Modelling Cyclic Structures in Agda

Coinductive formalizations of Linear Temporal Logic

Călin-Marian Diacicov
Supervisor(s): Jesper Cockx, Bohdan Liesnikov
EEMCS, Delft University of Technology, The Netherlands

A Thesis Submitted to EEMCS Faculty Delft University of Technology,
In Partial Fulfilment of the Requirements
For the Bachelor of Computer Science and Engineering
June 22, 2025

Name of the student: Călin-Marian Diacicov
Final project course: CSE3000 Research Project
Thesis committee: Jesper Cockx, Bohdan Liesnikov, Diomidis Spinellis

Abstract

This paper explores the formalization of Linear Temporal Logic (LTL) within the Agda proof assistant, focusing on the use of coinductive techniques to model infinite structures. Two primary questions guide this investigation: how can coinduction be employed to represent LTL formulas, and what are the limitations of Agda in doing so? To address these, we present two ways of encoding LTL in Agda. The suitability of the approaches is evaluated by formalizing logical properties, deriving inference rules, and encoding the Towers of Hanoi as a temporal state system. The results demonstrate that coinductive techniques in Agda are expressive enough for reasoning about temporal logic. The findings provide insights into the strengths and limitations of Agda for modeling temporal logics and suggest directions for future work, including the exploration of sized types and extensions to first-order temporal logic.

1 Introduction

Linear Temporal Logic (LTL), introduced by Pnueli in 1977 [Pnu77], has become a cornerstone of formal methods for software verification [Eme90]. Its ability to express temporal properties of reactive systems, such as liveness ("something good eventually happens") and safety ("nothing bad ever happens"), made it particularly valuable for model checking [Cla97]. As a modal logic operating over linear time models, LTL provides a natural framework for reasoning about state systems [KM08], much as classical logic serves mathematical reasoning.

In general, there is a long documented connection between logic and computation. The Curry-Howard correspondence [How80] has led to the emergence of proof assistants, such as Rocq and Agda. These are tools that have found applications in a number of fields. In formal mathematics, Georges Gonthier [Gon08] used Rocq to provide a formal proof of the four-colour-theorem. In program verification, Xavier Leroy [Ler09] used the same proof assistant Rocq to develop a verified C compiler as part of the CompCert project.

Proof assistants are required to be total languages, in the sense that they require all programs to be terminating. This condition is necessary to ensure the underlying logic is consistent. For example, in a language without totality, arbitrary types can be constructed via a self-referencing recursive function.. To allow the modelling of these structures, Agda provides coinduction, a technique categorically dual to induction. However, coinduction is generally much less developed than its counterpart. Specifically, all the options Agda provides for coinduction support are limited.

To the best of our knowledge, there are not many attempts to formalize LTL formulas and satisfaction using a proof assistant. Jeffrey [Jef12] and Jeltsch [Jel11] provide an encoding of LTL propositions as predicates indexed on time. However, they do not rely on coinduction in their proposed models. Gaducci et. al. [GLT23] provide an implementation of a quantified linear-time temporal logic for analyzing graph transformations. They make use of coinduction in their formalization, but they do not use it in modelling LTL semantics. As such, there is still a gap in the literature in modelling the semantics of LTL using coinduction.

This paper aims to bridge this gap by investigating how coinduction can be used to model Linear Temporal Logic (LTL) within the Agda proof assistant. In particular, it addresses two key subquestions. Firstly, how can coinduction be used to model LTL formulas? Secondly, what limitations of Agda arise in the process of modeling and reasoning about temporal logic? By exploring these questions, the paper aims to assess the suitability and

expressiveness of Agda as a framework for formalizing temporal logic through coinductive techniques.

To answer these questions, this paper provides two concrete implementations of Linear Temporal Logic in Agda. The paper comes with the following contributions:

- A deep embedding of the syntax and semantics of Propositional Linear Temporal Logic in Agda. (Section 3.1)
- A shallow embedding of the semantics of Linear Temporal Logic (Section 3.2)
- Derivations of axioms and rules of Linear Temporal Logic using both encodings (Sections 4.1 and 4.2)
- An encoding of a state system using the shallow embedding. (Section 4.3)
- An analysis of the limitations encountered in Agda (Section 5)

The paper is structured as follows. Section 2 provides the necessary background information in Linear Temporal Logic and coinduction. In section 3, two coinductive formalizations of LTL syntax and semantics are given. Section 4 will derive a proof system for LTL with both of those encodings and will show how to represent the Towers of Hanoi process as a state system to allow reasoning with LTL. Section 5 will provide a discussion on the limitations of Agda, and suggest a number of improvements. Section 6 will consider ethical considerations with regards to the research performed. Finally, Section 7 will give a summary of the paper, together with a number of conclusions.

2 Background

In this section discuss the relevant background introduction to Linear Temporal Logic and coinduction. Section 2.1 presents the syntax and semantics of the Propositional Linear Temporal Logic (PLTL). Section 2.2 provides an introduction to coinduction and how the Agda proof assistant supports it.

