Stuck in a (While) Loop

Assessing Coinduction in Agda using Cyclic Program Traces

Claire Stokka[1]
Supervisor(s): Jesper Cockx[1]**, Bohdan Liesnikov**[1]
[1]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
22nd June 2025

Name of the student: Claire Stokka
Final project course: CSE3000 Research Project
Thesis committee: Jesper Cockx, Bohdan Liesnikov, Diomidis Spinellis

Abstract

Interactive proof assistants such as Agda have powerful applications in proving the correctness of software. Non-terminating programs, such as those containing infinite loops, result in execution paths of infinite length, which can introduce challenges when reasoning about such programs. Agda, as a total language, relies on the concept of coinduction for reasoning about potentially infinite structures. Mutiple methods for coinduction exist in Agda, each with difficulties related to usage or soundness. To evaluate these limitations, I implement traces and semantics for a simple imperative programming language, While, using Agda's various methods of coinduction. The different encodings are compared in their abilities and limitations, and from this I identify areas for improvement in Agda's coinduction support.

1 Introduction

Infinite loops occur frequently when programming. Oftentimes, they are seen as a nuisance — a bug to be removed. However, in many critical applications, programs are required to be non-terminating. These applications include operating systems and industrial control, such as nuclear control systems [San11]. Clearly, the correctness of programs serving such important purposes is critical, and the ability to verify this correctness is a desirable aim.

Interactive proof assistants serve as a powerful tool in the field of computer science, allowing for the formal verification of programs as well as verification of mathematical proofs, with existing use in industry for critical applications such as aerospace control and chip design [Geu09]. However, there exists a gap caused by many interactive proof assistants, such as *Agda*, being *total* — meaning that any computation must provably terminate [Tur04]. Total languages require special consideration for reasoning about potentially infinite or cyclic structures, such as the *traces* arising from non-terminating program execution, in order to satisfy totality. These cyclic structures may be modeled and reasoned about through the concept of *coinduction*, the dual of mathematical induction [San11]. Coinduction, compared to mathematical induction, is a relatively recent area of study with the concept becoming relevant in the field of computer science as late as 1991 [Geu09]. Agda has support for coinductive reasoning, yet this support is not as well developed or understood as the support for induction [CA18]. This limitation makes it more difficult to verify properties of non-terminating programs.

My research aims to explore support for coinduction in Agda through implementing coinductive trace semantics for a simple imperative programming language. A *program trace* describes the progression of states a program passes through during its evaluation. Cycles can easily arise in such a trace, most clearly in cases of infinite loops. This makes traces a strong candidate for modeling coinductively, something which has already been described in literature. Nakata and Uustalu have formalized a coinductive trace semantics for While in Coq, from which much of my research is based [NU09]. Leroy and Grall implement a coinductive semantics for a call-by-value functional language, also in Coq [LG09]. Anacona et al. describe the use of *corules* for generalized inference systems, ultimately implementing a trace semantics for a Java-like language with I/O functionality [ADZ18]. Earlier research by Schmidt used coinduction for modeling big-step semantics of trace-based abstract interpretation [Sch98]. The commonality between such existing work is that it models these traces on paper exclusively or formalizes them in a different proof assistant such as Coq/Rocq. As such, the relevant gap in current research concerns how, and to what extent, these existing models may be adapted for use in Agda. The central question answered by my research con-

cerns evaluation of the different ways (potentially infinite) program traces may be modeled in Agda. I compare the approaches in their capabilities and limitations in terms of the ability to reason about properties arising from traces under each approach. Additionally, my research identifies areas where Agda may be improved in order to increase the ease with which coinductive reasoning can be done.

The main contribution of my research is in the findings and challenges encountered through implementing coinductive traces in Agda. Traces and their semantics were implemented for each of Agda's three methods of coinduction, and experiments using these encodings aimed to further identify gaps in the usability of their respective coinduction styles. I demonstrate problems associated with these styles — for example, I demonstrate a proof which I believe cannot be implemented using Agda's guarded coinduction[1] and my interpretation of the limitations causing this problem.[2] Further research should use this gained experience in order to improve support for the modeling of coinductive structures in Agda, with the eventual goal of expanding the practical capabilities and flexibility of Agda as an interactive proof assistant.

An overview of the relevant background and methodology is provided in section 3. Section 4 describes the trace representations implemented during the research. These implementations are then compared in section 5 through analysis of their capabilities and limitations. The research concludes with a discussion of potential improvements to the coinduction support of Agda based on the insights gained during the experimentation.

2 Responsible Research

While my research does not utilize data or produce an application intended for wider use, there are still ethical considerations which must be taken into account:

2.1 Reproducibility

Reproducibility of results is essential for ensuring that invalid results are not accepted into the scientific community. The current state of reproducibility has been called a crisis, with the central cause, according to the Yale Law School Roundtable on Data and Code Sharing, being the sheer amount of data or code required to produce research in computer science, as contrasted against research where the information required for reproduction can be contained in the article body [The10]. The Roundtable has further established a set of recommendations to address this, which I have endeavored to fulfil in my own research, to the extent that each recommendation is relevant. I summarize the relevant considerations below.

- **Code Availability**: *Recommendation 1* stipulates that code should be made publicly available. My complete work is available in a Git repository hosted at `https://github.com/clairradiant/cse3000-agda-traces`.
- **Licensing**: As directed by *recommendation 4*, my code is licensed under an open source license, namely MIT, to enable code reuse.
- **Versioning**: In order to avoid problems arising from tools and libraries my research depends on changing over time, I have declared the versions of relevant tools (Agda,

[1] See section 5.3.

