An Axiomatization of Linear Temporal Logic in the Calculus of Inductive Constructions

Solange Coupet-Grimal *
Université de Provence, Marseille

December 9, 2002

Abstract

We present in this paper a shallow embedding of Linear Temporal Logic in the Calculus of Inductive Constructions. Our axiomatization is based on a co-inductive representation of program executions. Temporal operators are implemented as co-inductive (respectively inductive) types when they are greatest (respectively least) fixpoints, and several generic lemmas are proved to allow elegant and efficient reasoning in practical cases. This work results in several reusable libraries in the Coq proof-assistant.

Keywords Formal methods, type theory, temporal logic, co-induction.

1 Introduction

Formal specification combined with mechanical verification is a profitable approach for achieving the high levels of assurance demanded of safety-critical systems. Moreover, establishing the correctness of such systems often requires reasoning about time, and, in this framework, temporal logics are now widely used.

In this paper, we present an axiomatization of Linear Temporal Logic (LTL) in the Calculus of Inductive Constructions.

The motivation for choosing this logical framework is threefold. First, its great expressiveness makes it possible to give clear, accurate and generic specifications, to reason elegantly on them, and to obtain general and reusable results. Second, it rests on firm logical foundations. Third, it is implemented as a proof assistant, the Coq system [16]. The latter, as it relies on a small kernel of rules, can be regarded as very reliable.

As far as temporal logics are concerned, most systems can be classified along various axes such as branching versus linear, higher order versus first order or propositional,

*Laboratoire d'Informatique Fondamentale de Marseille (UMR 6166), CMI, 39 rue Joliot-Curie, F-13453, Marseille, France. Solange.Coupet@cmi.univ-mrs.fr.

or past versus future-tense logics. It is a matter of debate whether branching or linear time is preferable (see for example [17]). The μ-Calculus ([7]), which subsumes LTL, CTL*, and CTL, provides an elegant framework for reasoning about theoretical aspects of temporal logics. However, it is not really convenient for actual case studies. Our emphasis is on developing an apparatus that is useful in practice and from this point of view, higher order linear temporal logic is expressive enough to handle the actual applications ([18]) and being more natural, closer to the designer's intuition, it is pleasant to work with. Finally, the complexity of model-checking, which is an argument in favor of the branching time paradigm, is not relevant in our approach.

We opt for a shallow embedding of the logic in which program executions are represented by infinite co-inductive lists and temporal operators as co-inductive or inductive types, depending on whether they are greatest or least fixpoints. This leads to clean and natural specifications, without any time parameter, and to an elegant way of reasoning. Consequently, we work with an "anchored" version of the logic, in which only formulas that hold at position 0 in a program execution are considered. Let us mention that this approach does not make possible to define past operators. But this is not a real drawback since it has been proved that the future fragment of Temporal Logic is expressively complete ([4]).

In our axiomatization, the rules of LTL correspond to lemmas that express either termination properties or invariance properties that are proved by induction or co-induction. This work results in several reusable Coq libraries that can be found in *The Users's Contributions* [2]. Let us mention that the feasibility of this approach is demonstrated in [3], by a non trivial verification of an incremental *mark-and-sweep* garbage collector.

This paper is self-contained and organized as follows. In section 2, we give a short overview of the Calculus of (Co)Inductive Constructions (CC). Section 3 is dedicated to the axiomatization of LTL in CC : infinite transition systems, temporal operators, and several properties of program executions (such as fairness) are defined. In section 4 we prove LTL rules and termination lemmas. Related work is discussed in section 5, and the conclusion is given in the same section.

2 A brief overview of the Calculus of (co-)Inductive Constructions

The Calculus of Constructions (CoC) has been introduced by T.Coquand and G.Huet [1] and enriched with inductive types by C.Paulin-Mohring [12] and co-inductive types by E. Giménez [5]. Simultaneously, successive versions of the system were implemented, resulting in the proof assistant Coq [16].

The Calculus of Constructions

CoC is a higher order λ-calculus which provides in a uniform logical framework both a functional programming language for specifying and a higher order intuitionistic logic for reasoning about specifications, via the Curry-Howard isomorphism. The system relies on a dual interpretation of types, either as sets or as propositions (a proposition is in fact considered as the set of its effective proofs). Thus a term `t`, inhabitant of a type `A`, can be both interpreted as an element of the set `A` and as a proof of the proposition `A`. CoC is obtained by enriching the simply typed λ-calculus in the following way. First, one allows types to depend on terms. For that, a sort `*` is introduced and `A:*` means "`A` is a type". New type constructors such as `P:A→*` can be interpreted both as predicates on the set `A` or as a family of sets, indexed by `A`. One can then construct types `(P a)` depending on terms `a` of type `A`. Under the assumptions `x:A, h:(P x)` one can derive $\vdash$ `λx:A.h : (∀x:A)(P x)`. The type `(∀x:A)(P x)` denotes either the product of a family of sets indexed by `A` or a universally quantified proposition. Polymorphism is also introduced by abstracting terms with respect to types. One can then define the polymorphic identity: `(λA:*)(λx:A)x:(∀A:*)A→A` as well as the conjunction `A∧B` of two propositions `A` and `B` by the type: `(∀C:*)(A→B→C)→C`. Finally, abstracting types with respect to types is also possible. The conjunction $\wedge$ for example, can be defined in the following way: `∧ := (λA:*)(λB:*)(∀C:*)(A→B→C)→C`.