2.1 Linear Temporal Logic

Linear Temporal Logic (LTL) [Pnu77] is a modal logic with modalities referring to time. LTL works with a linear time model, which separates it from other temporal logics, such as Computational Tree Logic (CTL) [Eme90], where time can take branches.

In general, the time model for LTL is a total order. A further assumption that is generally made is to consider time to be discrete, in the sense that there exists an isomorphism between the time model and natural numbers. In this paper, we will also make this assumption.

The syntax for LTL contains all the classical operators of propositional logic. In addition, the LTL we consider has three temporal modalities. We give below the definition of an LTL formula.

Definition 1. *Assume a set V of propositional variables. Then the alphabet of LTL formulas is given by the following grammar:*

$$P, Q ::= \textbf{false} \mid \neg P \mid P \vee Q \mid P \wedge Q \mid \bigcirc P \mid \Box P \mid \Diamond P$$

.

Classical expositions of LTL usually give a smaller grammar and exploit the relations between classical operators. We preferred a longer grammar as it better reflects the approaches in our code.

We refer to $\bigcirc P$ as "next" P, to $\Box P$ as "always" P and to $\Diamond P$ as "eventually" P. We further define implication and equivalence with the classical definitions:

$$P \to Q := \neg P \vee Q$$
$$P \leftrightarrow Q := (P \to Q) \wedge (Q \to P)$$

Kripke structures are constructs generally used in modal logics to define the semantics of the logic. Thus, we will present the Kripke (temporal) structure for LTL. A Kripke structure is a defined as a sequence $(\eta_i)_{i \in \mathbb{N}}$, where each term of the sequence is a function $\eta_i : V \to \{\text{true}, \text{false}\}$, giving a true or false valuation for all propositions. Note the two usages of the false keyword. We have used **false** to refer to a constant proposition in the syntax of an LTL formula, while false is a boolean. We will keep this notation consistent to avoid confusion.

With respect to a Kripke structure, any LTL formula P can have a valuation at the time $i \in \mathbb{N}$, which we will denote by $K_i(P)$.

Definition 2. *Let V be a set of propositional variables, K be a Kripke structure, and P be an LTL formula. Then the valuation of the formula $K_i(P)$, for $i \in \mathbb{N}$ with respect to the Kripke structure K is given by*

1. $K_i(v) = \eta_i(v)$ *for* $v \in V$
2. $K_i(\neg P) = \text{true} \Leftrightarrow K_i(P) = \text{false}$
3. $K_i(\textbf{false}) = \text{false}$
4. $K_i(P \vee Q) = \text{true} \Leftrightarrow (K_i(P) = \text{true})$ *or* $(K_i(Q) = \text{true})$
5. $K_i(P \wedge Q) = \text{true} \Leftrightarrow (K_i(P) = \text{true})$ *and* $(K_i(Q) = \text{true})$
6. $K_i(\bigcirc P) = K_{i+1}(P)$
7. $K_i(\Box P) = \text{true} \Leftrightarrow \forall j \geq i, K_i(P) = \text{true}$
8. $K_i(\Diamond P) = \text{true} \Leftrightarrow \exists j \geq i, K_i(P) = \text{true}$

In LTL, there are two notable types of semantics. We say a formula P is satisfied with respect to a Kripke structure K if $K_0(P) = \text{true}$. We note this relation with $K \vdash P$. Alternatively, we say a formula P is valid with respect to a Kripke structure K ($K \vDash P$) if $\forall i \in \mathbb{N}, K_i(P) = \text{true}$. The relation $K \vDash P \Leftrightarrow K \vdash \Box P$, immediately follows.

Note that based on the semantics we have provided, we have the equivalence $\Diamond P \Leftrightarrow \neg \Box \neg P$. This equivalence is not necessarily true in constructive logic. As such, we preferred to introduce the $\Diamond$ modality in the syntax.

We now give an axiomatization of LTL, based on the operators we have provided. These axioms are based on the validity semantics we have explored. We chose this because most expositions of LTL present the axiomatizations as such [KM08].

Axioms

(taut) All tautologically valid formulas

(Fun) $K \vDash \neg\bigcirc A \leftrightarrow \bigcirc\neg A$

(K$\bigcirc$) $K \vDash \bigcirc(A \to B) \to (\bigcirc A \to \bigcirc B)$

(ltl3) $K \vDash \Box A \to A \wedge \bigcirc\Box A$

(MP) $K \vDash A \to (A \to B) \to\ B$

(N$\Box$) $K \vDash A \to\ \Box A$

(Ind) $K \vDash (A \to \bigcirc A) \Rightarrow K \vDash (A \to \Box A)$

2.2 Coinduction

Inductive reasoning is a technique that is suitable for finite structures. This is because inductive types must be *well-founded*, meaning that each term should always be composed of a finite number of constructors. However, induction is not appropriate when dealing with infinite structures, which are not well-founded. For these cases, type theory can be extended with coinductive types. As opposed to induction, a coinductive type is not defined by its constructors. Rather, one defines a coinductive type via the destructors of the type. Intuitively, a coinductive type is defined by specifying all the possible observations that can be made on it. Definitions using coinductive types need to ensure productiveness, in the sense that every finite part of the output takes a finite number of steps [Coq93]. Agda provides three main methods to use coinduction: guarded coinduction, musical conduction, and sized types. Each of them comes with a different approach to ensuring productiveness.

