq. Therefore they cannot be instantiated with types such as `Set → Prop`, that are used as subsets. Streams that can be instantiated with types (`TStream`) are introduced. `TStream` has two new operators and co-inductive equality was not implemented once syntactic equality was sufficient for the present purposes.

The following are just the equivalent for `TStream` to the basic definitions of streams provided by the library of Coq `Stream.v` and introduced in chapter 10 of [PM96]. Refer to [PM96] and [INR96] for further explanations.

```
Section Type_Streams.

Section Type_Streams.

  Variable L : Type.

  CoInductive Type TStream := TScons :  L→TStream→TStream.

  Definition TShead :=
    [s:TStream] Cases s of (TScons a _) ⇒ a end.

  Definition TStail :=
    [s:TStream] Cases s of (TScons _ s') ⇒ s' end.

  Fixpoint TSnth[n:nat] :  TStream→L:=
    [s:TStream] Cases n of
          O   ⇒ (TShead s)
        |(S m) ⇒ (TSnth m (TStail s))
```

```
end.
```

The CoInductive command is similar in every aspect to the Inductive command. Note that the constructor of type TStream is TScons which given an element of type L (a label) and an element of type TStream gives an element of type TStream. This constructor may be applied an infinite number of times to itself.

The terms TShead and TStail are destructors of TStream that use the only available elimination principle, case analysis. Finally TSnth is a function that given a natural number n and a TStream s, gives the n-th element of TStream s.

The two new operators are now presented together with some lemmas about them.

```
Fixpoint TSnth_tail[n:nat] :  TStream→TStream :=
  [s:TStream] Cases n of
        O   ⇒ s
      |(S m) ⇒ (TStail (TSnth_tail m s))
  end.

Lemma one_step_nth_tail :  (s:TStream)(n:nat)
  (TStail (TSnth_tail n s)) == (TSnth_tail (S n) s).

Lemma multi_step_nth_tail :  (s:TStream)(n:nat)
  (TSnth_tail (S n) s) == (TSnth_tail n (TStail s)).
```

The operator TSnth_tail is defined inductively on nat. Given a natural n and a TStream s, it consists of applying the TStail operator n times to TStream s. The lemma one_step_nth_tail simply expresses the step of TSnth_tail and multi_step_nth_tail states that applying n+1 times TStail to TStream s is the same as applying n times TStail to (TStail s).

The other new operator is TSnth_conc and consists of concatenating the first n elements of a TStream s with a TStream s1.

```
Fixpoint TSnth_conc[n:nat] :  TStream→TStream→TStream :=
  [s,s1:TStream] Cases n of
        O   ⇒ s1
      |(S m) ⇒ (TScons (TShead s)
                        (TSnth_conc m (TStail s) s1))
  end.

Lemma multi_step_nth_conc:  (n:nat)(s,s1:TStream)
  (TSnth_conc (S n) s s1)
    == (TSnth_conc n s (TScons (TSnth n s) s1)).
```

Lemma multi_step_nth_conc shows that concatenating the first n+1 elements of TStream s with TStream s1 has the same result as concatenating the first n elements of TStream s with the TStream whose head is the n+1-th element of TStream s and whose tail is TStream s1.

The following lemmas relate TSnth_tail and TSnth_conc.

```
Lemma nth_tail_with_nth_conc :  (n:nat)(s,s1:TStream)
      (TSnth_tail n (TSnth_conc n s s1)) == s1.

Lemma nth_conc_with_nth_tail :  (n:nat)(s:TStream)
      (TSnth_conc n s (TSnth_tail n s)) == s.
```

The lemma `nth_tail_with_nth_conc` states that `TStream s1` is the result of applying `n` times `TStail` to the `TStream` resulting of concatenating the first `n` elements of `TStream s` to `TStream s1`. Lemma `nth_conc_with_nth_tail` expresses the fact that `s` is the result of concatenating the first `n` elements of `TStream s` to `(TSnth_tail n s)`.

The following lemma states that eliminating and then re-introducing a `TStream` yields the same `TStream`. The other two definitions are just syntactic sugar for `TShead` and `TStail` respectively.

```
Lemma unfold_TStream :  (s:TStream)
      s == (Case s of TScons end).

Syntactic Definition TShd := (TShead ?).

Syntactic Definition TStl := (TStail ?).

End Type_Streams.
```

2.5.2 Trees

In the branching case time can be seen as an infinite tree. The underlying structure of time can be described by the following properties:

- Time is discrete;
- Time has an initial moment (with no predecessors);
- Each moment has at least one successor.

Therefore, a timetree T can be given the following characterization:

- $T \subseteq {I\!N_o}^*$ satisfying:
  - if $w.n \in T$ then $w \in T$;
  - if $w \in T$ then there exists $n \in I\!N_o$ such that $w.n \in T$.

The first condition is just prefix closure. The second one ensures that each timeline in the tree is isomorphic to $(I\!N_o, <)$. For implementation reasons a third condition is required in order to ensure minimality of the representation:

- if $w.n \in T$ and $m < n$ then $w.m \in T$.

Again, a labeling function $\lambda : T \to L$, where L is a set of labels, is used.

Also a *path* definition is useful. A path p of a tree T is such that $\emptyset \neq p \subseteq T$ and satisfies:

- If $w.n \in p$ then $w \in p$;
- If $w \in p$ then $\exists^1 n$ s.t. $w.n \in p$.

The set of the paths of a tree is denoted by $Path(T)$.

The implementation of these trees in Coq uses a technique similar to the one used for TStreams but is considerably more elaborate due to the treatment of multisuccessors. It is therefore postponed to the chapter on branching temporal logic.

Chapter 3

Linear Temporal Logic

This chapter is devoted to commenting the aspects of the implementation of propositional linear temporal logic (PLTL) in Coq and its use for soundness verification.

3.1 Syntax of PLTL

Given a set P of propositional symbols, the set $PLTLform$ of propositional linear temporal formulae [Eme90] is inductively defined as follows:

1. Each element of P is a formula;
2. If a is a formula then $(\neg a)$ is a formula;
3. If a and b are formulae then $(a \Rightarrow b)$ is a formula;
4. If a is a formula then $(X\ a)$ is a formula;
5. If a and b are formulae then $(a\ U\ b)$ is a formula.

Besides the usual propositional connectives two temporal operators are provided: $(X\ a)$, "next time a" and $(a\ U\ b)$, "a until b". Usual abbreviations of propositional connectives and temporal operators are $(F\ a) \equiv_{abv} ((a \Rightarrow a)\ U\ a)$, "eventually a" and $(G\ a) \equiv_{abv} (\neg(F\ (\neg a)))$, "always in the future a".