[2] Discussed in sections 6.3 and 6.5.1, and appendix A.

agda-stdlib) in the README of the repository and created a Dockerfile defining an environment containing the correct versions of these tools. This ensures that my code can be used in the future and mirrors the instructions of *recommendation 3*.

2.2 Use of Generative AI

Generative AI in the form of Large Language Models can serve as a valuable tool in the production of research. However, care must be taken when using these tools. AI models such as ChatGPT are prone to numerous types of errors, termed *hallucinations*, including errors in logic, reasoning, and reproduction of facts [Sun+24]. For this reason, my use of generative AI in the project was limited.

My primary use of generative AI in the project was for obtaining an abstract understanding of concepts in Agda as well as Coq. If there was a concept or proof I encountered in my work that I did not understand, this is an instance where I may decide to apply AI to ask it to explain the concept, if it was something that would be difficult to search via traditional means. I also used generative AI to help think of ideas in creative applications, such as brainstorming ideas for a visualization to represent an infinite while loop, and for explaining abstract LaTeX concepts. To mitigate the risks associated with AI hallucination, I did not ask the AI to generate code for use in my repository or text for use in this report. No AI-generated or AI-edited code or text appears in the output of my research, nor did AI provide results analysis which was used to guide my conclusions. For the purpose of transparency, a listing of prompts given to AI may be found in appendix C.

3 Background

3.1 Program Traces

Program traces describe the progression of states a program passes through during its evaluation. A trace can be described as a non-empty *colist* (list of potentially infinite length) of states, entirely encapsulating the progression of variables during a program's execution. Each state is a complete view of the value each variable takes at each step of the computation. In the sample program illustrated by figure 1, a trace from a starting state $x := 0$ would take the form

$$x = 0 \;\Rightarrow\; x = 1 \;\Rightarrow\; x = 2 \;\Rightarrow\; x = 3 \;\Rightarrow\; ...$$

under the assumption that the variable with identifier 0 is referred to by x. Traces may be more complex, involving multiple variables, and they may be terminating or non-terminating. Often, for terminating programs, the final state is of particular interest, representing the result of a computation. Traces of non-terminating programs, such as those which infinitely loop, have no final state and so properties of the trace as a whole are considered.

In my research, I consider the traces of an imperative language called While. While is a simple subset of C with support for variable assignment, conditional statements, sequential execution, and while loops. I chose to use While as the language I modeled in order to maintain a reasonable scope for the project, enabling exploration of the different modes of coinduction in Agda. Using While introduced some limitations, however. As While is a deterministic language, certain interesting properties of traces are not available. Traces are often used in concurrency theory for reasoning about behavior of concurrent processes

[San11], but While lacks a concept of concurrency. Further, While lacks a concept of interactivity through I/O. In practical applications, user input will change the behavior of the system, and this creates interesting considerations for program verification that can be reasoned about using traces. This is especially relevant for non-terminating applications, such as operating systems, which do not produce an end result and terminate but instead continuously react to input. Despite these limitations, While enables reasoning about some interesting properties of basic programs and their traces. Concretely, I attempt to verify three types of property in my research:

1. Proofs that the variables in a trace satisfy a specific property.
2. Proofs that a statement (program) results in a specific trace.
3. A proof that the language is deterministic.

These properties are of particular interest because they combine to provide a complete description of the function of a program. We can say, for example, that a given statement will *always* (property 3) produce a specific trace (property 2), and this trace infinitely produces the sequence of natural numbers (property 1), essentially giving us a guarantee of the overall behavior of the program. Despite While lacking features which would allow for verification of larger, practical applications, the chosen properties still serve as a proof-of-concept for analysis of non-terminating programs via their traces which may be extended to a more complex language in future research.

```
loop : Stmt
loop = Swhile
  (λ _ → 1)
  (Sassign 0 (λ st → st 0 + 1))
```

(a) A statement defining a while loop whose condition is always 1 (true) and whose body adds one to the variable at location 0.

```
while (1) {
    x = x + 1;
}
```

(b) C equivalent of loop.

Figure 1: Statement describing an infinite loop increasing the value of a variable by one in each loop, and its C equivalent.

My implementation of traces and semantics for While in Agda follows the work done by Nakata and Uustalu in Coq [NU09]. A summary of the syntax implemented is described by figure 2. The language assumes the presence of expressions without side effects, which are represented by lambda expressions in Agda. States are modeled as functions from identifiers to values. Concretely, this is a function $\mathbb{N} \to \mathbb{N}$. For the purpose of conditions in if statements and while loops, expressions evaluating to zero are considered false, and expressions evaluating to nonzero values are considered true.

The connection between statements, starting states, and traces, is defined by the semantics of the language. I implement a semantics equivalent to the big-step relational semantics implemented by Nakata and Uustalu. The semantics are split into an inductive and coinductive part — exec, relating a statement and starting state to a trace, and execseq, relating a statement and partial trace to a combined trace of running the statement from the end of the partial trace. Individual trace implementations are discussed in section 4 and discussion of experiments demonstrating the chosen properties is present in section 5.

```
data Stmt : Set where
  Sskip       : Stmt
  Sassign     : Id   → Expr → Stmt
  Sseq        : Stmt → Stmt → Stmt
  Sifthenelse : Expr → Stmt → Stmt → Stmt
  Swhile      : Expr → Stmt → Stmt
```

(a) Statements defined as an Agda data type. As an example, SWhile models a while loop, taking an expression (Expr) as the loop condition and statement (Stmt) as the loop body.

```
Id : Set
Id = ℕ

Val : Set
Val = ℕ

State : Set
State = Id → Val

Expr : Set
Expr = State → Val
```

(b) Supporting definitions required for statements.