Musical coinduction

Musical coinduction is the "old" method for defining coinductive types [Tea25]. In musical notation, one manually forces and delays computations in the definition of coinductive types. To declare a coinductive type, the $\infty\_$ operator is used. This operator comes with two functions, a delay function $\sharp\_ : A \to \infty A$ and a force function $\flat : \infty A \to A$. Agda ensures productivity by forcing every recursive call in a corecursive definition to be directly under a constructor [VvdW19]. This is a rather strict requirement, which has lead to musical coinduction currently being deprecated, in favor of guarded coinduction.

As an example, we will consider the coinductive type of streams. Using musical notation, we can encode this type as a datatype with one constructor cons : $A \to \infty\text{Stream}\, A \to \text{Stream}\, A$. The delay operator used for the second parameter suggests that the type is coinductive and the second parameter should not be computed unless explicitly forced. Figure 1 shows the definition of a map : $(A \to B) \to \text{Stream}\, A \to \text{Stream}\, B$ function, which computes a new stream by applying the function given as the first parameter to all the elements. This definition is accepted by the productivity checker because the recursive call is placed directly under the constructor cons.

```
map : {A B : Set} → (A → B) → Stream A → Stream B
map f (x :: xs) = f x :: ♯ map f (♭ xs)
```

Figure 1: Definition of a map function for a Stream in musical notation

Guarded coinduction

Guarded coinduction is currently the standard way of defining coinductive types in Agda and relies on the use of coinductive records and copattern matching. Abel [APTS13] presented copattern matching as a method to encode and construct infinite objects through the observations that can be made on them, rather than by their constructors. Abel also shows that definitions via copatterns lead to strongly normalizing rewriting rules and a semantics without infinitary rewriting [APTS13].

Productivity in this instance becomes an instance of termination [AP13]. This leads to a unified approach to recursion and corecursion and to more flexibility in defining corecursive functions than the musical approach.

Encoding streams using coinductive records is straightforward. An object of type Stream A one can make two observations: the head of the stream, a term of type A, and the tail of the stream, which is a term of type Stream A. Figure 2 shows implementations for a stream and the definition of the map function.

```
record Stream (A : Set) : Set where
  coinductive
  field
    head : A
    tail : Stream A


map : {A B : Set} → (A → B) → Stream A → Stream B
map f s .head = f (head s)
map f s .tail = map f (tail s)
```

Figure 2: Encoding of the stream type and the definition of the map function using coinductive records

Sized types

A sized type is a type annotated with a size parameter which specifies the number of constructors the type can unfold from itself. In Agda, one can use sized types in combination with coinductive records to encode coinductive types [APTS13] [AVW17]. In this instance, Agda ensures productiveness syntactically by enforcing the size parameter to decrease during a constructor. Currently, the implementation of sized types in Agda is known to be inconsistent, as presented by this GitHub issue.

3 Encodings

In this section, we will present two methods in which Linear Temporal Logic, as exposed in the previous section, can be implemented in the Agda proof assistant. The first encoding is a *deep embedding*, in the sense that we explicitly define the syntax as an inductive type, and then model semantics based on the syntax. The second encoding is a *shallow embedding*, meaning that we leverage the constructs Agda already provides to define the semantics of LTL.

3.1 Deep embedding of the semantics

To provide a deep embedding of LTL, we need to encode both the syntax and the semantics into the proof system. We will first discuss the syntax, and then proceed to the semantics.

Syntax

The syntax of LTL has been presented in Section 2 and the inductive definition can be represented in a similar fashion in Agda (Figure 3). However, there are two implementation details that are relevant for the encoding, which we will now discuss.

Firstly, we assume that the set of propositional constants is finite. As such, we identify this set with the type $\mathrm{Fin}\, n$, for a given $n : \mathbb{N}$. We could have also identified the set of propositional constants with the type $\mathbb{N}$ and thus have the possibility of infinite propositional variables. For the purposes of this paper, the distinction has proven irrelevant.

Secondly, the treatment of the $\neg$ operators requires special attention. To ease the definition of the semantics, the inductive definition of an LTL formula only allows negating propositional constants. We then define arbitrary negation separately using De Morgan's laws. This approach imposes a classical framework from the beginning. Since Agda is intuitionistic by default, this requires us to assume the Law of Excluded Middle as a postulate.

```
data LTLFormula : Set where
  ⊥ : LTLFormula
  ⊤ : LTLFormula
  Var : Fin n → LTLFormula
  Neg : Fin n → LTLFormula
  _∨_ : LTLFormula → LTLFormula → LTLFormula
  _∧_ : LTLFormula → LTLFormula → LTLFormula
  ◯_ : LTLFormula → LTLFormula
  □_ : LTLFormula → LTLFormula
  ◇_ : LTLFormula → LTLFormula
```

Figure 3: Inductive definition of an LTL Formula

Kripke structures

Temporal (Kripke) structures are necessary to define the semantics. Classically, they are encoded as sequences of states, where each state provides a valuation for every propositional constant. We translate this idea by encoding a Kripke structure as an infinite stream of states. Each state is a function with domain $\mathrm{Fin}\, n$ and codomain Bool. As such, at each point in time, all the propositional constants will be either true or false.