It is easy to implement the syntax of propositional linear temporal logic in Coq. Given a set of propositional symbols P, PLTLform is inductively defined by:

```
Require Export TStreams.

Section PLTL.

  Variable P : Set.

  Inductive Type PLTLform :=
      id   :  P → PLTLform
    | no   :  PLTLform → PLTLform
    | imp  :  PLTLform → PLTLform → PLTLform
```

```
| ort  :  PLTLform → PLTLform → PLTLform
| andt :  PLTLform → PLTLform → PLTLform
| X   : PLTLform → PLTLform
| F   : PLTLform → PLTLform
| G   : PLTLform → PLTLform
| U   : PLTLform → PLTLform → PLTLform.
```

It should be noticed that, as in the previous chapter, there are five propositional constructors. Nevertheless, it is possible to define equivalence as an abbreviation using the previous constructors.

Definition `ifft : PLTLform→PLTLform→PLTLform :=`
`[a,b:PLTLform](andt (imp a b) (imp b a)).`

3.2 Semantics of PLTL

The semantics of a formula of propositional linear temporal logic is defined with respect to a linear time structure isomorphic to the natural numbers with their usual ordering $(\mathbb{N}_o, <)$. It is discrete, has an initial moment with no predecessors and is infinite to the future.

The interpretation structures of linear temporal logic are sequences of subsets of the set of propositional symbols. They can be easily implemented using `TStreams` instantiated with:

Definition `Subset_P := P → Prop.`

Thus, the interpretation structures of propositional linear temporal logic are written in Coq as

`(TStream Subset_P).`

The satisfaction of a formula at a given position of an interpretation structure is first defined. The fact that the interpretation structure s satisfies the formula f at the position k is denoted by $(s,k) \Vdash f$. This relation is inductively defined on the structure of the formulae as follows:

- $(s,k) \Vdash a$ iff $a \in s_k$, if $a \in P$;
- $(s,k) \Vdash (\neg a)$ iff it is not the case that $(s,k) \Vdash a$;
- $(s,k) \Vdash (a \Rightarrow b)$ iff not $(s,k) \Vdash a$ or $(s,k) \Vdash b$;
- $(s,k) \Vdash (X\ a)$ iff $(s,k+1) \Vdash a$;
- $(s,k) \Vdash (a\ U\ b)$ there is a $k' \geq k$ such that for all n such that $k \leq n < k'$ $(s,n) \Vdash a$ and $(s,k') \Vdash b$.

Clearly, s_k denotes the subset of the set of propositional symbols in the position k of s.

Such relation is implemented in Coq by the Fixpoint definition of Sat_pos that corresponds to an inductive relation on the structure of the formulae.

```
Fixpoint Sat_pos [p:PLTLform] :  (TStream Subset_P) → nat → Prop :=
  [s:(TStream Subset_P)][k:nat]
    Case p of
      [a:P]           ((TSnth Subset_P k s) a)
      [a:PLTLform]   (¬(Sat_pos a s k))
      [a,b:PLTLform] ((Sat_pos a s k)→(Sat_pos b s k))
      [a,b:PLTLform] ((Sat_pos a s k)∨(Sat_pos b s k))
      [a,b:PLTLform] ((Sat_pos a s k)∧(Sat_pos b s k))
      [a:PLTLform]   (Sat_pos a s (S k))
      [a:PLTLform]   (Ex [k':nat] (ge k' k)∧(Sat_pos a s k'))
      [a:PLTLform]   ((k':nat) (ge k' k)→(Sat_pos a s k'))
      [a,b:PLTLform] (Ex [k':nat] (ge k' k)
                     ∧((n:nat) (ge n k)→(lt n k')→(Sat_pos a s n))
                     ∧(Sat_pos b s k'))
    end.
```

It is now possible to define the satisfaction relation. Given an interpretation structure s and a formula f, s satisfies f (written $s \Vdash f$) iff for all the positions k of s, $(s,k) \Vdash f$. That is, $s \Vdash f$ iff $\forall k \in I\!N_o\ (s,k) \Vdash f$.

The satisfaction of a formula (Sat) is now straightforward in *Gallina*, using the previously defined relation Sat_pos.

```
Definition Sat := [s:(TStream Subset_P)] [f:PLTLform]
  (k:nat)(Sat_pos f s k).
```

A formula is said to be valid iff it is satisfied by all the interpretation structures. This definition is implemented as follows:

```
Definition Valid :  PLTLform → Prop := [f:PLTLform]
  (s:(TStream Subset_P))(Sat s f).
```

The type of subsets of propositional linear temporal formulae is defined as a map from formulae to $\{0,1\}$. The entailment relation is similar to the one in the previous chapter. Given a subset of propositional linear temporal formulae Φ and a formula f, Φ entails f (written $\Phi \models f$) iff for all interpretation structures s, $s \Vdash f$ whenever $\forall \phi \in \Phi$, $s \Vdash \phi$.

The implementation of the type of subsets of the set of propositional linear temporal formulae and the semantic entailment relation Entailment are as in the previous chapter.

```
Definition Subset_PLTLform := PLTLform → Prop.

Definition Entailment :  Subset_PLTLform → PLTLform → Prop :=
  [Phi:Subset_PLTLform] [f:PLTLform]
    (s:(TStream Subset_P))
```

```
(((a:PLTLform)((Phi a) → (Sat s a))) → (Sat s f)).

End PLTL.

Hint Unfold Valid.
Hint Unfold Sat.
Hint Unfold ifft.
```

3.3 Soundness verification of PLTL

It is now possible to use this implementation of the semantics of propositional linear temporal logic to prove the soundness of the axiomatization. The axiomatization used is taken from [Gol92], and includes the axioms:

$Fx1$. $((\neg(X\ a)) \Leftrightarrow (X\ (\neg a)))$

$Fx2$. $((X\ (a \Rightarrow b)) \Leftrightarrow ((X\ a) \Rightarrow (X\ b)))$

$Fx3$. $((G(a \Rightarrow b)) \Rightarrow ((G\ a) \Rightarrow (G\ b)))$

$Fx4$. $((G(a \Rightarrow (X\ a))) \Rightarrow (a \Rightarrow (G\ a)))$

$Fx5$. $((G\ a) \Rightarrow (a \wedge (G\ a)))$

$Fx6$. $((a\ U\ b) \Rightarrow (F\ b))$

$Fx7$. $((G(a\ U\ b)) \Leftrightarrow (b \vee (a \wedge (X\ (a\ U\ b)))))$

and the inference rules:

Nec_X: From a it is derived $(X\ a)$.

Nec_G: From a it is derived $(G\ a)$.

MP: From a and $(a \Rightarrow b)$ it is derived b.