Sort * versus sorts `Prop` and `Set`

As the underlying logic is constructive, proofs are effective. This implies in particular that any proof of a statement of the form: `(∀x:A)(∃y:B)(P x y)` is a pair `(f:A→B, p)` where `f` is a program that, for all `x:A`, computes a witness `y=(f x)` and `p` is a proof of the proposition: `(∀x:A)(P x (f x))`. In this sense, such a term can be viewed as a certified program since `f` is accompanied with a "certificate" `p` ensuring that it meets some property. It is then interesting to be able to erase the logical part of the proof term, exactly as a compiler ignores comments, and, in such a way, to get the purely computational component `f` which has been proved to be correct. But this is meaningless in a system as the one we have presented, since proofs and programs are essentially the same objects, namely λ-terms. These considerations lead to splitting the sort `*` into two twin sorts called `Set` and `Prop`, enforcing the distinction between the two possible type semantics. That is how the system Coq can provide a mechanism which automatically extracts from a term the certified computational part.

Induction, Co-induction

Finally the Calculus of (Co)-Inductive Constructions (CC) is obtained by adding to CoC the possibility of defining inductive and co-inductive types. Such a type is characterized by a finite set of typed constructors. Each constructor corresponds to an introduction rule of the underlying natural deduction system. For example, if `A` is of type `Set`, one can define the two following types:

```
Inductive list : Set := nil : list | cons : A→list→list
```

or

```
CoInductive list : Set := nil : list | cons : A→list→list
```

with two constructors `nil` and `cons`.

An induction principle is associated with each inductive type (and is automatically generated by the system Coq). It corresponds to an elimination rule for reasoning on the terms in the free algebra generated by the constructors. The inductive type `list` represents the set of finite lists the elements of which are in `A`. Total functions can be defined on inductive types by structural recursion.

In a dual way, the co-inductive type `list` corresponds to a greatest fixpoint, and denotes the set of all the finite and infinite lists of elements in `A`. The proofs of co-inductive statements are co-recursive terms (representing non ending processes that build infinite objects step by step) which must meet a guard condition to be well-formed. More precisely, a recursive call, within such a term, must occur just under a constructor. This ensures the singleness of the fixpoint.

Let us now illustrate this section by presenting a shallow embedding of Linear Temporal Logic in this logical framework.

3 Axiomatization of Linear Temporal Logic

We have chosen to specify all the temporal notions without introducing any quantification on time parameters. The representation of infinite state sequences as co-inductive lists instead of functions on the set of the successive steps of time (usually modelled by natural numbers) leads to a direct axiomatization of the future temporal operators as least or greatest fixpoints using inductive or co-inductive predicates on these infinite traces.

3.1 Transition Systems

Here are the basic definitions introduced when encoding transition systems. The specification is parameterized by:

- the set `state` of the states and the set `label` of the actions labelling the transitions,
- two predicates `init_state` and `fair` characterizing respectively the initial states and the actions for which a fairness condition is required,
- a function `transition` which associates with each label `a` a binary relation on the states denoted by $\xrightarrow{a}$.

The relations $\xrightarrow{step}$ and $\xrightarrow{fair}$ between states are inductively defined respectively as the union of all the transition relations $\xrightarrow{a}$, `a`$\in$`label` and as the union of all transition relations $\xrightarrow{a}$, `a`$\in$`label` and `(fair a)`. The reflexive closure of the relation $\xrightarrow{step}$ is denoted by $\xrightarrow{=}$ and for all state relations `r` a predicate `(enabled r)` is defined on the states by:

`(enabled r)` :=λ`s:state.` ($\exists$`t:state) (r s t).`

Our encoding relies on the following co-inductive specification of infinite sequences of states (streams) :

```
CoInductive stream:  Set := cons:  state → stream → stream.
```

We then define the temporal formulas as being the predicates on streams (stream formulas).

The head and the tail of a stream σ are respectively denoted by `(hd` σ`)` and `(tl` σ`)`. We will sometimes write σ_0 for `(hd` σ`)` and more generally σ_n for `(hd(tl`^{n}`(`σ`))`.