Semantics

We encode satisfaction of an LTL formula, given a Kripke structure, by mostly translating the traditional formulation, using the Curry-Howard correspondence. The cases where we deviate are negation and the $\Box$ and $\Diamond$ modalities.

```
_⊢_ : (KripkeStructure n) → LTLFormula {n} → Set
K ⊢ (Var x₁) = (head K) x₁ ≡ true
K ⊢ (Neg x₁) = (head K) x₁ ≡ false
```

Figure 4: Inductive definition of the satisfaction relation. Only two cases are presented, for propositional variables and negation.

```
record _⊨_ (K : KripkeStructure n) (P : LTLFormula) : Set where
  coinductive
  field
    TrueNow : K ⊢ P
    TrueInFuture : (tail K) ⊨ P
```

Figure 5: Valid semantics of an LTL formula with respect to a Kripke structure

We have primitively defined negation as a constructor that takes as a parameter only a propositional constant, rather than defining it more generally as a constructor that can take any propositional formula. If we had opted for the latter option, then we would have encountered problems in defining the semantics. This is because inductive definitions are required to be positive, meaning that no recursive call can be present on the left of an arrow.

With our definition, we can easily preserve the positivity. We achieve this by considering the negation of a propositional constant to be satisfied if at the current time, it is evaluated to be false (Figure 3).

The semantics of the $\Box$ modality are defined to reflect that the proposition is true at all times. Since we defined the temporal structure as an infinite stream, the simplest option to encode it is via a coinductive record $\_ \vDash \_$ with two fields. One of the fields is of type $K \vdash P$, to represent satisfaction at the current time, while the second field is of type $(\text{tail}\, K) \vDash P$ to recursively iterate through all the states (Figure 5).

3.2 Shallow embedding of the semantics

This embedding is adapted from the work by Jeffrey [Jef12]. Where we diverge from the model provided in his paper is in the way we encode LTL propositions. Jeffrey models an LTL proposition as a function Time → Set, where Time is a type embedded with the required operations. On the other hand, the discrete time model we assumed allows us to encode an LTL proposition as an infinite stream of sets (Figure 6).

```
LTLProp : Set₁
LTLProp = Stream Set
```

Figure 6: Definition of LTLProp as a stream of sets.

Logical operators

In the "propositions as types" paradigm, types represent propositions, with terms of those types representing proofs of the said proposition. Similarly, if we encode LTL propositions as streams of sets, a proof that the proposition is valid at all times should be represented by an "infinite stream of inhabitants". Such a structure can be defined as a coinductive record (Figure 7). With this definition, a proposition P is valid if the type $\vDash P$ is satisfied.

```
record ⊨_ (P : LTLProp) : Set where
  coinductive
  field
    ⊨-head : head P
    ⊨-tail : ⊨ (tail P)
```

Figure 7: Validity of a type

This encoding allows us to provide compact implementations of almost all of the operators used in LTL. The standard logical operators ($\vee$, $\wedge$, $\neg$, etc.) can be defined using pointwise liftings. Figure 8 provides an example, while Table 1 shows all the operators and their corresponding types.

```
_∧_ : LTLProp → LTLProp → LTLProp
(p ∧ q) .head = (head p) × (head q)
(p ∧ q) .tail = (tail p) ∧ (tail q)
```

Figure 8: Example of defining operators for the conjunction of two LTL propositions.

Operator	**Type**
$P \vee Q$	$\text{head}\, P \uplus \text{head}\, Q$
$P \wedge Q$	$\text{head}\, P \times \text{head}\, Q$
$P \to Q$	$\text{head}\, P \to \text{head}\, Q$
$\neg P$	$\text{head}\, P \to \bot$
$\Box P$	$\vDash P$
$\Diamond P$	$\sum_{(n:\mathbb{N})} \text{Eventually}\, P\, n$

Table 1: Summary of logical operators and their associated types

Temporal modalities

To define the $\bigcirc$ modality, we opted not to define it using this standard approach. Rather, one can observe that defining $\bigcirc P := \text{tail}\, P$ is an equivalent definition which has the advantage that it is easier to work with in practice.

The $\Box$ modality represents that a proposition is true at all times from the current point forward. This, in general, leads to infinite computations. Consequently, inductive types are not appropriate, and coinductive definitions are required. The $\vDash \_$ definition we described earlier encodes exactly what the $\Box$ modality represents. Thus, we define head $(\Box P) := (\vDash P)$. This definition is also motivated by the fact that a proposition P is valid if and only if $\Box P$ is satisfied.