The proofs of the following soundness theorems in Coq are all quite similar. After an hypothesis introduction, the definitions of `Valid` and `Sat` are "unfolded". Then some simplifications are made in order to make the goal more readable. These are the first steps of the proofs. After these the proofs are somehow intuitive.

```
Require Export PLTL.

Require Gt.
```

Section `PLTL_Soundness.`

```
Variable P : Set.
```

Theorem `FX1 :  (a:(PLTLform P))`
```
      (Valid P (ifft P (no P (X P a)) (X P (no P a)))).
```

Theorem `FX2 :  (a,b:(PLTLform P))`
```
     (Valid P (ifft P (X P (imp P a b)) (imp P (X P a) (X P b)))).
```

Theorem `FX3 :  (a,b:(PLTLform P))`
```
      (Valid P (imp P (G P (imp P a b)) (imp P (G P a) (G P b)))).
```

Theorem `FX4 :  (a:(PLTLform P))`
```
      (Valid P (imp P (G P (imp P a (X P a))) (imp P a (G P a)))).
```

Theorem `FX5 :  (a:(PLTLform P))`
```
      (Valid P (imp P (G P a) (andt P a (G P a)))).
```

Theorem `FX6 :  (a,b:(PLTLform P))`
```
      (Valid P (imp P (U P a b) (F P b))).
```

The theorem `FX7` is the most problematic to prove in this section. The proof is briefly explained in the next three paragraphs.

After the simplification of the definitions of `Valid`, `Sat` and simplification of the goal, the tactic `Split` is used so that the implications can be proved separately.

The first one is proved by extracting the witness from the existential quantification in the hypothesis with the command `Inversion H` and then making a case analysis in the `ge` relation with the command `Inversion H1`.

The second one is proved by case analysis on the disjunction hypothesis with the command `Inversion H`. The first case is when the formula is satisfied in the actual position of the interpretation structure. The second is when the formula is satisfied in a future position of the interpretation structure.

Refer to Appendix A for the full proof.

Theorem `FX7 :(a,b:(PLTLform P))`
```
    (Valid P (ifft P (U P a b) (ort P b (andt P a (X P (U P a b)))))).
```

The proofs of the inference rules are as simple as one can get. After "unfolding" the definition of `Sat` and simplifying the goal the proofs are trivial.

Theorem `Nec_X : (a:(PLTLform P))(s:(TStream (Subset_P P)))`
```
   (Sat P s a)→(Sat P s (X P a)).
```

Theorem `Nec_G : (a:(PLTLform P))(s:(TStream (Subset_P P)))`
```
   (Sat P s a)→(Sat P s (G P a)).
```

```
Theorem MP : (a,b:(PLTLform P))(s:(TStream (Subset_P P)))
   (Sat P s a)∧ (Sat P s (imp P a b))→(Sat P s b).
```

The following are the proofs of the abbreviations that were referred to in the section about the syntax of propositional linear temporal logic.

The proof of $(F\ a)\ \equiv_{abv}\ ((a \Rightarrow a)\ U\ a)$ uses both theorems `Abv_F_U1` and `Abv_F_U2` as lemmas.

```
Theorem Abv_F_U1 :(a:(PLTLform P))
   (Valid P (U P (imp P a a) a))→ (Valid P (F P a)).

Theorem Abv_F_U2 :(a:(PLTLform P))
   (Valid P (F P a))→ (Valid P (U P (imp P a a) a)).

Theorem Abv_F_U :(a:(PLTLform P))
   (Valid P (F P a))↔ (Valid P (U P (imp P a a) a)).
```

Note that, in this case, X cannot be given the usual abbreviation using U. In fact U is reflexive and therefore $((\neg(a \Rightarrow a))\ U\ a)\ \equiv_{abv}\ a$.

```
Theorem Abv_a_U :(a:(PLTLform P))
   (Valid P a)↔ (Valid P (U P (no P (imp P a a)) a)).
```

The proof of $(G\ a)\ \equiv_{abv}\ (\neg(F(\neg a)))$ makes use of classical propositional logic. As in the previous chapter the Excluded Middle hypothesis must be introduced. As well, lemma `NNPP` must be proved. These are used in the proof of `Abv_G_F2`.

```
Theorem Abv_G_F1 :(a:(PLTLform P))
   (Valid P (G P a))→ (Valid P (no P (F P (no P a)))).

Hypothesis EM : (X:Prop)(X∨¬X).

Lemma NNPP : (X:Prop)(¬¬X→X).


Theorem Abv_G_F2 :(a:(PLTLform P))
   (Valid P (no P (F P (no P a))))→ (Valid P (G P a)).

Theorem Abv_G_F :(a:(PLTLform P))
   (Valid P (no P (F P (no P a))))↔ (Valid P (G P a)).

End PLTL_Soundness.
```

Both theorems `Abv_G_F1` and `Abv_G_F2` are used as lemmas to prove `Abv_G_F`.

3.4 Application examples

Two application examples are now presented.

The first example concerns the proof of the following lemma:

$\{X\ b\} \models (b \Rightarrow G\ b)$

Its coding and corresponding proof follow.

```
Require Export PLTL_Soundness.

Variable P:Set.

Variable b:(PLTLform P).

Definition Gamma:(Subset_PLTLform P):=[t:(PLTLform P)]
  Cases t of
      (X b) ⇒ True
      | _ ⇒ False
  end.

Lemma eg1 :  (Entailment P Gamma (imp P b (G P b))).
Proof.
  Unfold Entailment.
  Intros.
  Cut (Sat P s (G P (imp P b (X P b)))).
  Intros.
  Cut (Valid P (imp P (G P (imp P b (X P b))) (imp P b (G P b)))).
  Unfold Valid.
  Intros.
  Cut (Sat P s (imp P (G P (imp P b (X P b))) (imp P b (G P b)))).
  Intros.
  EApply MP.
  EAuto.
  Apply H1.
  Apply FX4.
  Cut (Sat P s (imp P b (X P b))).
  Intro.
  Apply Nec_G.
  Assumption.
  Cut (Sat P s (X P b)).
  (Unfold Sat; Simpl).
  Intros.
  Apply H0.
  Apply H.
  Simpl.
  Auto.
```

Qed.

The proof of the lemma eg1 is made bottom up.

It starts by introducing as hypothesis for the interpretation structures that satisfy Gamma, the satisfaction of $(b \Rightarrow (X\ b))$. Then FX4 is introduced as an hypothesis, as well as its particular case for the interpretation structures that satisfy Gamma. By applying MP it is obtained $(b \Rightarrow (G\ b))$. The latter hypothesis introductions are proved using FX4.

By Nec_G it is only necessary to prove that $(b \Rightarrow (X\ b))$ is satisfied by the considered interpretation structures. Unfolding the definition of Sat and using the fact that $(X\ b)$ is true in the considered interpretation structures the proof is concluded.