Every state formula `P` (`P` is a predicate on states) is coerced to the corresponding stream formula $\overline{\texttt{P}}$ as follows: $\forall\sigma$`:stream. (`$\overline{\texttt{P}}$ σ`) := (P` σ_0`)`

For each transition $\xrightarrow{a}$, a predicate `(taken` $\xrightarrow{a}$`)` is defined on the streams by:
`(taken` $\xrightarrow{a}$`)` := $\lambda\sigma$`:stream.` $\sigma_0 \xrightarrow{a} \sigma_1$

The relation `leads_to` on the state formulas is defined by:
`(leads_to P Q) := (`$\forall$`s, t : state) (P s)`$\rightarrow$`(s`$\xrightarrow{step}$`t)`$\rightarrow$`(Q t).`

and the fact that a state formula `P` is invariant if encoded by:
`(invariant P) := (leads_to P P).`

3.2 Temporal Operators

In temporal logics with past operators, the definition of the interpretation of temporal operators ([10]) is based on the notion of a formula `P` holding at a position j ($j \geq 0$), in a sequence σ. This is denoted by $(\sigma,\ j) \models$`P`. This version of the logic, when restricted to formulas without past operators, is equivalent in fact to considering formulas that hold at position 0 in the suffix starting at the jth element of σ. That is why we have chosen a version of LTL in which parameter j is "anchored" at 0, instead of being allowed to "float". Only models σ such that `P` holds at position 0 are considered. The temporal formulas are thus simply predicates on the sequences σ and `(P` σ`)` stands for $(\sigma,\ 0) \models$`P`. The only suffixes to be examined are tails, when using structural recursion on sequences. This approach does not decrease the expressive power of LTL and leads to neat and uniform specifications.

Let us now review the LTL basic operators. Throughout this subsection, `P`, `Q`, and `R` are stream formula variables (of type `stream` $\rightarrow$`Prop`). For each operator, we first

recall the informal definition, then we give our representation in CC.

- **The** *Next* **operator** $\bigcirc$

```
◯P(σ0,σ1,...,σn,...) := P(σ1,...,σn,...).

◯P : stream → Prop := λσ:stream.(P (tl σ)).
```

- **The** *henceforth* **operator** $\Box$

```
□P(σ0,σ1,...,σn,...) := ∀i∈ ℕ P(σi,...,σn,...)

CoInductive □P : stream → Prop :=
            C_always : (∀s:state)(∀σ:stream)
            (P (cons s σ)) →(□P σ)  →(□P (cons s σ))).
```

- **The** *Eventually* **operator** $\Diamond$

```
◇P(σ0,σ1,...,σn,...) := ∃i∈ ℕ P(σi,...,σn,...)

Inductive ◇P : stream → Prop :=
          ev_h : (∀σ:stream)(P σ) →(◇P σ) |
          ev_t : (∀s:state)(∀σ:stream)
                  (◇P σ) →(◇P (cons s σ)).
```

- **The** *Until* **operator** $\mathcal{U}$

```
P𝒰Q(σ0,σ1,...,σn,...) :=
     ∃i∈ ℕ (∀j∈ {0, ...,i-1} P(σj,...,σn,...))∧Q(σi,...,σn,...)

Inductive P 𝒰 Q : stream → Prop :=
          until_h : (∀σ:stream)(Q σ)→((P 𝒰 Q) σ) |
          until_t : (∀s:state)(∀σ:stream)
                     (P (cons s σ))→((P 𝒰 Q) σ)→
                     ((P 𝒰 Q)(cons s σ)).
```

- **The** *Unless* (*Waiting-for*) **operator** $\mathcal{W}$

```
P𝒲Q(σ0,σ1,...,σn,...) :=
     (∃i∈ ℕ (∀j∈ {0, ...,i-1} P(σj,...,σn,...))∧ Q(σi,...,σn,...)) ∨
     ∀i∈ ℕ P(σi,...,σn,...).

CoInductive P 𝒲 Q : stream → Prop :=
          unless_h : (∀σ:stream)(Q σ)→((P 𝒲 Q) σ) |
          unless_t : (∀s:state)(∀σ:stream)
                     (P (cons s σ)) →((P 𝒲 Q) σ)→
                     ((P 𝒲 Q)(cons s σ)).
```

At this point, one can notice the duality between the *Until* and the *Unless* operators. P$\mathcal{U}$Q and P$\mathcal{W}$Q both mean that P holds as long as Q does not hold, but contrary to the latter, the former states that Q will eventually hold. The types of the constructors follow the same syntax. The difference lies in the operator type which is inductive for *Until* and co-inductive for *Unless*.