Figure 2: Definition of statements as an Agda data type

3.2 Agda Coinduction

As a total language, Agda guarantees that all computations will terminate and outuput a value and as a result enjoys a closer connection between mathematics and code [Tur04]. However, totality introduces the restriction that potentially infinite data must be considered carefully in order to not allow potentially non-terminating computation over these infinite structures. Coinduction is the tool employed by languages such as Agda to counter this problem [VW19]. Agda supports multiple methods for performing coinduction, each of which is briefly described in the following sections. Each method of coinduction serves the purpose of guaranteeing *productivity*, defined as the requirement that any finite piece of the output is computable in only a finite number of steps [VW19] — essentially, the next "step" can always be made. In the example of an infinite sequence of values (referred to as a *stream*), productivity means that the next value can always be taken, even though the end of the structure is never reached.

3.2.1 Musical Coinduction

Musical coinduction involves the explicit use of "delay" and "force" operators when reasoning about coinductive structures. It is considered "old" and discouraged compared to the other methods of coinduction [Agd24].

The notation provides two functions: "delay" (represented by ♯), converting an element of a type to its delayed counterpart, and "force" (represented by ♭), converting a delayed expression to its value. The type of a delayed computation is represented by ∞. Using musical notation requires manual consideration of when types must be delayed in order to ensure productivity for the purposes of the termination checker. Specifically, corecursive calls can only occur within usages of a constructor which contains a delayed type [VW19]. These delays inform Agda where laziness in evaluation can occur, allowing for reasoning to be done over coinductive types without necessitating the full unfolding of the type's (potentially infinite) structure.

Shown in figure 3 is an example of a stream using musical coinduction. A stream consists of its head and the delayed remainder of the stream. Note that without the musical notation introducing the concept of delays, evaluating a stream would require infinitely unfolding its structure, entailing searching for a base case which does not exist and failing Agda's termination checker.

```
data Stream (A : Set a) : Set a where
  _::_ : (x : A) (xs : ∞ (Stream A)) → Stream A
```

(a) Definition of the Stream type. A stream is constructed as a pair of an element and a delayed remainder of the stream.

```
zeroes : Stream ℕ
zeroes = 0 :: ♯ zeroes
```

(b) Infinite stream of zeroes. The head of the stream is zero, and the tail is a delayed ♯ evaluation of the stream itself.

Figure 3: Musical stream definition and example.

3.2.2 Guarded Coinduction

Guarded coinduction relies on the use of coinductive record types. Where a typical inductive data type will be defined by its constructors — the functions used to create elements of the type, coinductive records are defined by their *destructors*, or what can be observed about the type. Coinductive records can be instantiated using *copattern matching*. The technique of copattern matching is described by Abel et al. [Abe+13].

An example of a stream using guarded coinduction is shown in figure 4. Here, a stream is defined by its head, a value, and tail, another stream. A stream of infinitely repeating numbers can be defined via copattern matching. Despite the structure being infinite in size, the computation is guarded by the copattern matching, ensuring that the next step of the computation can always be computed, satisfying productivity.

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

(a) Definition of a Stream as a coinductive record. The stream consists of a head, an element, and a tail, the remainder of the stream.

```
zeroes : Stream ℕ
zeroes .head = 0
zeroes .tail = zeroes
```

(b) Infinite stream of zeroes defined using copattern matching. The head of the stream is always zero, and the tail is the stream itself.

Figure 4: Guarded coinductive record stream definition and example.

3.2.3 Sized Types

Sized types introduce the concept of "sizes" to types. A size represents an upper bound on the number of unfoldings a structure can undergo [VW19]. Concretely, a sized list `1 :: 2 :: []` has size greater than `1 :: []` due to being made up of an additional constructor. Agda represents sizes using the types `Size` and `Size< i`, where for all j : `Size< i`, $j < i$ [Abe16]. To support coinduction, Agda has a size ∞ of potentially infinitely deep structures which satisfies i : `Size<` ∞ for all i [VW19].

Sizes guide the termination checker by allowing recursive calls which operate on a size strictly smaller than the current call, replacing the need for the strict criteria of syntactic guardedness [Abe16]. This requirement is seen in the example of a stream in figure 5. In

particular, indexing on the size i as seen in 5b is a useful pattern, allowing Agda to infer that a recursive call can take on a size less than the current size.

Sized types are known to have inconsistencies related to ∞ , particularly the notion that ∞ satisfies Size< ∞. [Cha21]. These inconsistencies allow for the derivation of contradictions in some cases [Abe17]. Resolution of these inconsistencies is still an open issue, and as a result sized types are no longer considered part of "safe" Agda [Coc21].

```
record Thunk {ℓ} (F : SizedSet ℓ) (i : Size) : Set ℓ where
  coinductive
  field force : {j : Size< i} → F j
open Thunk

data Stream (A : Set a) (i : Size) : Set a where
  _::_ : A → Thunk (Stream A) i → Stream A i
```

(a) Definition of a Stream using sized types. The reduction in size is encoded by the force field of Thunk. A Thunk represents a delayed computation and can be created with a copattern matching lambda, seen in figure 5b.

```
zeroesBad : Stream ℕ ∞
zeroesBad = 0 :: (λ where .force → zeroesBad)

zeroes : ∀ {i} → Stream ℕ i
zeroes = 0 :: (λ where .force → zeroes)
```

(b) Infinite stream of zeroes, showing the size reduction criteria. zeroesBad is rejected by the termination checker because the definition of size ∞ makes a recursive call of size ∞, and $\infty \not< \infty$.[3]Indexing on the size, as in zeroes allows Agda to see that the recursive call operates on a smaller size.