The second example is the coding and proof of the lemma:

$$\emptyset \models (F(b \Rightarrow (G\ b))).$$

The EM hypothesis and three auxiliary lemmas are used. The first auxiliary lemma not_G_F_not is used to prove the second one not_G_not_F which could be proved earlier as an abbreviation. The third is used as an auxiliary lemma to make the proof smaller and more general.

```
Require Export PLTL_Soundness.

Variable P:Set.

Variable b:(PLTLform P).

Definition Empty :  (Subset_PLTLform P) := [t:(PLTLform P)] False.

Hypothesis EM:(P:Prop)(P∨¬P).

Lemma not_G_F_not :  (a:(PLTLform P))(s:(TStream (Subset_P P)))
      (Sat P s (no P (G P a)))→(Sat P s (F P (no P a))).

Lemma not_G_not_F : (a:(PLTLform P))(s:(TStream (Subset_P P)))
      (Sat P s (no P (G P (no P a))))→(Sat P s (F P a)).

Lemma not_G_and_not_not_G_not_imp :
  (a,c:(PLTLform P))(s:(TStream (Subset_P P)))
    (Sat P s (no P (G P (andt P a (no P c)))))→
              (Sat P s (no P (G P (no P (imp P a c))))).

Lemma eg2 :  (Entailment P Empty (F P (imp P b (G P b)))).
Proof.
  (Unfold Entailment; Intros; Apply not_G_not_F;
     Apply not_G_and_not_not_G_not_imp).
  (Unfold Sat; Simpl; Unfold not; Intros).
  Elim (H0 k).
  Intros.
```

```
  Absurd (k':nat)(ge k' k)->(Sat_pos P b s k').
  Assumption.
  Unfold ge.
  Intros.
  Inversion H3.
  (Replace k' with k; Assumption).
  Elim (H0 (S m)).
  Intros.
  Assumption.
  Unfold ge.
  (Replace (S m) with k'; Assumption).
  Auto.
Qed.
```

The proof of `eg2`, after applying `not_G_not_F` and `not_G_and_not_not_G_not_imp`, is based on the fact that $(\neg(G\ (b \wedge (\neg(G\ b)))))$ is true. Then by absurd it is proved that $(G\ b)$ is not true. The first case of the `Absurd` tactic is just one of the hypothesis. The other one is made by case analysis and using the hypothesis `H0`. This concludes the proof.

Chapter 4

Branching Temporal Logic

The implementation of two branching temporal logics in Coq, UB - Unified system of branching time and CTL - Computation tree logic, and its use for soundness verification are discussed in this chapter. Co-inductive trees of infinite depth and at least one branch (possibly infinite) are used as interpretation structures.

4.1 Time structure

In branching temporal logics time can be seen as an infinite tree. Time is discrete, has an initial moment with no predecessors and at each moment has at least one successor.

Thus, it is possible to characterize a timetree T as a prefix closed subset of ${\mathbb{N}_o}^*$. It is also required a minimality condition for implementation reasons. Refer to Section 2.5.2 for further details. A useful definition is the one of path of a tree required to represent possible futures in the tree. A path is a nonempty subset of a tree, prefix closed and at each moment has an unique successor.

The implementation of these trees in Coq uses also co-inductive constructions. Only this time the co-inductive constructions are defined mutually. Refer to chapter 10 of [PM96] for further explanations.

```
Require Export TStreams.

Require Export utilities.

Require Le.

Require Double.

Section Trees.

  Variable L : Type.

  Mutual CoInductive Tree :  Type :=
                      node :  L → Forest → Tree
         with Forest :  Type :=
```

```
           last :  Tree → Forest
         | next :  Tree → Forest → Forest.
```

```
Definition Path :  Type := (TStream nat).
```

In the previous definitions `L` is the type of labels. Note that the constructor of type `Tree` is `node` for which given a label and an element of type `Forest` gives an element of type `Tree`.

The constructors of type `Forest` are `last` and `next`. Given an element of type `Tree` to the `last` constructor, this one gives a `Forest` which allows to have an unique branch. The `next` constructor may be applied an infinite number of times to itself which allows to have infinite branching or may be applied a finite number of times to itself having as basis the `last` constructor allowing to have finite branching.

The type of `Path` is defined as the type of `TStream` instantiated with `nat`. The n-th element of the `Path` reflects the choice of the branch at depth n of the tree. I. e., if m is the first element of the `Path`, the chosen branch of the root is the m-th, and if k is the second element of the `Path` the chosen branch is the k-th of the previous choice and so on.

However, not all `Path` are paths of a tree. If a tree has finite branching at a given depth, the value of the corresponding element in the considered `Path` may exceed the number of branches of the tree, thus being undefined. It is then necessary to make an extension to trees joining a symbol to represent undefinedness. That is made by creating the `Und` inductive type with an unique element `und`, and then making the disjoint sum of this type with the previous ones, using `TSum` (refer to Apendix A and similar command `Sum` in [PM95]).

```
Inductive Und :  Type :=
    und :  Und.

Definition Label_plus_Und :  Type := (Tsum L Und).

Definition Forest_plus_Und :  Type := (Tsum Forest Und).

Definition Tree_plus_Und :  Type := (Tsum Tree Und).
```

It is now possible to define auxiliary functions for trees, that allow to define a correct notion of path of a tree, the `Path_of_tree` structure.

```
Definition root_with_Und :  Tree_plus_Und→Label_plus_Und :=
  [t:Tree_plus_Und] Cases t of
    (Tinl t') ⇒ Case t' of
                 [a:L][_ :  Forest](Tinl L Und a) end
    |_ ⇒ (Tinr L Und und)
  end.

Definition branches_with_Und :
      Tree_plus_Und → Forest_plus_Und :=
```

```
[t:Tree_plus_Und] Cases t of
  (Tinl t') ⇒ Case t' of
               [_:L][f: Forest] (Tinl Forest Und f)end
  | _ ⇒ (Tinr Forest Und und)
end.
```

```
Definition first_branch : Forest→Tree := [f:Forest]
  Cases f of
     (last t) ⇒ t
    |(next t _) ⇒ t
  end.
```

```
Definition other_branches_with_Und :
  Forest_plus_Und →Forest_plus_Und :=
  [f:Forest_plus_Und] Cases f of
     (Tinl f') ⇒ Cases f' of
                    (last _) ⇒ (Tinr Forest Und und)
                   |(next _ f'') ⇒
                          (Tinl Forest Und f'') end
    | _ ⇒ _
  end.
```

```
Fixpoint nth_branch_with_Und[n:nat] :
  Forest_plus_Und→Tree_plus_Und:=
  [f:Forest_plus_Und] Cases n f of
       O (Tinl f') ⇒ (Tinl Tree Und (first_branch f'))
    | O (Tinr _ ) ⇒ (Tinr Tree Und und)
    |(S p) _ ⇒ (nth_branch_with_Und
                       p (other_branches_with_Und f))
  end.
```

```
CoInductive is_path_of : Tree → Path → Prop :=
  build_i_p_o : (t:Tree)(p:Path) (ExT [t':Tree]
    (nth_branch_with_Und (TShead nat p)
          (branches_with_Und (Tinl Tree Und t)))
     == (Tinl Tree Und t')
    ∧ (is_path_of t' (TStail nat p)))
    → (is_path_of t p) .
```

```
Structure Path_of_tree[t:Tree] : Type :=
  mkpot {
    pot : Path;
    pot_cond : (is_path_of t pot) }.
```

The co-inductive proposition `is_path_of` states that given a `Tree t` and a `Path p`, the elements of `p` never exceed the number of branches of `t` at any depth.