From these basic definitions, one can obtain classically derived operators such as :

- (infinitely_often P) := $\Box\Diamond$P
- (eventually_permanently P) := $\Diamond\Box$P
- (is_followed P Q) := $\lambda\sigma$:stream. (P σ)$\rightarrow$($\Diamond$Q σ)
- (is_always_followed P Q) := $\Box$(is_followed P Q)
- (P $\hookrightarrow$ Q) := $\Box(\lambda\sigma$:stream. (P σ)$\rightarrow$(Q σ))
- (once_always P Q) := $\Box$(P $\hookrightarrow \Box$Q)
- (leads_to_via P Q R) := P $\hookrightarrow$ (Q $\mathcal{U}$ R)
- (once_until P Q) := (leads_to_via P P Q)

3.3 Infinite Computation Properties

Infinite state sequences are classified using the notions of *trace* and *run*. A trace is a sequence σ_0, σ_1 ,..., σ_i, σ_{i+1},..., such that for each position i, either $\sigma_i = \sigma_{i+1}$, or there exists a label a such that $\sigma_i \xrightarrow{a} \sigma_{i+1}$. This definition allows us to represent finite computations as infinite sequences, with an idle transition looping on the last state. If, in addition, the first element belongs to the set of initial states, the *trace* is called a *run*. Formally we write:

trace := $\Box(\lambda\sigma$:stream. $\sigma_0 \xrightarrow{=} \sigma_1$).
run := $\lambda\sigma$:stream. (trace σ)$\wedge$(init_state σ_0).

Now, a stream formula P is said to be *safe* if it is satisfied by the suffixes of all runs. That is:

(safe P) := $\forall\sigma$:stream. (run σ)$\rightarrow$($\Box$P σ)

As far as fairness is concerned, several notions can be introduced. In the following, we will say that a sequence is a fair stream if it satisfies the process justice requirement as introduced in [10].

fairstr :stream_formula :=
(infinitely_often $\lambda\sigma$.(enabled $\xrightarrow{fair}$ σ_0)$\rightarrow$ (taken $\xrightarrow{fair}$ σ)).

We also introduce a strong fairness condition that requires the fair set of actions to be served infinitely often :

`strong_fair := (infinitely_often (taken` $\xrightarrow{fair}$`))`

Let us now present some of the lemmas we have established for temporal reasoning.

4 Reasoning on temporal formulas

Co-inductive reasoning is a valuable tool that clarifies and simplifies the work when handling temporal aspects of infinite executions properties. Exploiting the duality between induction and co-induction leads to a uniform and systematic treatment of goals and hypotheses. Inductive hypotheses are eliminated whereas co-inductive ones are inverted. Inductive goals are proved by applying constructors (introduction rules) whereas goals with a co-inductive conclusion are established by considering the goal as an hypothesis that can be applied under a constructor of the conclusion (this will be explained in detail in the sequel). The structure of the proof is then closely connected to the structure (defined by means of their own constructors) of the operators involved in the statement under consideration. This is much more understandable than handling a unique time parameter which is shared by all operators.

Let us first focus on co-inductive proofs.

4.1 Safety

Let us consider as a simple example the lemma establishing the idempotence of the $\Box$ operator.

Lemma 1 `(∀P:stream_formula)(∀σ:stream) (□Pσ)` $\rightarrow$ `(□□Pσ).`

Proof A proof term for such a proposition is obtained co-recursively as follows :

`νX.λP λσ λH (C_always s σ' H (X P σ' q))`

where :

- ν is the fixpoint operator
- `P` has type `stream_formula`, σ has type `stream` and `H` is a proof of `(□P σ)`
- σ`=(cons s σ')` where `s` is a `state` and σ' is a `stream` (deconstruction of σ).
- `H=(C_always s σ' p q)` where `p` is a proof of `(P σ)` and `q` is a proof of `(□P σ')` (deconstruction of `H`).

The recursive call `(X P σ'q)` in the proof term above is of type `(□□ P σ')`, that amounts to assuming that the property holds on the tail σ' of σ. This recursive call occurs just under the constructor `C_always` which is the guard condition to be satisfied for the term to be well formed.

Even such a simple example can illustrate the benefit taken from the use of co-inductive types. The usual definition of the $\Box$ operator semantics:

$$\texttt{(}\Box\texttt{P }\sigma\texttt{) := (}\forall\texttt{i}\in I\!N\texttt{)(P }\sigma_{|i}\texttt{)} \text{ where } \sigma_{|i} = (\sigma_i, \ldots, \sigma_n, \ldots)$$

requires manipulating quantifiers and arithmetics on the indices, which obfuscates the temporal reasoning. We would have had to prove:

$$(\forall t \in I\!N)(P\ \sigma_{|t}) \rightarrow (\forall t \in I\!N)(\forall t' \in I\!N)(P\ (\sigma_{|t})_{|t'})$$

and then to handle suffixes of suffixes and to establish some additional properties such as

$$(\forall t \in I\!N)(\forall t' \in I\!N)(\sigma_{|t})_{|t'} = \sigma_{t+t'}$$

Our approach is more elegant in the sense that the temporal part of the reasoning simply consists in deconstructing σ and proving that its tail satisfies the universally quantified formula stated in the lemma, which follows immediately from the co-induction hypothesis. The resulting proof term is extremely terse. In general, the more complex the formula, the more important the gain in clarity.