Defining the $\Diamond$ modality requires some discussion. The $\Diamond$ modality is used to express that a statement will eventually hold. In a classical setting, this can be defined as $\exists i.\, K_i(P) = \mathit{true}$. However, classical formulations define it with the equivalent $\Diamond P := \neg \Box \neg P$. In a constructive framework, those two definitions are not equivalent. The first definition means the ability to construct the moment in time when the proposition will be true, while the second formulation merely means that the proposition is not always false. Thus, to preserve the full strength of the $\Diamond$ modality, we cannot use the standard definition.

As a consequence, we will define the $\Diamond$ modality using an inductive data type.

Given $n : \mathbb{N}$, `Eventually P n` encodes a term that will be true n steps into the future. Our definition of $\Diamond P$ will be encoded as a dependent pair taking a natural number n and an inhabitant of `Eventually P n`. The decision to embed the number of steps into the data type has been taken is done to ensure that the termination checker recognizes the finite nature of the data type.

4 Experiments

To evaluate the effectiveness of the encodings, we have attempted to prove properties that encompass several desirable features of an LTL encoding. Concretely, we have done the following:

- Verify the soundness of the encodings to ensure the encodings properly characterise the propositional LTL we considered.
- Proven a number of absorption theorems to assess whether the $\Box$ and $\Diamond$ modalities interoperate as desired.
- Encoded Towers of Hanoi as a state system in LTL and used the encoding to prove features of the system. This was performed to assess the usability of the LTL encodings.

Table 2 provides a quick summary of the results of the experiments. In the following subsections we will proceed with a discussion for each

Encoding	Soundness	Absorption Theorems	Towers of Hanoi
Shallow embedding	✓	✓	✓
Deep embedding	✓	✓	✗

Table 2: Summary of the suitability of each encoding for each experiment type

4.1 Soundness

To assess the correctness of our encodings, the first experiments consisted of deriving an axiomatization of LTL (Section 2).

```
induction-helper : {P : LTLProp} → ⊨ (P ⇒ (◯ P)) → ⊢ P → ⊨ P
induction-helper valFP p .⊨-head = p
induction-helper valFP p .⊨-tail = induction-helper (⊨-tail valFP) ((⊨-head valFP) p)
```

Figure 9: Proof of the helper lemma for induction, which constructs the first destructor for the Induction lemma

```
Ind : {P : LTLProp} → ⊨ (P ⇒ (◯ P)) → ⊨ (P ⇒ (□ P))
Ind valFP .⊨-head p = induction-helper valFP p
Ind valFP .⊨-tail = Ind (⊨-tail valFP)
```

Figure 10: Proof of the full induction principle

The results of this experiment have been positive. We have successfully proven most of the axioms and rules. We have not managed to prove axiom A_3 in the case of the shallow embedding. However, this is strictly because we deal with an intuitionistic framework. In classical logic, the axiom is equivalent to the Law of Excluded Middle. Therefore, the inability to prove axiom A_3 was expected, and if the Law of Excluded Middle is assumed, the axiom becomes provable. Other than axiom A_3, the proofs for the shallow embedding have been straightforward and compact. For all identities proven, the most important issue lies in constructing the destructor $\vDash$ −head. The second constructor can be trivially constructed by using a corecursive call. To showcase the general approach, consider the axiom Ind, which states that the formula $\Box(P \Rightarrow \bigcirc P) \Rightarrow P \Rightarrow \Box P$ is always valid in LTL. To derive this identity, we first construct the first destructor, which can be done using the lemma in Figure 9. Afterwards, the proof of the full induction principle can be seen in Figure 10.

In the case of the deep embedding, we have opted for a classical definition of implication. Two lemmas have proven to be useful when proving identities in LTL. The first one states that having a proposition and its negation satisfied at the same time yields a contradiction (Figure 11 for the type definition). The proof for this proposition is by induction on the derivation of P. The second one asserts that if a proposition Q is satisfied, given that P is satisfied, then the proposition $P \Rightarrow Q$ is satisfied. This proposition is a consequence of our assumption of LEM, and the proof is done by case analysis on whether P or $\neg P$ holds (the Law of Excluded Middle assumption).

```
¬-contra : {K} {P} → (K ⊢ P) → (K ⊢ (¬ P)) → ∅
→-⇒-help : {K} {P Q} → (K ⊢ P → K ⊢ Q) → K ⊢ (P ⇒ Q)
```

Figure 11: Type definitions for the two lemmas discussed. The type ∅ represents the empty type. We have adopted this notation to avoid name conflicts. Type annotations for implicit parameters were omitted for clarity.