With this proposition the `Structure Path_of_tree` can be defined (refer to section

11.4.8 of [PM96]). Given a `Tree t` the `pot_cond` condition ensures that `pot` is a path of the `Tree t`, never exceeding the number of branches of `t` at any depth.

The definition of a subtree at a given depth of a tree according to a path is made as follows.

```
Fixpoint Subtree_with_Und
  [t:Tree_plus_Und; p:Path; n:nat] :  Tree_plus_Und :=
    Cases n of
      O ⇒ t
    | (S m) ⇒ (Subtree_with_Und
                    (nth_branch_with_Und (TShead nat p)
                             (branches_with_Und t))
                    (TStail nat p)
                    m)
    end.

Theorem Subtree_with_Und_well_def :
  (t:Tree)(p:(Path_of_tree t))(n:nat)
(ExT [t':Tree]
  (Subtree_with_Und (Tinl Tree Und t) (pot t p) n)
    == (Tinl Tree Und t')).
```

The theorem `Subtree_with_Und_well_def` ensures that the previous definition agrees with the expected result, i.e., there is always a corresponding `Tree` to a subtree. Although, with this definition of subtree, it is not possible to extract the corresponding `Tree` of the disjoint sum with the mechanisms furnished by Coq.

But theorem `Subtree_with_Und_well_def` also ensures that with the `Path_of_tree` definition it is possible to define maps over `Tree`. With an extension to be made: every finite `Forest` is extended by repeatedly “adding” the last tree. This extension serves only the `other_branches` definition, having no other use because the number of branches is never exceeded, once a `Path_of_tree` is beeing used in the following `Subtree` definition.

```
Definition root :  Tree→L :=
  [t:Tree]Case t of [l:L][_:Forest]lend.

Definition branches :  Tree → Forest :=
  [t:Tree] Case t of [_:L][f:  Forest] f end.

Definition other_branches :  Forest → Forest :=
  [f:Forest] Cases f of
    (last _) ⇒ (last _)
    |(next _ f') ⇒ f'
  end.

Fixpoint nth_branch[n:nat] :  Forest → Tree :=
  [f:Forest] Cases n of
      O ⇒ (first_branch f)
```

```
    |(S p) ⇒ (nth_branch p (other_branches f))
  end.

Theorem step_is_path_of :
  (t:Tree)(p:(Path_of_tree t))
  (is_path_of (nth_branch (TShead nat (pot t p))
                          (branches t))
              (TStail nat (pot t p))).

Fixpoint Subtree
  [t:Tree; p:(Path_of_tree t); n:nat]:  Tree :=
    Cases n of
      O ⇒ t
    | (S m) ⇒ (Subtree
                (nth_branch (TShead nat (pot t p))
                            (branches t))
                (mkpot (nth_branch
                          (TShead nat (pot t p))
                          (branches t))
                       (TStail nat (pot t p))
                       (step_is_path_of t p))
                m)
    end.

End Trees.
```

Thus, the previous definitions that it is possible to extract the `Tree` corresponding to a `Subtree`. Refer to Appendix A for further details on these definitions.

4.2 Syntax of UB

Given a set of propositional symbols P, the set *UBform* of UB formulae is inductively defined as follows:

- Each element of P is a formula;
- If a is a formula then $(\neg a)$ is a formula;
- If a and b are formulae then $(a \Rightarrow b)$ is a formula;
- If a is a formula then $(\exists G\ a)$ is a formula;
- If a is a formula then $(\exists F\ a)$ is a formula;
- If a is a formula then $(\exists X\ a)$ is a formula.

The basic abbreviations are:

- $(\forall G\ a) \equiv_{abv} (\neg(\exists F\ (\neg a)))$,

- $(\forall F\ a) \equiv_{abv} (\neg(\exists G\ (\neg a)))$,
- $(\forall X\ a) \equiv_{abv} (\neg(\exists X\ (\neg a)))$.

The set of UB formulae is equivalent to the one defined in [Pen95]. The temporal operators have intuitive meanings linked to linear temporal logic: $(\exists G\ a)$ "there is a path where $(G\ a)$", $(\exists F\ a)$ "there is a path where $(F\ a)$" and $(\exists X\ a)$ "there is a path where $(X\ a)$". The abbreviations have dual meanings: $(\forall G\ a)$ "for all paths $(G\ a)$", $(\forall F\ a)$ "for all paths $(F\ a)$" and $(\forall X\ a)$ "for all paths $(X\ a)$".

As in the previous chapters, given a set `P` of propositional symbols, the type of UB formulae `UBform` is inductively defined by:

```
Require Export Trees.
Require Lt.

Section UB.

  Variable P : Set.

  Inductive Type UBform :=
      id   :  P → UBform
    | no   :  UBform → UBform
    | imp  :  UBform → UBform → UBform
    | andt :  UBform → UBform → UBform
    | ort  :  UBform → UBform → UBform
    | EG  : UBform → UBform
    | EF  : UBform → UBform
    | EX  : UBform → UBform
    | AG  : UBform → UBform
    | AF  : UBform → UBform
    | AX  : UBform → UBform.
```

It is also possible to define the equivalence as an abbreviation with the previous constructors.

```
  Definition ifft :  UBform→UBform→UBform :=
    [a,b:UBform](andt (imp a b) (imp b a)).
```

4.3 Semantics of UB

The semantics of a formula of *UBform* is defined with respect to a branching time structure. The considered time structure is the one in Section 4.1.

The interpretation structures are trees labeled with subsets of the set of propositional symbols.

```
  Definition Subset_P := P → Prop.
```

Thus, the interpretation structures of propositional linear temporal logic are implemented in Coq as

```
(Tree Subset_P).
```

The satisfaction of a formula at the root of an interpretation structure is first defined. The fact that the interpretation structure T satisfies the formula f at its root is denoted by $T \Vdash_o f$. This relation is inductively defined on the structure of the formulae as follows:

- $T \Vdash_o a$ iff $a \in \lambda_T(\varepsilon)$, if $a \in P$;
- $T \Vdash_o (\neg a)$ iff is not the case that $T \Vdash_o a$, if $a \in \mathit{UBform}$;
- $T \Vdash_o (a \Rightarrow b)$ iff not $T \Vdash_o a$ or $T \Vdash_o b$, if $a, b \in \mathit{UBform}$;
- $T \Vdash_o (\exists G\ a)$ iff $\exists p \in \mathit{Path}(T)$ s.t. $\forall w \in p$, $\{u \in I\!N_o{}^* : w.u \in T\} \Vdash_o a$, if $a \in \mathit{UBform}$;
- $T \Vdash_o (\exists F\ a)$ iff $\exists p \in \mathit{Path}(T)$ s.t. $\exists w \in p$, $\{u \in I\!N_o{}^* : w.u \in T\} \Vdash_o a$, if $a \in \mathit{UBform}$;
- $T \Vdash_o (\exists X\ a)$ iff $\exists p \in \mathit{Path}(T)$ s.t. $\exists w \in p$ with $|w| = 1$, $\{u \in I\!N_o{}^* : w.u \in T\} \Vdash_o a$, if $a \in \mathit{UBform}$.