In our certified library, we establish several results that simplify the work in the subsequent case studies, in particular in treating once and for all the co-inductive temporal aspects of the goal. Let us give some examples. The *safety* theorem is the basis for proving that a state formula is *safe*, i.e. that it is continuously satisfied by all runs.

Theorem 2 (safety)
```
(∀P:state_formula)
((∀s:state) (init_state s) -> (P s)) ->
(invariant P)->
(safe P̄).
```

Proof This theorem is a corollary of the next lemma.

Lemma 3
```
(∀P:state_formula)
(invariant P)->
(∀σ:stream)(trace σ) -> (P σ0) ->(□P̄ σ).
```

Proof The proof is performed by co-induction.

The next result generalizes lemma 1. It reduces a proof of a co-inductive goal of the form `(□P σ)` under a co-inductive hypothesis to a simpler goal `(P σ)`.

Lemma 4
```
(∀P,Q:stream_formula)
(∀σ:stream)((□Q σ)→(P σ))→(∀σ:stream)(□Q σ)→(□P σ).
```

Proof As previously, after having discharged all the hypotheses, the proof of the goal $\Box P\sigma$ is performed in two steps:

- the proof of `(P σ)`
- the proof of `□P(tl σ)`

The first one follows immediately from the hypothesis : $(\forall\sigma{:}\texttt{stream})(\Box Q\sigma \rightarrow P\sigma)$. The second is proved by applying the "co-induction hypothesis" (that is the property stated in the lemma) to $(\texttt{tl}\ \sigma)$.

The next lemma is particularly useful when reasoning on *runs*. Actually, the notion of a *run* involves a condition on the initial state of a stream, which prevents us from reasoning co-inductively on it (unlike the *traces*). In general, the initial condition is needed to prove that a property which is stable by transition, continuously holds on the sequence. Thus we can replace an awkward hypothesis of the form $(\texttt{run}\ \sigma)$ by $(\texttt{trace}\ \sigma)\wedge(\Box I\ \sigma)$ provided that I continuously holds on all the runs (I is *safe*).

Lemma 5 Let P and Q be temporal formulas and let I be a safe property. Then assuming that: $\forall\sigma{:}\texttt{stream}\ (\texttt{trace}\ \sigma)\rightarrow(\Box I\ \sigma)\rightarrow(\Box Q\ \sigma)\rightarrow(P\ \sigma)$
one can prove that $\forall\sigma{:}\texttt{stream}\ (\texttt{run}\ \sigma)\rightarrow(\Box Q\ \sigma)\rightarrow(\Box P\ \sigma)$.

As in lemma 4, this result highly simplifies the future work. To prove that a property P continuously holds on the runs σ, it will be sufficient to prove merely that it holds (initially) on the *traces* that always satisfy a certain $safe$ property.

Let us now consider some lemmas that mix inductive and co-inductive reasoning.

4.2 Monotonicity

Establishing the monotonicity of temporal operators generally requires both induction and co-induction. The monotonicity of the basic temporal operators can be stated as follows.

Theorem 6 Let P, Q, P', and Q' be stream formulas. Then, for all streams σ:
1- $((P\hookrightarrow Q)\ \sigma) \rightarrow ((\Box P)\hookrightarrow(\Box Q)\ \sigma)$
2- $((P\hookrightarrow Q)\ \sigma) \rightarrow ((\Diamond P)\hookrightarrow(\Diamond Q)\ \sigma)$
3- $((P\hookrightarrow P')\ \sigma) \rightarrow ((Q\hookrightarrow Q')\ \sigma) \rightarrow ((P\ \mathcal{U}\ Q)\hookrightarrow(P'\mathcal{U}\ Q')\ \sigma)$
4- $((P\hookrightarrow P')\ \sigma) \rightarrow ((Q\hookrightarrow Q')\ \sigma) \rightarrow ((P\ \mathcal{W}\ Q)\hookrightarrow(P'\mathcal{W}\ Q')\ \sigma)$

This theorem is obtained as a corollary of a simpler result.