Figure 5: Sized types stream definition and example.

4 Encodings

My research implemented traces and semantics in each of Agda's three types of coinduction. Each encoding is discussed below, with particular attention drawn to differences arising from the different methods of coinduction.

4.1 Musical Coinduction

The first encoding implemented over the course of the research was done in Musical notation, as the most "natural" translation from the work of Nakata and Uustalu in Coq to Agda. A definition of traces and their bisimilarity is presented in figure 6. Bisimilarity is used as the equivalence relation over coinductive structures as structural equality would require a complete unfolding of the potentially infinite structure [Tur04].

I consider the encoding rather "natural" in that instances are constructed in a similar method to a non-coinductive list, either being an instance of tcons or tnil, with the exception that tnil takes a State due to the stipulation that traces are non-empty. Bisimilarity follows the structure of the trace — a trace consisting of a tnil is bisimilar to another tnil containing the same state, and given a (delayed) proof two traces are bisimilar, extending each trace with a state via tcons preserves bisimilarity. The remaining part of the encoding concerns the relational semantics of the language. An account of the semantics can be found in appendix B.1.

[3]Note that the failure here is in termination checking, *not* in type checking. ∞ is a term of Size< ∞. Certain definitions involving $\infty < \infty$ will (erroneously) be accepted by termination checking [Abe17].

```
data Trace₁ : Set where
  tnil  : State → Trace₁
  tcons : State → ∞ Trace₁ → Trace₁
```

(a) Definition of Trace_1. A trace is constructed from a single state (tnil) or by prepending a state to the delayed remainder of a trace, which may be infinite (tcons).

```
data _≈_ : Rel Trace₁ Level.zero where
  tnil  : ∀ {st} → (tnil st) ≈ (tnil st)
  tcons : ∀ {st tr₁ tr₂}
    → ∞ (♭ tr₁ ≈ ♭ tr₂)
    → (tcons st tr₁) ≈ (tcons st tr₂)
```

(b) Definition of bisimilarity for Trace_1. tnil states that two traces consisting of the same single state are bisimilar, and tcons states that two bisimilar traces may be prepended with another state.

Figure 6: Musical encoding of traces and bisimilarity.

4.2 Guarded Coinduction

My research using Agda's guarded coinduction led to two different possible encodings. Both encodings resulted in a result not suitable for the proofs I was able to define using the other methods, but the encodings are still discussed here.

4.2.1 Coinductive Record 1

The first coinductive record encoding, which I refer to as Trace_2, stems from an adaptation of the work on colists done by Abel et al. [Abe+13]. Traces are modeled as a mutual inductive data type and coinductive record. The inductive type provides a similar interface to non-coinductive lists in its constructors, much the same as the musical encoding discussed in 4.1. The record provides the coinductive part and allows Agda to ensure guardedness via use of copattern matching.

The definition of the inductive part rTrace_2 takes the same shape as the definition of Trace_1, with rTrace_2 corresponding to Trace_1 and Trace_2 corresponding to the lifted type $\infty\ \mathsf{Trace}_1$. Bisimilarity takes a similar approach, having an inductive part r≈ operating on rTrace_2 and coinductive part ≈ operating on Trace_2.

```
data rTrace₂ : Set where
  tnil  : State → rTrace₂
  tcons : State → Trace₂ → rTrace₂

record Trace₂ : Set where
  coinductive
  constructor mkTr
  field
    out : rTrace₂
```

(a) Trace definition using mutual induction (via rTrace_2) and coinductive record (via Trace_2). Instances of Trace_2 can be destructed by projecting on out and pattern matching on the constructors of rTrace_2.

```
data _r≈_ : Rel rTrace₂ Level.zero where
  tnil  : ∀ {st}
    → (tnil st) r≈ (tnil st)
  tcons : ∀ {st tr₁ tr₂} → tr₁ ≈ tr₂
    → (tcons st tr₁) r≈ (tcons st tr₂)

record _≈_ (tr₁ tr₂ : Trace₂) : Set where
  coinductive
  constructor mkBisim
  field
    p : (out tr₁) r≈ (out tr₂)
```

(b) Bisimilarity of Trace_2, mirroring the structure of the trace encoding itself.

Figure 7: Mutual induction and coinduction definition of traces and their bisimilarity using coinductive records.

4.2.2 Coinductive Record 2

The second coinductive record encoding attempted in my research follows the colist representation described by Ciccone [Cic20]. In this represenation, a trace is modeled as a coinductive record containing two fields: hd, a State present at the head of the trace, and tl, a Maybe Trace_3. The Maybe type encodes the fact that the tail of the trace may either be nothing, indicating that the trace ends, or contain another trace, indicating continuation. I found this encoding unsuitable for the experiments attempted on the other encodings. Discussion of the limitations of Trace_3 may be found in appendix B.2.

```
record Trace₃ : Set where
  coinductive
  constructor mkTr
  field
    hd : State
    tl : Maybe Trace₃
```

(a) Definition of Trace_3. The trace consists of a head and a tail. The head is a state, and the tail is either the remainder of the trace, or nothing. The "choice" between the two is encoded by the Maybe type.

```
record _≈_ (tr₁ tr₂ : Trace₃) : Set where
  coinductive
  field
    hd : hd tr₁ ≡ hd tr₂
    tl : (tl tr₁ ≡ nothing × tl tr₂ ≡ nothing)
         ⊎
         ∃ {A = (Trace₃ × Trace₃)} λ x → (
         tl tr₁ ≡ just (proj₁ x)
         ×
         tl tr₂ ≡ just (proj₂ x)
         ×
         (proj₁ x) ≈ (proj₂ x))
```

(b) Definition of bisimilarity over Trace_3. The head is a proof that the heads of the two traces are equal. The tail is a proof that either both tails are empty, or each a trace which is bisimilar to the other.