Clearly, $\lambda_T(\varepsilon)$ denotes the subset of the set of propositional formulae at the root of T.

Such relation is implemented in Coq by the Fixpoint definition of Sat_pos that corresponds to an inductive relation over the structure of the formulae.

```
Fixpoint Sat_pos [f:UBform] :  (Tree Subset_P)→Prop :=
  [t:(Tree Subset_P)] Case f of
    [a:P]        ((root Subset_P t) a)
    [a:UBform]   (¬(Sat_pos a t))
    [a,b:UBform] ((Sat_pos a t) → (Sat_pos b t))
    [a,b:UBform] ((Sat_pos a t) ∧ (Sat_pos b t))
    [a,b:UBform] ((Sat_pos a t) ∨ (Sat_pos b t))
    [a:UBform]   (ExT [p:(Path_of_tree Subset_P t)]
      (n:nat)(Sat_pos a (Subtree Subset_P t p n)))
    [a:UBform]   (ExT [p:(Path_of_tree Subset_P t)]
      (Ex [n:nat] (Sat_pos a (Subtree Subset_P t p n))))
    [a:UBform]   (ExT [p:(Path_of_tree Subset_P t)]
       (Sat_pos a (Subtree Subset_P t p (S O))))
    [a:UBform]   ((p:(Path_of_tree Subset_P t))
       (n:nat)(Sat_pos a (Subtree Subset_P t p n)))
    [a:UBform]   ((p:(Path_of_tree Subset_P t))
       (Ex [n:nat](Sat_pos a (Subtree Subset_P t p n))))
    [a:UBform]   ((p:(Path_of_tree Subset_P t))
       (Sat_pos a (Subtree Subset_P t p (S O))))
  end.
```

It is now possible to define the satisfaction relation. Given an interpretation structure T and a formula f, T satisfies f denoted by $T \Vdash f$ iff for all $p \in Path(T)$, and $w \in p$, $\{u \in I\!N_o{}^* : w.u \in T\} \Vdash_o f$. This corresponds to the following implementation in Coq:

```
Definition Sat :  (Tree Subset_P)→UBform→Prop:=
  [t:(Tree Subset_P)][a:UBform]
    ((p:(Path_of_tree Subset_P t))(n:nat)
      (Sat_pos a (Subtree Subset_P t p n))).
```

An UB formula f is said to be valid if it is satisfied by all the interpretation structures. This is simply implemented by:

```
Definition Valid :  UBform → Prop := [a:UBform]
  ((t:(Tree Subset_P))(Sat t a)).
```

The type of subsets of the set of UB formulas is defined as a map from *UBform* to $\{0, 1\}$. Given a subset of the UB formulae, Φ, and a UB formula, f, then Φ entails f, which is written $\Phi \models f$, iff for all interpretation structures T, $T \Vdash f$ whenever $T \Vdash a$ for each $a \in \Phi$.

These two concepts are implemented as follows:

```
Definition Subset_UBform := UBform → Prop.

Definition Entailment :  Subset_UBform → UBform → Prop :=
  [Phi:Subset_UBform][f:UBform]
    (t:(Tree Subset_P))
      (((a:UBform)((Phi a) → (Sat t a))) → (Sat t f)).

End UB.
```

4.4 Soundness verification of UB

The defined semantics of UB logic is now used to prove the soundness of the axiomatization in [Pen95], including the axioms:

$UBAx1$. $((\exists X\ (a \vee b)) \Leftrightarrow ((\exists X\ a) \vee (\exists X\ b)))$

$UBAx2$. $((\exists F\ a) \Leftrightarrow (a \vee (\exists X\ (\exists F\ a))))$

$UBAx3$. $((\exists G\ a) \Leftrightarrow (a \wedge (\exists X\ (\exists G\ a))))$

$UBAx4$. $(\exists X\ (a \Rightarrow a))$

and inference rules:

MP: From a and $(a \Rightarrow b)$ it is derived b.

$UBR2$: From $(a \Rightarrow b)$ it is derived $((\exists X\ a) \Rightarrow (\exists X\ b))$.

$UBR3$: From $(a \Rightarrow ((\neg b) \wedge (\exists X\ a)))$ it is derived $(a \Rightarrow (\exists G\ (\neg b)))$.

$UBR4$: From $(a \Rightarrow ((\neg b) \wedge (\forall X\ (a \vee (\neg(\exists F\ b))))))$ it is derived $(a \Rightarrow (\neg(\exists F\ b)))$.

These axioms and inference rules correspond to the following soundness theorems in Coq. The proofs are quite intuitive once the interpretation structures are fully understood.

```
Require Export UB.

Section UB_Soundness.

  Variable P : Set.

  Theorem UBAx1 :  (a,b:(UBform P))
      (Valid P (ifft P (EX P (ort P a b)) (ort P (EX P a) (EX P b)))).

  Theorem UBAx2 :  (a:(UBform P))
      (Valid P (ifft P (EF P a) (ort P a (EX P (EF P a))))).

  Theorem UBAx3 :  (a:(UBform P))
        (Valid P (ifft P (EG P a) (andt P a (EX P (EG P a))))).

  Theorem UBAx4 :  (a:(UBform P))(Valid P (EX P (imp P a a))).

  Theorem MP : (a,b:(UBform P))(t:(Tree (Subset_P P)))
     (Sat P t a)∧ (Sat P t (imp P a b))→(Sat P t b ).

  Theorem UBR2:(a,b:(UBform P))(t:(Tree (Subset_P P)))
       (Sat P t (imp P a b))→(Sat P t (imp P (EX P a) (EX P b))).

  Theorem UBR3 :  (a,b:(UBform P))(t:(Tree (Subset_P P)))
    (Sat P t (imp P a (andt P (no P b) (EX P a))))
      →(Sat P t (imp P a (EG P (no P b)))).

  Theorem UBR4 :  (a,b:(UBform P))(t:(Tree (Subset_P P)))
    (Sat P t (imp P a (andt P (no P b)
                     (AX P (ort P a (no P (EX P b)))))))
        → (Sat P t (imp P a (no P (EG P b)))).

End UB_Soundness.
```

Nevertheless, at this stage, theorem UBR3 is still not proved due to technical difficulties passing from inductive definitions to co-inductive definitions. Refer to Appendix A for further details.

4.5 Syntax of CTL

As usual given a set of propositional symbols P, the set *CTLform* of CTL formulae is inductively defined as follows and it is equivalent to the one in [Pen95] :

- Each element of P is a formula;
- If a is a formula then $(\neg a)$ is a formula;
- If a and b are formulae then $(a \Rightarrow b)$ is a formula;
- If a and b are formulae then $(\exists(a\ U\ b))$ is a formula;
- If a and b are formulae then $(\forall(a\ U\ b))$ is a formula;
- If a is a formula then $(\exists X\ a)$ is a formula.