Lemma 7 Let P, Q, P', and Q' be stream formulas. Then, for all streams σ:
1- $((P\hookrightarrow Q)\ \sigma) \rightarrow (\Box P\ \sigma)\rightarrow(\Box Q\ \sigma)$
2- $((P\hookrightarrow Q)\ \sigma) \rightarrow (\Diamond P\ \sigma)\rightarrow(\Diamond Q\ \sigma)$
3- $((P\hookrightarrow P')\ \sigma) \rightarrow ((Q\hookrightarrow Q')\ \sigma) \rightarrow ((P\ \mathcal{U}\ Q)\ \sigma)\rightarrow((P'\mathcal{U}\ Q')\ \sigma)$
4- $((P\hookrightarrow P')\ \sigma) \rightarrow ((Q\hookrightarrow Q')\ \sigma) \rightarrow ((P\ \mathcal{W}\ Q)\ \sigma)\rightarrow((P'\mathcal{W}\ Q')\ \sigma)$

Proof of theorem 6. Statements 1 and 2 follow straightforwardly from the corresponding statements in lemma 7 by applying lemma 4. Statements 3 and 4 are obtained from the corresponding statements in lemma 7 by co-induction.

Proof of lemma 7. Classically, the proofs require the inversion of the co-inductive hypotheses. Statements 1 and 4 are proved by co-induction whereas statement 2 is proved by induction on the hypothesis $(\Diamond P\ \sigma)$, and statement 3 by induction on the hypothesis $(P\ \mathcal{U}\ Q)$.

The monotonicity of derived operators follows immediately from that of the basic operators.

4.3 Liveness

As far as liveness is concerned, the next theorem states a useful property of traces that satisfy the weak fairness condition.

Theorem 8 Let `P` and `Q` be state formulas such that `(leads_to P Q)` and that $(\forall$`s:state)(P s)`$\rightarrow$`(enabled` $\stackrel{fair}{\longrightarrow}$ `s)`. Then:
$(\forall$`σ:stream)(trace σ)`$\rightarrow$`(fairstr σ)`$\rightarrow$`(once_until P Q σ)`.

Proof This theorem is obtained by co-induction from the following result.

Lemma 9 Let `P` and `Q` be state formulas such that `(leads_to P Q)` **(1)** and that $(\forall$`s:state)(P s)`$\rightarrow$`(enabled` $\stackrel{fair}{\longrightarrow}$ `s)` **(2)**. Then:
$(\forall$`σ:stream)(trace σ)`$\rightarrow$`(fairstr σ)`$\rightarrow(\overline{\texttt{P}}\,\sigma)\rightarrow(\overline{\texttt{P}}\,\mathcal{U}\,\overline{\texttt{Q}}\,\sigma)$.

Proof Let σ be a trace such that `(fairstr σ)`. It follows from the definition of `fairstr` that: `(`$\Diamond$`(`$\lambda\tau$`(enabled` $\stackrel{fair}{\longrightarrow}$ τ_0`)`$\rightarrow$`(`τ_0 $\stackrel{fair}{\longrightarrow}$ τ_1`)) σ)`. We prove by induction on this term that σ satisfies `(trace σ)`$\rightarrow(\overline{\texttt{P}}\,\sigma)\rightarrow(\overline{\texttt{P}}\,\mathcal{U}\,\overline{\texttt{Q}}\,\sigma)$.

Base case. Let us assume that `(enabled` $\stackrel{fair}{\longrightarrow}$ σ_0`)`$\rightarrow$`(`σ_0 $\stackrel{fair}{\longrightarrow}$ σ_1`)`. From $(\overline{\texttt{P}}\,\sigma)$ we have `(P` σ_0`)`. Therefore, it follows from the hypothesis **(2)** that `(enabled` $\stackrel{fair}{\longrightarrow}$ σ_0`)`, and thus that $\sigma_0 \stackrel{fair}{\longrightarrow} \sigma_1$ hold. From **(1)** we can deduce `(Q` σ_1`)`. Then, obviously the formula $(\overline{\texttt{P}}\,\mathcal{U}\,\overline{\texttt{Q}}\,\sigma)$ holds.

Induction step. Let `s` be a state and σ be a stream such that:

- H1 : `(`$\Diamond$`(`$\lambda\tau$`(enabled` $\stackrel{fair}{\longrightarrow}$ τ_0`)`$\rightarrow$`(`τ_0 $\stackrel{fair}{\longrightarrow}$ τ_1`)) σ)`
- H2 : `(trace σ)`$\rightarrow$ $(\overline{\texttt{P}}\,\sigma)\rightarrow(\overline{\texttt{P}}\,\mathcal{U}\,\overline{\texttt{Q}}\,\sigma)$
- H3 : `(trace (cons s σ))`
- H4 : $(\overline{\texttt{P}}$`(cons s σ))`

Let us prove that $(\overline{\texttt{P}}\,\mathcal{U}\,\overline{\texttt{Q}}$ `(cons s σ))` holds. As `s` satisfies `P` from H4, it is sufficient to establish $(\overline{\texttt{P}}\,\mathcal{U}\,\overline{\texttt{Q}}\,\sigma)$, that is, from H2, that σ is a trace that satisfies `(P` σ_0`)`. From H3 we know that σ is a trace. Moreover, as `(cons s σ)` is a trace, either σ_0`= s` or $s \stackrel{step}{\longrightarrow} \sigma_0$. In the first case, `(P` σ_0`)` follows from H4. In the second case, from **(1)**, we have `(Q` σ_0`)`, and then $(\overline{\texttt{P}}\,\mathcal{U}\,\overline{\texttt{Q}}$ `(cons s σ))` also holds.