4.2 Absorption properties

To explore the interplay between the $\Box$ and the $\Diamond$ modalities, we have proven a number of absorption rules for both of the encodings. More concretely, we have proven the following properties:

1. $\Box\,\Box\,P \Leftrightarrow \Box P$
2. $\Diamond\Diamond P \Leftrightarrow \Diamond P$
3. $\Box\Diamond\,\Box\,P \Leftrightarrow \Diamond\,\Box\,P$
4. $\Diamond\,\Box\,\Diamond P \Leftrightarrow \Box\Diamond P$

The results of this experiment were positive for both encodings. The proofs follow the standard procedure we presented in subsection 4.1. To prove the second implication of property 3, we have first constructed a helper lemma, showing that if $\Box P$ is valid after n steps, then $\bigcirc\,\Box\,P$ is valid after n steps. This lemma is possible because we can construct terms of type Eventually $P\,n$ for multiple values of n. If we instead only allowed Eventually $P\,n$ to be constructed for the smallest value of n, then the lemma is not provable.

4.3 Towers of Hanoi

Towers of Hanoi is a classical example of a state system. A state consists of three poles, where each of the two poles has a number of differently sized disks placed in increasing order from the top to the bottom. The valid transitions from one state to another can be done by moving a disk that sits on top of the pole to another disk, as long as the disks remain in increasing sizes.

We managed to encode the Towers of Hanoi using the shallow encoding. We encoded the poles as Lists of terms from the type Fin 3, the type which contains exactly 3 terms. Then the game state was defined as a record containing three poles. Finally, the Hanoi process was defined as a stream of game states. Figure 12 shows the definition for the three terms.

The definition we have given so far do not fully characterize all of the constraints of the Hanoi process. More concretely, we have not yet enforced any constraints on the valid transitions from one state to another, nor the fact that the disks need to be in increasing order on the poles. We encoded these constraints as functions HanoiProcess → LTLProp. For the valid transitions, we defined an inductive type Move specifying the valid moves (Figure 12). Then the axiom becomes $\text{HanoiAxiom}_1\,h := \text{Move}\,(\text{head}\,h)\,(\text{head}\,(\text{tail}\,h))$. We followed a similar approach for the second constraint. We first defined an inductive type IsIncreasing encoding when a pole is increasing (Figure 13), so that the second axiom is a conjunction for all three poles (Figure 14).

With these encodings in place, we have shown two properties of a Hanoi process. Firstly, we have proven that the number of disks remains constant throughout the Hanoi process. Secondly, we have shown that if initially two of the poles are full and the third one is empty, then, after one move, there will be a disk on the third pole.

For the deep embedding, we have not attempted to encode the Towers of Hanoi as a state system, due to time constraints. We believe an encoding should be possible. However, the lack of quantifiers in propositional linear temporal logic would make the encoding a laborious process. Therefore, we believe that this encoding is not suitable to encode the Towers of Hanoi as a state system.

```
Pole : Set
Pole = List (Fin 3)
```

(a) Definition of Pole

```
record GameState : Set where
  constructor state
  field
    a b c : Pole
```

(b) Game State record

Figure 12: Definitions for poles and a game state

```
data Increasing : List (Fin 3) → Set where
[]-Increasing : Increasing []
[x]-Increasing : ∀ {x} → Increasing (x :: [])
::-Increasing : ∀ {x y xs} → x < y → Increasing (y :: xs) → Increasing (x :: y :: xs)

AIncreasing : HanoiProcess → LTLProp
AIncreasing h .head = (Increasing ((head h) .a))
AIncreasing h .tail = AIncreasing (tail h)
```

Figure 13: Inductive data type which is inhabited whenever a pole is strictly increasing (`Increasing`) and a LTLProp stating that the first pole is in increasing order. (`AIncreasing`)

5 Limitations of Agda

One of the most important limitations Agda currently faces with regards to using coinduction is its usability barrier. This barrier exhibits itself in numerous circumstances.

Documentation

Documentation in some instances is not completely clear. While the documentation on musical coinduction and guarded coinduction explain in sufficient amount of detail the concepts and how to use them in Agda, the documentation on sized types is severely lacking. The section only provides an example, failing to cover and explain all the primitives of sized types. The documentation regarding sized types refers to another section, where the primitives are presented. However, it is not clear what these primitives represent and how they should be used when the standard library is included. A potential improvement to the documentation could be the improvement of the section on Sized types by providing examples which cover all of the features of Sized types, while also providing some sort of explanation of how sized types work and how to reason with them.

Error messages

Agda supports interactive programming [Nor09], where definitions are incrementally constructed by starting with a placeholder "hole" and iteratively refining it until the complete program is derived. During this process, it is common for errors to appear. However, the

```
HanoiAxiom₁ : HanoiProcess → LTLProp
HanoiAxiom₁ h = AIncreasing h ∧ BIncreasing h ∧ CIncreasing h

HanoiAxiom₂ : HanoiProcess → LTLProp
HanoiAxiom₂ h .head = Move (head h) (head (tail h))
HanoiAxiom₂ h .tail = HanoiAxiom₂ (tail h)
```

Figure 14: Definition of the two Hanoi axioms.

error messages are usually very hard to understand. For example, when dealing with equational reasoning, the error messages provided are at most unhelpful in indicating what part of the equality is incorrect. As a further example, when termination checking fails, Agda highlights any potential part of the code that might non-terminating. When there are few cases this is not problematic. However, it becomes highly problematic when the number of cases in a function increases.