The basic abbreviations are:

- $(\forall X\ a) \equiv_{abv} (\neg(\exists X\ (\neg a)))$;
- $(\exists F\ a) \equiv_{abv} (\exists((a \Rightarrow a)\ U\ a))$;
- $(\forall F\ a) \equiv_{abv} (\forall((a \Rightarrow a)\ U\ a))$;
- $(\exists G\ a) \equiv_{abv} (\neg(\forall(F\ (\neg a)))$;
- $(\forall G\ a) \equiv_{abv} (\neg(\exists(F\ (\neg a)))$.

The meaning of the temporal operators should now be straightforward.

It is now introduced the corresponding `Coq` implementation, given a set `P` of propositional symbols, the type of CTL formulae `CTLform` is inductively defined:

```
Require Trees.
Require Lt.

Section CTL.

  Variable P:Set.

  Definition Subset_P := P → Prop.

  Inductive Type CTLform :=
      id   :  P → CTLform
    | no   :  CTLform → CTLform
    | imp  :  CTLform → CTLform → CTLform
    | andt :  CTLform → CTLform → CTLform
    | ort  :  CTLform → CTLform → CTLform
    | EU  : CTLform → CTLform → CTLform
```

```
| AU  : CTLform → CTLform → CTLform
| EX  : CTLform → CTLform
| AX  : CTLform → CTLform
| EF  : CTLform → CTLform
| AF  : CTLform → CTLform
| EG  : CTLform → CTLform
| AG  : CTLform → CTLform.
```

Once again the equivalence is introduced as an abbreviation of the previous constructors.

```
Definition ifft :  CTLform→CTLform→CTLform :=
  [a,b:CTLform](andt (imp a b) (imp b a)).
```

4.6 Semantics of CTL

As in the UB logic, the semantics of a formula of *CTLform* is defined with respect to a branching time structure. The interpretation structures are as in UB logic.

The fact that an interpretation structure T satisfies a formula f at its root is denoted by $T \Vdash_o f$. This relation is inductively defined on the structure of the formulae as follows:

- $T \Vdash_o a$ iff $a \in \lambda_T(\varepsilon)$, if $a \in P$;
- $T \Vdash_o (\neg a)$ iff is not the case that $T \Vdash_o a$, if $a \in$ *CTLform*;
- $T \Vdash_o (a \Rightarrow b)$ iff not $T \Vdash_o a$ or $t \Vdash_t b$, if $a, b \in$ *CTLform*;
- $T \Vdash_o (\exists(a\ U\ b))$ iff $\exists p \in Path(T)$ s.t. $\exists w \in p$, $\forall w' \in p$, $|w'| < |w|$, $\{u \in I\!N_o{}^* : w'.u \in T\} \Vdash_o a$ and $\{u \in I\!N_o{}^* : w.u \in T\} \Vdash_o b$, if $a, b \in$ *CTLform*;
- $T \Vdash_o (\forall(a\ U\ b))$ iff $\forall p \in Path(T)$, $\exists w \in p$, $\forall w' \in p$, $|w'| < |w|$, $\{u \in I\!N_o{}^* : w'.u \in T\} \Vdash_o a$ and $\{u \in I\!N_o{}^* : w.u \in T\} \Vdash_o b$, if $a, b \in$ *CTLform*;
- $T \Vdash_o (\exists X\ a)$ iff $\exists p \in Path(T)$ s.t. $\exists w \in p$ with $|w| = 1$, $\{u \in I\!N_o{}^* : w.u \in T\} \Vdash_o a$, if $a \in$ *CTLform*.

Where $\lambda_T(\varepsilon)$ denotes subset of the set of propositional formulae at the root of T.

Such relation is implemented in Coq by the `Fixpoint` definition of `Sat_pos` that corresponds to an inductive relation over the structure of the formulae.

```
Fixpoint Sat_pos [f:CTLform] :  (Tree Subset_P)→Prop :=
  [t:(Tree Subset_P)] Case f of
    [a:P]         ((root Subset_P t) a)
    [a:CTLform]  (¬(Sat_pos a t))
    [a,b:CTLform] ((Sat_pos a t) → (Sat_pos b t))
    [a,b:CTLform] ((Sat_pos a t) ∧ (Sat_pos b t))
    [a,b:CTLform] ((Sat_pos a t) ∨ (Sat_pos b t))
    [a,b:CTLform] (ExT [p:(Path_of_tree Subset_P t)]
```

```
      (Ex [n:nat] (ge n O)∧
        ((n':nat) (ge n' O)→(lt n' n)→
             (Sat_pos a (Subtree Subset_P t p n')))
        ∧(Sat_pos b (Subtree Subset_P t p n))))
   [a,b:CTLform] ((p:(Path_of_tree Subset_P t))
      (Ex [n:nat] (ge n O)∧
        ((n':nat) (ge n' O)→(lt n' n)→
             (Sat_pos a (Subtree Subset_P t p n')))
        ∧(Sat_pos b (Subtree Subset_P t p n))))
   [a:CTLform]  (ExT [p:(Path_of_tree Subset_P t)]
      (Sat_pos a (Subtree Subset_P t p (S O))))
   [a:CTLform]  ((p:(Path_of_tree Subset_P t))
      (Sat_pos a (Subtree Subset_P t p (S O))))
   [a:CTLform]  (ExT [p:(Path_of_tree Subset_P t)]
     (Ex [n:nat] (Sat_pos a (Subtree Subset_P t p n))))
   [a:CTLform]  ((p:(Path_of_tree Subset_P t))
      (Ex [n:nat](Sat_pos a (Subtree Subset_P t p n))))
   [a:CTLform]  (ExT [p:(Path_of_tree Subset_P t)]
     (n:nat)(Sat_pos a (Subtree Subset_P t p n)))
   [a:CTLform]  ((p:(Path_of_tree Subset_P t))
      (n:nat)(Sat_pos a (Subtree Subset_P t p n)))
 end.
```

Once more the satisfaction relation is defined using the satisfaction at the root of an interpretation structure. Given an interpretation structure T and a formula f, T satisfies f denoted by $T \Vdash f$ iff for all $p \in Path(T)$, $\forall w \in p$ is s.t. $\{u \in I\!N_o{}^* : w.u \in T\} \Vdash_o f$. The implementation in Coq is as follows:

```
Definition Sat :  (Tree Subset_P)→CTLform→Prop:=
  [t:(Tree Subset_P)][a:CTLform]
    ((p:(Path_of_tree Subset_P t))(n:nat)
      (Sat_pos a (Subtree Subset_P t p n))).
```

A formula f is said to be valid if it is satisfied by all the interpretation structures. This is implemented as:

```
Definition Valid :  CTLform → Prop := [a:CTLform]
  ((t:(Tree Subset_P))(Sat t a)).
```