The CC underlying logic being intuitionistic, it is worth establishing some properties on decidable temporal formulas such as the following one.

Lemma 10 Let `P` be a decidable predicate on the streams, that is such that:

$$(\forall\texttt{σ:stream})(\texttt{P}\ \sigma)\vee\neg(\texttt{P}\ \sigma) \quad .$$

Then, for all streams σ,

$$\texttt{(is\_followed P }\neg\texttt{P }\sigma)\rightarrow(\texttt{P }\mathcal{U}\neg\texttt{P }\sigma) \quad .$$

Proof If $\neg$`(P σ)` holds, the goal is proved immediately by applying the first constructor. If `(P σ)` holds, we can deduce `(`$\Diamond\neg$`P σ)`. The goal is proved by induction on this term.

Liveness properties are often expressed by means of `leads_to_via` operators (as examplified in [3]). The next lemma makes it possible to decompose complex liveness proofs into simpler ones.

Lemma 11 Let `A`, `B`, `C`, `D`, and `E` stream formulas and σ a stream. Then:
`(leads_to_via A B C)`$\rightarrow$
`(leads_to_via C D E )`$\rightarrow$
`(leads_to_via A`$\vee$`C B`$\vee$`D E )`.

Among the liveness lemmas, those of the next subsection are essential in order to establish that a program eventually terminates.

4.4 Termination

The main termination lemma is classical. It makes it possible to conclude that, provided that a property `A` initially holds on a stream, a property `B` continuously holds *until* a property `C` is eventually satisfied. The proof is based on the existence of a measure on the states ranging over a set equipped with a well-founded relation. We assume that eventually, either the measure strictly decreases, or `C` becomes true. Meanwhile, `B` remains true.

Theorem 12 Let `Alpha` be a set, $\prec$ a well-founded relation on `Alpha` and `meas` a measure specified as a relation of type `state`$\rightarrow$`Alpha`$\rightarrow$`Prop`. Let `P`, `Q`, and `R` be state formulas, `v` a value in `Alpha` and σ a stream. If σ satisfies the formula
`(leads_to_via` $\lambda\sigma$`.(P` σ_0`)`$\wedge$`(meas` σ_0 `v)`
$\overline{\texttt{Q}}$
`(`$\lambda\sigma$`.(P` σ_0`)`$\wedge$`((`$\exists$`t:Alpha)(meas` σ_0 `t)`$\wedge$`(t`$\prec$`v))) `$\vee$` (R` σ_0`))`
then it satisfies:
`(leads_to_via (`$\lambda\sigma$`.(P` σ_0`) `$\wedge$`(`$\exists$ `v:Alpha)(meas` σ_0 `v))` $\overline{\texttt{Q}}$ $\overline{\texttt{R}}$`)`

Proof It is obtained by co-induction from the following lemma.

Lemma 13 Under the same hypotheses as those in lemma 12, the stream σ satisfies:
`(P` σ_0`)` $\wedge$`(`$\exists$`v:Alpha)(meas` σ_0 `v)` $\rightarrow$ `(`$\overline{\texttt{Q}}\,\mathcal{U}\,\overline{\texttt{R}}\,\sigma$`)`

Proof The proof is performed by induction on the accessibility of the measure `v` of σ_0 for the well-founded relation $\prec$.

5 Related Work and Conclusion

We have taken advantage of the powerful features of the CC type system to give natural operational definitions to the temporal operators. We have shown that the use of co-inductive types to specify infinite executions and temporal operators interpreted as greatest fixpoints leads to elegant reasoning and concise proofs. Thus, we have developed a certified library of LTL lemmas, reusable in practical cases to establish safety or liveness properties.

In this field several other studies have been performed.

In [13], the author presents a system for verifying concurrent systems on the Boyer-Moore theorem prover. It is based on a model of concurrency suggested in [9]. A concurrent program is a finite set of processes represented as directed graphs whose edges are labelled with a pre-condition and a transition value. Program executions are axiomatized using time parameters (natural numbers) and the author uses the implicit universal quantification of variables in the Boyer-Moore logic to encode *always* formulas. The formulation of eventuality properties is complicated by the absence of existential quantifiers and requires the introduction of an additional axiom.