Figure 8: Definition and bisimilarity for Trace_3.

4.3 Sized Types

The sized implementation of traces extends a natural inductive definition of lists with sizes and Thunks. A Thunk, whose definition is shown in figure 5a, encodes a delayed computation and enforces the size-shrinking constraint. As can be seen in figure 9, the type of a trace is indexed by an i : Size. A (potentially infinite) trace has size bounded only by ∞, and so would have type $\text{Trace}_4\ \infty$. Instantiating a Thunk is more verbose than the musical delay ♯, requiring a copattern matching lambda, visible in figure 5b. exec and execseq are also both indexed by Size. Notably, exec is sized despite not directly using coinduction or Thunk in order to allow tracking of size reduction for its *uses* of execseq.

```
data Trace₄ (i : Size) : Set where
  tnil  : State → Trace₄ i
  tcons : State
    → Thunk Trace₄ i
    → Trace₄ i
```

(a) Definition of Trace_4. The Thunk in the tcons case encodes the size-reduction constraint on the structure.

```
data Bisim (i : Size) : Rel (Trace₄ ∞) Level.zero where
  tnil  : ∀ {st} → Bisim i (tnil st) (tnil st)
  tcons : ∀ {st tr₁ tr₂}
    → Thunk (λ s → Bisim s (force tr₁) (force tr₂)) i
    → Bisim i (tcons st tr₁) (tcons st tr₂)
```

(b) Definition of bisimilarity over Trace_4.

Figure 9: Definition and bisimilarity for Trace_4.

Table 1: Summary of the suitability of each encoding for each experiment type.

Type	Encoding	Proof		
		Trace	Program	Language
Musical	Trace_1	✓	✓	✓
Guarded	Trace_2	✓	✓	✗
	Trace_3	✗	✗	✗
Sized	Trace_4	✓	✓	✓

5 Experiments

My work performing experiments using the encodings discussed in section 4 can broadly be broken into three categories:

- Proofs that individual (infinite) traces satisfy a property.
- Proofs that individual programs satisfy a given (infinite) trace.
- A proof that the semantics define a deterministic language.

Each type of experiment was applied to each attempted encoding. Table 1 summarizes the results of each experiment for each encoding.

5.1 Trace-Property Satisfaction

The first proof type I considered aimed at proving properties of the variables described by traces. An example of such a property is "this trace produces, infinitely, the set of natural numbers, increasing by one every two states."[4] This property is exactly the property I attempted to prove for a trace in each encoding described in section 4.

In Agda, this proof takes the form of a term of a data type encoding that a variable in a trace is always incrementing in this "wait-step-wait" pattern. This type is coinductively defined, as it reasons about an infinte trace. This predicate defined using musical coinduction can be seen in figure 10. A term of this type, incrementingAlwaysIncrements : increasing 0 0 incrementingtrace, is interpreted as "the variable at location 0 begins with value zero in trace incrementingtrace, and follows the progression of increasing by one every two states."

As proofs of this type are reducible to proofs about properties of colists, it is unsurprising that given the existing use of colists in Agda, my research was able to create these proofs in each encoding for which they were attempted.[5]

5.2 Program-Trace Satisfaction

The second type of proof explored in my research was proofs, using the semantics, that a given program satisfies a given trace. That is, the execution of the program under a starting

[4] Justification for incrementing happening every two "steps" is provided in section 5.2

[5] As noted in section 4.2.2, my work on Trace_3 stopped after implementing the trace and its bisimilarity relation, due to the cumbersome nature of working with the encoding. I postulate that this proof is still possible for this trace type given the work done by Ciccone on the colist encoding on which Trace_3 is based, but this was not attempted in my research.

```
data increasing : Id → Val → Trace₁ → Set where
  increasingCons : {id : Id} {v : Val} {st : State} {tr tr₁ : Trace₁}
    → st id ≡ v
    → tr₁ ≈ tcons st (♯ (tcons st (♯ tr)))
    → ∞ (increasing id (suc v) tr)
    → increasing id v tr₁
```

Figure 10: Agda data type encoding the "increasing every two steps" property for traces, using musical coinduction. The proof may only be constructed through its constructor increasingCons, taking proofs that the variable in a state st takes the expected value v, that a trace tr_1 takes two steps in this state and continues on to trace tr, and that tr satisfies this predicate for the next value of v as a lifted (delayed) proof.

state will result in exactly a specific trace. A proof of this type has the type exec. For programs which do not terminate, these proofs will necessarily be coinductive, relying on the coinductive part of the semantics execseq. For example, consider proof : exec program fromState trace, where proof is a proof that program, run starting at fromState results in trace. Writing such a proof essentially involves writing out the execution path of the program using the different cases of exec. To define "success" for this proof type, I attempted to implement a proof exloopincrementing for each encoding type, encoding that the program Swhile (λ _ → 1) (Sassign 0 add1), intuitively understood as a while loop with a condition always evaluating to 1 (true) whose body increments the variable at position 0 at each iteration, starting from a state where this variable begins with value zero, results in a trace where 0 begins with value zero and increments by one each two "steps" of the trace.[6] An example of this proof type for the encoding Trace_1 may be found in appendix B.3. This proof type was successful for Trace_1, Trace_2, and Trace_4. This result is more notable than that of 5.1 as it depends on the semantics in addition to traces, essentially introducing another "layer" of coinductive structure to reason about.