Termination checking

Termination of programs has always been a problem of interest in computer science since the very beginning. For proof assistants such as Agda, termination of all programs is essential to ensure that the underlying logic of the proof assistant remains consistent. A major issue arises from the undecidability of the halting program, which implies that there is no algorithm that can decide whether a program will terminate or not. As such, total languages must restrict the number of programs that can be expressed in their languages, to ensure that a decidable termination checking algorithm can be used to enforce totality.

Agda comes equipped with a powerful termination checker, being able to determine termination of a large number of programs with non-trivial recursions. However, there are instances where the termination checker is still failing, where it probably should not.

Consider, for example, the code in Figure 15. In both cases, the recursion is structurally terminating based on the parameter $m : \mathbb{N}$, which is strictly decreasing. However, the termination checker does not recognize this program to be terminating. This problem can be worked around by using helper functions to treat these two special cases. However, it is interesting to note that this problem only arises in the case where the definition of both cases is done explicitly in the function. If either the case for $\Box P$ or $\Diamond P$ is replace by a call to a helper function, then the termination checker correctly determines that the function terminates.

6 Responsible Research

In this section, we will discuss ethical considerations of the performed research. Firstly, we will discuss considerations regarding the validity of the results. Finally, we will talk about the reproducibility of results.

```
¬-contra : {K : KripkeStructure n} {P : LTLFormula} → (K ⊢ P) → (K ⊢ (¬ P)) → ∅
¬-contra {K} {□ P} kvP (0 , now x) = ¬-contra {K} {P} (TrueNow kvP) x
¬-contra {K} {□ P} kvP (suc m , later kNP) =
          ¬-contra {tail K} {□ P} (TrueInFuture kvP) (m , kNP)
¬-contra {K} {◇ P} (0 , now x) (kvNp) = ¬-contra {K} {P} x (TrueNow kvNp)
¬-contra {K} {◇ P} (suc m , later kNP) kvNp =
          ¬-contra {tail K} {◇ P} (m , kNP) (TrueInFuture kvNp)
```

Figure 15: Example of a terminating function that is not detected as terminating by the termination checker. The cases that are not relevant to the issue at hand have been omitted. The calls highlighted in red are calls Agda considers non-terminating.

6.1 Validity of results

Proof assistants are software tools with the purpose of aiding in developing formal proofs. Consequently, it is necessary to ensure that the results obtained with the help of a proof assistant are valid. This is done by ensuring the logical foundations of the proof assistant are consistent. For example, in the Agda proof assistant, it should be impossible to construct terms of the empty type.

However, the Agda proof assistant is a proof assistant in active development. As a consequence, there always exists the possibility that the underlying type checker to have errors which can lead to inconsistencies.

First of all, Agda provides a multitude of features, not all of them being fully compatible between themselves. For example, the current implementation of sized types in Agda is known to provide inconsistencies, as exemplified by the following GitHub issue[1]. As a further example, Agda allows to formulate postulates, which assumes a term of an arbitrary type. As such, careful consideration must be taken to ensure that the features used in the implementation do not lead to inconsistencies.

To mitigate this issue, Agda provides the `--safe` flag, which can be given to the type-checker. This flag disables several features which are known to lead to inconsistencies in a way or another [Tea25].

In this paper, most of the code provided has been done with the `--safe` flag enabled. For the first encoding, the Law of Excluded Middle was considered as a postulate. However, we have chosen a subset of the language that is known to be consistent with the Law of Excluded Middle. As such, careful consideration has been taken to ensure the results are valid and not the result of inconsistencies in the proof assistant.

6.2 Reproducibility of results

The codebase has been verified to compile with Agda version 2.7.0.1[2] and Agda Standard Library version 2.2[3]. The complete Agda formalization accompanying this paper, including all encodings and proofs, is publicly available in the following GitLab repository.

[1] `https://github.com/agda/agda/issues/3026`
[2] `https://github.com/agda/agda/releases/tag/v2.7.0.1`
[3] `https://github.com/agda/agda-stdlib/releases/tag/v2.2`

https://gitlab.ewi.tudelft.nl/bohdan/bscp-25-calin#

6.3 Usage of Generative AI

Generative AI tools, mainly in the form of Large Language Models (LLMs) are tools that can be used to a great extent in scholarly work. They can be helpful in many aspects, from automation of trivial but labor-intensive tasks, to code generation. However, the nature of LLMs as black-box models raises important ethical concerns that cannot be overlooked.

LLMs are good at providing answers that may seem correct at first glance. Nevertheless, they are prone to making errors in judgment, called *hallucinations*. Therefore, one must take extreme care whenever using input generated by LLMs. We have used generative AI in this paper specifically to mitigate this issue. We limited the LLM usage in critical areas. Generative AI was not used in any generation of code, its usage being limited to writing the research paper. More concretely, we used GenAI to correct spelling errors, help with formatting LaTeX, get feedback on the general paper, and rephrase sentences and paragraphs. The suggestions were carefully analyzed to ensure no mistakes were introduced.