In [6], an encoding of Unity is presented and a proof of consistency and completeness of the logic with respect to operational definitions is given. The executions are represented as functions defined on the natural numbers and the operational temporal operators are classically defined by means of quantifications on time parameters.

In [15] a verified library for temporal rules for Unity is set up but no execution semantics is formalized. Temporal operators are defined in terms of the structure of the transition system, in the tradition of the Floyd-Hoare method, but not relative to particular executions. A drawback of this approach is that it does not make different fairness requirements possible.

Let us also mention the Coq contribution in [8] where co-induction is used to axiomatize CTL, but as far as we know, this work is not documented.

Several formalizations of the μ-calculus have been set up in CC ([11], [14]). The μ-calculus subsumes LTL, CTL, CTL* but it is not comfortable to work with when verifying actual case studies, especially when it is specified as a deep embedding, and either interpreted over labelled transition systems ([14]) or associated with a proof system in Natural Deduction Style ([11]).

As an illustration of our approach and as a demonstration of the utility of our axiomatization, we have performed a rather complex formal certification of an incremental garbage collector ([3]) in Coq that relies of the libraries resulting of this study.

References

[1] Thierry Coquand and Gérard Huet. Constructions : A Higher Order Proof System for Mechanizing Mathematics. *EUROCAL 85, Linz Springer-Verlag LNCS 203*, 1985.

[2] Solange Coupet-Grimal. An Axiomatization of Linear Temporal Logic. *The Coq Users's Contributions*, July 2002. http://pauillac.inria.fr/coq/contribs-eng.htlm.

[3] Solange Coupet-Grimal and Catherine Nouvet. Verification of an Incremental Garbage Collector in Type Theory. *The Journal of Logic and Computation*, 2002.

[4] Dov M. Gabbay, Amir Pnueli, Saharon Shelah, and Jonathan Stavi. On the Temporal Analysis of Fairness. In *7th ACM Symposium on Principles of Programming Languages*, pages 163–173, 1980.

[5] Eduardo Giménez. *Un Calcul de Constructions Infinies et son application a la vérification de systèmes communicants*. Thèse d'université, Ecole Normale Supérieure de Lyon, December 1996.

[6] Barbara Heyd and Pierre Crégut. A Modular Coding of Unity in Coq. In *The 1996 International Conference on Theorem Proving in Higher Order Logics, TPHOL'96*, Turku, Finland, August 1996. Springer-Verlag.

[7] Dexter Kozen. Results on the Propositional μ-Calculus. *Theoritical Computer Science*, 27:333–354, 1983.

[8] Carlos Daniel Luna. Computation Tree Logic for Reactive Systems and Timed Computaion Tree Logic for Real Time Systems. *The Coq Users's Contributions*, March 2000. http://pauillac.inria.fr/coq/contribs-eng.htlm.

[9] Zohar Manna and Amir Pnueli. Verification of Concurrent Programs: the Temporal Framework. In R.S. Boyer and J. Moore, editors, *The Correctness Problem in Computer Science*, Academic Press. London, 1981.

[10] Zohar Manna and Amir Pnueli. *The Temporal Logic of Reactive and Concurrent Systems*. Springer-Verlag, 1991.

[11] Marino Miculan. On the Formalization of the Modal μ-Calculus in the Calculus of Inductive Constructions. *Information and Computation*, 164(1), 2001.

[12] Christine Paulin-Mohring. *Définitions Inductives en Théorie des Types d'Ordre Supérieur*. Habilitation à diriger les recherches, Université Claude Bernard Lyon I, December 1996.

[13] David M. Russinoff. A Verification System for Concurrent Programs Based on the Boyer-Moore Prover. *Formal Aspects of Computing*, 4:597–611, 1992.

[14] Christoph Sprenger. A Verified Model Checker for the Modal μ-Calculus in Coq. In *Tools and Algorithms for Construction and Analysis of Systems, 4th International Conference, TACAS '98*, number 1384 in LNCS, pages 167–183. Springer-Verlag, 1998.

[15] Mark-Oliver Stehr. Embedding UNITY into the Calculus of Constructions. Research Report FBI-HH-B-214/98, University of Hamburg, Department of Computer Science, September 1998.

[16] The Coq Development Team. The Coq Proof Assistant Reference Manual – Version V7.1. Technical report, LogiCal Project-INRIA, 2001.

[17] Moshe Vardi. Branching vs. Linear Time: Final Showdown. In *The European Joint Conferences on Theory and Practice of Software (Etaps'01)*, (Invited Lecture), Genova, 2001.

[18] Pierre Wolper. Using Temporal Logic Can Be More Expressive. *Inform. and Control*, 56(1,2):181–209, 1983.