5.3 Language Proof: Determinism

Where the previous two experiments have focused on properties of individual traces and programs, this type concerns the language as a whole. Given that While has no operations which can introduce nondeterminism such as concurrency or randomness, it is a natural conclusion that the language is deterministic. Concretely defined, determinism states that if one executes a statement from a given starting state twice, the resulting two traces from the executions must be the same.[7] This property was proven in Coq by Nakata and Uustalu in their work with While [NU09]. My proof follows the reasoning set down in theirs, translated into Agda. In Agda, this proof takes the form of a function execDeterministic : {s : Stmt} {st : State} {tr_1 tr_2 : Trace} → exec s st tr_1 → exec s st tr_2 → $tr_1 \approx tr_2$. Implementation of this proof involves multiple helper lemmas, for example proving that execseq is also deterministic or proving determinism specifically for the SWhile case. Due to the complexity of the proof, this experiment took the majority of the time in my research and revealed the most limitations with Agda's support for coinduction.

This proof was successful for Trace_1 and Trace_4 and attempted but unsuccessful for Trace_2. Specifically, Trace_2 became stuck at the first helper lemma $\text{execseqDeterministic}_0$.

[6] Each increment happens after two "steps" due to one step being taken to evaluate the loop condition, and another to assign the variable.

[7] When considering coinductive traces, "the same" is taken to mean bisimilarity.

The underlying limitations are discussed further in section 6.3. Implementation for Trace_4 was significantly smoother than for Trace_1 due to the lack of the strict condition of guardedness described in section 6.3, even allowing for one helper lemma to be dropped entirely.

6 Agda Insights

In implementing the encodings and experiments, various complications and insights arising from Agda's limitations around coinduction became apparent. In some cases, these limitations led me to determine that an encoding is not suitable for use in Agda in the current state. Some complications arose in regards to specific encodings, and some points apply to coinduction in Agda as a whole. These difficulties are discussed here with the goal of suggesting ways that coinduction support in Agda can be improved.

6.1 Documentation

Documentation around coinduction in Agda is very limited. The official documentation for coinduction offers limited examples of how to use the different types of coinduction. Often, I found in my research that the best way to learn was to analyze the implementation of structures in the Agda standard library, essentially working backwards from an understanding of specific implementations to a general understanding of the methods of applying the techniques used in the implementations. Perhaps most egregiously, the main page for coinduction on Agda's documentation does not describe sized types at all, only linking to a stub article presenting an example of the usage of Sized types [Agd24]. Learning about the constructs Size and Size<, as well as how sized types may be used, was only possible through reading the Agda standard library code and published literature which used sized types.

6.2 Error Messages

While working with coinduction in Agda, error messages are frequent, and often difficult to interpret. Particularly, when facing issues with termination checking, Agda will highlight any call that can form a part of a non-terminating computation. This makes it difficult to identify which call is actually causing the problem, often necessitating commenting out parts of code one-by-one to find the problem. Additionally, when encountering unification problems, as shown in figure 13, the error message does not make the underlying problem (indexing on the result of function application) clear. This makes it necessary to seek out external resources to identify the root cause of problems, making debugging a more involved process than if the error messages could guide a user towards a solution more directly.

6.3 Guardedness

Guardedness is used as the productivity condition for both the musical and guarded coinductive record methods of coinduction. Guardedness demands that a guarded call have no non-constructor function between the left-hand side of the function and a corecursive call [DA10]. This condition is too strict, and can exclude productive definitions. Consider the minimized example over coinductive natural numbers presented in figure 11. inf is accepted as productive by the termination checker, because only a constructor appears before the corecrusive call. It is clear that each "step" of the definition can be unfolded one call to suc

further. However, Agda rejects the definition of infBad due to the call to id. However, I argue that infBad is clearly still productive — id does nothing to the data, and as such is essentially removable. Indeed, this example passes termination checking with sized types, as sized types do not use guardedness as the criterion for productivity.[8] The effect of this limitation is that special care must be taken to ensure guardedness, sometimes requiring the use of additional helper lemmas. In one case, described in appendix A, this limitation contributed to making an experiment infeasible.

```
mutual
  data Coℕr : Set where
    zero : Coℕr
    suc  : Coℕ → Coℕr

  record Coℕ : Set where
    coinductive
    field
      out : Coℕr
```

(a) Example datastructure mirroring the encoding of Trace_2, the conaturals (potentially infinite natural numbers).

```
id : ∀ {a} {A : Set a} → A → A
id x = x

inf : Coℕ
inf .out = suc inf

infBad : Coℕ
infBad .out = id (suc infBad)
```

(b) inf is accepted by the termination checker, but infBad is rejected due to the presence of a non-constructor function before the corecursive call, despite the function id having no effect.

Figure 11: Minimized example showing the strictness of guardedness as a criteria for productivity.

6.4 Musical: Delay Application in Types

As noted by the documentation on musical coinduction, a known limitation is that indexing types on applications of delay ♯ should be avoided [Agd24]. Consider the example in figure 12a. To avoid applying delay in the type of suc, as in the case of ≈bad, it is necessary to lift n and m so that they can be given to suc without delays. This is to prevent similar problems related to indexing on function applicaton seen in section 6.5.1. I refer to this restriction as the need to "move the music left" in a definition. In figure 12b, tr had to be introduced and made bisimilar to the result of the execution, so that the result could be constructed with the use of delay without indexing the type of execWhileFalse on this application of delay. This limitation increases the complexity of defining types using musical coinduction.