7 Conclusion

This paper explored the formalization of Linear Temporal Logic (LTL) in the Agda proof assistant using coinductive techniques. Two encodings were developed: a deep embedding that models both the syntax and semantics of LTL, and a shallow embedding that treats LTL propositions as infinite streams of sets. The research addressed key questions about the representation of LTL formulas in Agda, the properties provable with coinductive models, and the limitations of Agda in this context. The encodings were evaluated by proving standard LTL tautologies, deriving inference rules, and modeling the Towers of Hanoi as a temporal state system.

The results demonstrated that coinductive techniques in Agda are expressive enough to model and reason about LTL, as evidenced by the successful proofs of soundness and absorption properties. However, the experiments also highlighted significant limitations of Agda, particularly in usability and documentation, such as unclear guidelines for sized types and occasional failures of the termination checker to recognize structurally terminating programs. These challenges underscore the need for further improvements in Agda's support for coinduction.

The paper has several limitations. First, it focuses solely on propositional temporal logic, leaving first-order extensions unexplored. Second, the analysis was restricted to guarded coinduction, omitting other coinductive techniques like sized types. Third, the work did not investigate binary temporal modalities, which are essential for more complex temporal reasoning. These limitations suggest directions for future research to broaden the scope and applicability of the findings.

Future work could extend this research in several ways. One direction is to explore extensions of PLTL, either by adding quantifiers ($\forall$, $\exists$), or with more temporal modalities. Another avenue is to investigate alternative coinductive techniques, such as sized types, to address the limitations of guarded coinduction. These extensions could further enhance the utility of Agda for formal verification tasks.

In closing, this paper contributes to the understanding of coinductive techniques in Agda for formalizing temporal logic, offering practical insights and highlighting areas for improve-

ment. While challenges remain, the findings pave the way for future advancements in the use of proof assistants for temporal reasoning and verification.

References

[AP13] Andreas M Abel and Brigitte Pientka. Wellfounded recursion with copatterns: A unified approach to termination and productivity. *ACM SIGPLAN Notices*, 48(9):185–196, 2013. Publisher: ACM New York, NY, USA.

[APTS13] Andreas Abel, Brigitte Pientka, David Thibodeau, and Anton Setzer. Copatterns: programming infinite structures by observations. *ACM SIGPLAN Notices*, 48(1):27–38, 2013. Publisher: ACM New York, NY, USA.

[AVW17] Andreas Abel, Andrea Vezzosi, and Theo Winterhalter. Normalization by evaluation for sized dependent types. *Proceedings of the ACM on Programming Languages*, 1(ICFP):1–30, 2017. Publisher: ACM New York, NY, USA.

[Cla97] Edmund M. Clarke. Model checking. In S. Ramesh and G. Sivakumar, editors, *Foundations of Software Technology and Theoretical Computer Science*, pages 54–56, Berlin, Heidelberg, 1997. Springer.

[Coq93] Thierry Coquand. Infinite objects in type theory. pages 62–78. Springer, 1993.

[Eme90] E. Allen Emerson. CHAPTER 16 - Temporal and Modal Logic. In JAN Van leeuwen, editor, *Formal Models and Semantics*, Handbook of Theoretical Computer Science, pages 995–1072. Elsevier, Amsterdam, January 1990.

[GLT23] Fabio Gadducci, Andrea Laretto, and Davide Trotta. Specification and Verification of a Linear-Time Temporal Logic for Graph Transformation. In Maribel Fernández and Christopher M. Poskitt, editors, *Graph Transformation*, pages 22–42, Cham, 2023. Springer Nature Switzerland.

[Gon08] Georges Gonthier. Formal proof–the four-color theorem. *Notices of the AMS*, 55(11):1382–1393, 2008.

[How80] William A Howard. The formulae-as-types notion of construction. *To HB Curry: essays on combinatory logic, lambda calculus and formalism*, 44:479–490, 1980.

[Jef12] Alan Jeffrey. LTL types FRP: linear-time temporal logic propositions as types, proofs as functional reactive programs. In *Proceedings of the sixth workshop on Programming languages meets program verification*, PLPV '12, pages 49–60, New York, NY, USA, January 2012. Association for Computing Machinery.

[Jel11] Wolfgang Jeltsch. Programming in linear temporal logic. 2011.

[KM08] Fred Kröger and Stephan Merz. *Temporal Logic and State Systems*. Texts in Theoretical Computer Science. Springer, Berlin, Heidelberg, 2008.

[Ler09] Xavier Leroy. A formally verified compiler back-end. *Journal of Automated Reasoning*, 43:363–446, 2009. Publisher: Springer.

[Nor09] Ulf Norell. Dependently typed programming in Agda. pages 1–2, 2009.

[Pnu77] Amir Pnueli. The temporal logic of programs. pages 46–57. ieee, 1977.

[Tea25] The Agda Team. Agda User Manual, April 2025.

[VvdW19] Niccolò Veltri and Niels van der Weide. Guarded recursion in Agda via sized types. pages 32–1. Schloss Dagstuhl–Leibniz-Zentrum für Informatik, 2019.