The type of subsets of *CTLform* is defined as a map from $CTLform$ to $\{0,1\}$. Given a subset Φ of $CTLform$ and $f \in CTLform$, then Φ entails f, which is written $\Phi \models f$, if for all interpretation structures T, $T \Vdash f$ whenever $T \Vdash a$ for each $a \in \Phi$. The following implements these definitions in Coq:

```
Definition Subset_CTLform := CTLform → Prop.

Definition Entailment :  Subset_CTLform → CTLform → Prop :=
  [Phi:Subset_CTLform][f:CTLform]
```

```
    (t:(Tree Subset_P))
      (((a:CTLform)((Phi a) → (Sat t a))) → (Sat t f)).

End CTL.
```

4.7 Soundness verification of CTL

It is now possible to use the CTL semantics to prove the axiomatization in [Pen95], which includes axioms:

$CTLAx1$. $((\exists X\ (a \vee b)) \Leftrightarrow ((\exists X\ a) \vee (\exists X\ b)))$

$CTLAx2$. $((\exists(a\ U\ b) \Leftrightarrow (b \vee (a \wedge (\exists X\ (\exists(a\ U\ b))))))$

$CTLAx3$. $((\forall(a\ U\ b) \Leftrightarrow (b \vee (a \wedge (\forall X\ (\forall(a\ U\ b))))))$

$CTLAx4$. $(\exists X\ (a \Rightarrow a))$

and the inference rules:

MP: From a and $(a \Rightarrow b)$ it is derived b.

$CTLR2$: From $(a \Rightarrow b)$ it is derived $((\exists X\ a) \Rightarrow (\exists X\ b))$.

$CTLR3$: From $(a \Rightarrow ((\neg b) \wedge (\exists X\ a)))$ it is derived $(a \Rightarrow (\neg(\forall(c\ U\ b))))$.

$CTLR4$: From $(a \Rightarrow ((\neg b) \wedge (\forall X\ (a \vee (\neg(\exists(c\ U\ b))))))$ it is derived $(a \Rightarrow (\neg(\exists(c\ U\ b))))$.

The previous axioms and inference rules correspond to the following soundness theorems in Coq. The proofs are in every aspect similar to the ones of UB.

```
Require Export CTL.

Section CTL_Soundness.

  Variable P : Set.

  Theorem CTLAx1 :  (a,b:(CTLform P))
    (Valid P (ifft P (EX P (ort P a b)) (ort P (EX P a) (EX P b)))).

  Theorem CTLAx2 :  (a,b:(CTLform P))
    (Valid P (ifft P (EU P a b) (ort P b (andt P a (EX P (EU P a b)))))).

  Theorem CTLAx3 : (a,b:(CTLform P))
    (Valid P (ifft P (ort P b (andt P a (AX P (AU P a b)))) (AU P a b))).
```

```
Theorem CTLAx4 :  (a:(CTLform P))
    (Valid P (EX P (imp P a a))).

Theorem MP : (a,b:(CTLform P))(t:(Tree (Subset_P P)))
   (Sat P t a)∧ (Sat P t (imp P a b))→(Sat P t b).

Theorem CTLR2:(a,b:(CTLform P))(t:(Tree (Subset_P P)))
     (Sat P t (imp P a b))→(Sat P t (imp P (EX P a) (EX P b))).

Theorem CTLR3 :  (a,b,c:(CTLform P))(t:(Tree (Subset_P P)))
  (Sat P t (imp P a (andt P (no P b) (EX P a))))
      -> (Sat P t (imp P a (no P (AU P c b)))).

Theorem CTLR4 :  (a,b,c:(CTLform P))(t:(Tree (Subset_P P)))
  (Sat P t (imp P a (andt P (no P b)
                    (AX P (ort P a (no P (EU P c b)))))))
      → (Sat P t (imp P a (no P (EU P c b)))).

End CTL_Soundness.
```

Again, theorems `CTLAx3`, `CTLR3` and `CTLR4` are still not proved due to the same difficulties of `UBR3`. Refer to Appendix A for further details.

Chapter 5

Concluding remarks

The language and the semantics of linear and branching temporal logic were implemented in the `Coq` system. This system was used as a meta-language for representing the proposed logics. Soundness verification of the proposed logics was made using the `Coq` proof system.

The implementation of linear temporal logic semantics was used to verify soundness of the axiomatization and was successfully accomplished has initially proposed. The interpretation structures were simply implemented using a prior knowledge of type `Stream` and making some adjustments to create a new type `TStream`. Some application examples were presented, that can be viewed as tests of expressiveness of the implemented semantics.

Two branching temporal logics – UB and CTL – were presented. A new type `Tree` was used to implement the interpretation structures of these logics. This type was harder to implement due to its added complexity. Some problems arose while verifying the soundness of the axiomatization. Due to technical difficulties in passing from a finite hypothesis to an infinite definition the proofs of `CTLAx3`, `UBR3`, `CTLR3` and `CTLR4` were not completed.

However, in spite of the latter difficulties, this work achieved a significant part of its objectives. Both linear and branching temporal logics semantics were successfully implemented. Soundness verification of the axiomatization of linear temporal logic was completed. Part of the soundness verification of branching temporal logics was also achieved.

Finally, although at a very early stage, this work can be considered as a starting point to more complex implementations, such as distributed temporal logics.

Bibliography

[Coq97] T. Coquand. Course notes in typed lambda calculus. Technical report, Chalmers University, 1997.

[Eme90] E. Emerson. Temporal and modal logic. In J. van Leeuwen, editor, *Handbook of theoretical computer science*, pages 995–1072. Elsevier, 1990.

[Gol92] R. Goldblatt. *Logics of time and computation*. CSLI, 1992.

[HKPM96] G. Huet, G. Kahn, and C. Paulin-Mohring. *The Coq Proof Assistant: A Tutorial*. INRIA Rocquencourt, 1996.

[INR96] INRIA Rocquencourt. *The Coq Proof Assistant: The Standard Library*, 1996.

[Men79] E. Mendelson. *Introduction to Mathematical Logic*. Van Nostrand, 1979.

[Pen95] W. Penczek. Branching time and partial order in temporal logics. In L. Bolc and A. Szalas, editors, *Time & logic: a computational approach*, pages 179–228. UCL, 1995.

[PM95] C. Paulin-Mohring. Circuits as streams in Coq: Verification of a sequential multiplier. Technical report, ENS Lyon, 1995.

[PM96] C. Paulin-Mohring. *The Coq Proof Assistant: Reference Manual V6.1*. INRIA Rocquencourt, 1996.

[Sel92] J. Seldin. Coquand's calculus of constructions: A mathematical foundation for a proof development system. *Formal Aspects of Computing*, 4:425–441, 1992.

[Ser93] C. Sernadas. *Introdução à teoria da computação*. Presença, 1993.

[SSR93] A. Sernadas, C. Sernadas, and J. Ramos. Folhas de Elementos Lógicos da Programação I, Secção de Ciência da Computação, DMIST, 1993.