6.5 Coinductive Records

6.5.1 Unification Under Function Application

Indexing types on the result of function application is a known cause of unification problems in Agda, with general advice being to only use variables and constructors in indices of data types [Coc17]. However, the use of mutual induction and coinduction in the encoding of Trace_2 and related types necessitates that the coinductive part of bisimilarity indexing on the result of out, as can be seen in figure 7. These unification problems restrict Agda's knowledge when destructing terms of these types. The effects of this limitation can be seen in the example described in appendix A.

[8] The example shown in figure 11 as well as an equivalent example using sized types which passes the termination checker are both present in the repository as the modules `TermCheck` and `TermCheckSized`.

```
data _≈bad_ : Coℕ → Coℕ → Set where
  zero : zero ≈bad zero
  suc  : {n m : Coℕ}
    → ∞ (n ≈bad m)
    → suc (♯ n) ≈bad suc (♯ m)

data _≈good_ : Coℕ → Coℕ → Set where
  zero : zero ≈good zero
  suc  : {n m : ∞ Coℕ}
    → ∞ (♭ n ≈good ♭ m)
    → suc n ≈good suc m
```

(a) Example showing bisimilarity over the conaturals both with and without ♯ in the type of suc.

```
data exec : Stmt → State → Trace₁ → Set where
  execWhileFalse :
    {c : Expr} {st : State} {tr : Trace₁} (b : Stmt)
    → isTrue (c st) ≡ false
    → tr ≈ (tcons st (♯ tnil st))
    → exec (Swhile c b) st tr
```

(b) Example of the limitation necessitating a different definition in the semantics. Here, to avoid use of ♯ in the type of execWhileFalse, tr has to be introduced and defined bisimilar to the trace that should result from this call.

Figure 12: Examples of avoiding the application of delay in types.

6.5.2 Pattern Matching

Agda does not allow pattern matching on the fields of coinductive records. As such, to destruct an instance of a coinductive record it is necessary to project on its field and use a with-abstraction to pattern match on the result. This limitation often leads to multiple nested layers of with-abstraction, increasing the verbosity of proofs and making them more time consuming to write.

6.6 Summary

My findings regarding the different methods of coinduction in Agda may be summarized for each coinduction type as follows:

1. Musical coinduction, despite its success in my experimentation, suffered from proofs being made more cumbersome by restrictions on delay application in types and the strictness of guardedness.

2. Guarded coinduction using coinductive records is poorly suited to my application. For Trace_2, the limitations which prevented complete implementation of the experiments were a combination of the strictness of guardedness as well as limitations surrounding unification under function application. For Trace_3, the verbosity introduced by the need to account for a potential end to a trace made working with the encoding impractical. I conclude that guarded coinduction in Agda is poorly suited to applications such as mine involving nested layers of coinductive records which necessarily have the concept of "choice", understood as having multiple possible constructors in an inductive setting. Either a more refined criteria for guardedness, if possible, or improvement of Agda's ability to unify under function application (or specifically under projection) would allow for proofs such as the experiments attempted in my research to succeed.

3. Sized types were the best suited to my application, having the smoothest experience in implementing the experiments I attempted. However, this method suffered the most from a lack of documentation, which initially discouraged me from attempting an encoding using sized types. Additionally, the known issues with soundness of sized types discussed in 3.2.3 serve as a limitation of trust in the correctness of the proofs.

7 Conclusions and Future Work

My research attempted to use the three methods of coinduction (musical, guarded, and sized) available in Agda to encode coinductive program traces and their semantics. To this end, I implemented traces and semantics for While, a simple imperative language, following the work of Nakata and Uustalu in Coq. Subsequently, I used the encodings to perform experiments, attempting to prove properties of traces, programs, and the language as a whole. This work revealed limitations present in Agda's coinduction support.

Musical coinduction, despite being considered the "old" way of performing coinduction, had success in my experimentation. Its closeness to typical inductive data types worked well for my colist-based trace representation. However, limitations were still present in musical coinduction, particularly related to reasoning about guardedness and avoiding indexing on delays introducing additional complexity.

Guarded coinduction experienced more difficulties in use. Encodings either modeled the "choice" between constructors poorly (as in the case of Trace_3) or experienced a combination of limitations due to unification under function application and strict guardedness criteria that made some experiments not feasible (as in the case of Trace_2). This led me to conclude that guarded coinduction is poorly suited for my application at this time. Future work exploring methods of improving unification under fields of coinductive records, identifying cases which lack guardedness but are still productive, or exploration into alternative methods of modeling colist-like structures using guarded coinduction may help to fill this gap.

Sized types presented the smoothest experience of implementation with the least battles with the termination checker. It is for this reason that I consider it regrettable that sized types have the most lacking documentation of the three coinduction methods in my opinion, only earning a link to a stub article on the main coinduction documentation page. I was initially discouraged from attempting to use sized types by this disparity in documentation.

More generally, improvements to documentation would benefit all three of the coinduction methods, in addition to clearer error messages. Clearer examples in the documentation, rather than exclusively in the implementation of the standard library, may make getting started with coinduction more accessible. Improvements to error messages, particularly around unification issues and termination checking, would help to make debugging coinductive definitions less cumbersome.

The limitations of my research primarily stem from time limitations and relative inexperience in utilizing coinduction. For example, I elected to not pursue experimentation on the trace encoding Trace_3 in order to preserve time for other coinduction methods due to the difficulty of working with Trace_3. Additionally, while I have attempted to verify my claims that difficulties encountered in my research represent limitations inherent to the coinduction implementations in Agda, through the use of minimized examples and comparison to known existing limitations, the possibility remains that workarounds exist which I was unable to identify in my research. Future research should aim to assess the extent to which the limitations I identify are legitimate shortcomings of Agda.

My research regarding coinductive traces and their semantics in Agda provides an exploration into the capabilities and limitations of coinduction in Agda. Future work in the area may utilize these limitations I have identified to improve this coinduction support, with the aim of making coinduction a more accessible part of Agda. Given the wide applications of infinite structures [San11], improvement to coinduction in Agda will improve its versatility as an interactive proof assistant, potentially allowing the verification of larger, practical non-terminating programs.

References

[Abe+13] Andreas Abel et al. "Copatterns: programming infinite structures by observations". In: *Proceedings of the 40th annual ACM SIGPLAN-SIGACT symposium on Principles of programming languages*. POPL '13: The 40th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages. Rome Italy: ACM, 23rd Jan. 2013, pp. 27–38. ISBN: 978-1-4503-1832-7. DOI: 10.1145/2429069.2429075. URL: https://dl.acm.org/doi/10.1145/2429069.2429075 (visited on 22/04/2025).

[Abe16] Andreas Abel. *Equational Reasoning about Formal Languages in Coalgebraic Style*. 23rd Dec. 2016. URL: https://www.cse.chalmers.se/~abela/jlamp17.pdf (visited on 05/06/2025).

[Abe17] Andreas Abel. *Equality is incompatible with sized types · Issue #2820 · agda/agda*. GitHub. 27th Oct. 2017. URL: https://github.com/agda/agda/issues/2820 (visited on 18/06/2025).

[ADZ18] Davide Ancona, Francesco Dagnino and Elena Zucca. "Modeling Infinite Behaviour by Corules". In: *LIPIcs, Volume 109, ECOOP 2018* 109 (2018). Ed. by Todd Millstein. Artwork Size: 31 pages, 801714 bytes ISBN: 9783959770798 Medium: application/pdf Publisher: Schloss Dagstuhl – Leibniz-Zentrum für Informatik, 21:1–21:31. ISSN: 1868-8969. DOI: 10.4230/LIPICS.ECOOP.2018.21. URL: https://drops.dagstuhl.de/entities/document/10.4230/LIPIcs.ECOOP.2018.21 (visited on 23/04/2025).

[Agd24] Agda Developers. *Coinduction — Agda 2.7.0.1 documentation*. 12th Sept. 2024. URL: https://agda.readthedocs.io/en/v2.7.0.1/language/coinduction.html (visited on 23/05/2025).

[CA18] Jesper Cockx and Andreas Abel. "Elaborating dependent (co)pattern matching". In: *Proceedings of the ACM on Programming Languages* 2 (ICFP 30th July 2018), pp. 1–30. ISSN: 2475-1421. DOI: 10.1145/3236770. URL: https://dl.acm.org/doi/10.1145/3236770 (visited on 26/04/2025).

[Cha21] Jonathan Chan. *The State of Sized Types*. ⟨ortho|normal⟩. Section: Type Theory. 4th Aug. 2021. URL: https://ionathan.ch//2021/08/04/sized-types (visited on 18/06/2025).

[Cic20] Luca Ciccone. *Flexible Coinduction in Agda*. 17th Feb. 2020. DOI: 10.48550/arXiv.2002.06047. arXiv: 2002.06047[cs]. URL: http://arxiv.org/abs/2002.06047 (visited on 28/05/2025).

[Coc17] Jesper Cockx. *[Agda] Cannot apply injectivity*. E-mail. 21st Sept. 2017. URL: https://lists.chalmers.se/pipermail/agda/2017/009739.html (visited on 31/05/2025).

[Coc21] Jesper Cockx. *Sized types are no longer safe by jespercockx · Pull Request #5354 · agda/agda*. 7th May 2021. URL: https://github.com/agda/agda/pull/5354 (visited on 18/06/2025).

[DA10] Nils Anders Danielsson and Thorsten Altenkirch. "Subtyping, Declaratively". In: *Mathematics of Program Construction*. Ed. by Claude Bolduc, Jules Desharnais and Béchir Ktari. Red. by David Hutchison et al. Vol. 6120. Series Title: Lecture Notes in Computer Science. Berlin, Heidelberg: Springer Berlin Heidelberg, 2010,

pp. 100–118. ISBN: 978-3-642-13320-6 978-3-642-13321-3. DOI: 10.1007/978-3-642-13321-3_8. URL: http://link.springer.com/10.1007/978-3-642-13321-3_8 (visited on 31/05/2025).

[Geu09] H. Geuvers. "Proof assistants: History, ideas and future". In: *Sadhana* 34.1 (Feb. 2009), pp. 3–25. ISSN: 0256-2499, 0973-7677. DOI: 10.1007/s12046-009-0001-5. URL: http://link.springer.com/10.1007/s12046-009-0001-5 (visited on 02/05/2025).

[LG09] Xavier Leroy and Hervé Grall. "Coinductive big-step operational semantics". In: *Information and Computation* 207.2 (Feb. 2009), pp. 284–304. ISSN: 08905401. DOI: 10.1016/j.ic.2007.12.004. URL: https://linkinghub.elsevier.com/retrieve/pii/S0890540108001296 (visited on 25/04/2025).

[NU09] Keiko Nakata and Tarmo Uustalu. "Trace-Based Coinductive Operational Semantics for While". In: *Theorem Proving in Higher Order Logics*. Ed. by Stefan Berghofer et al. Vol. 5674. Series Title: Lecture Notes in Computer Science. Berlin, Heidelberg: Springer Berlin Heidelberg, 2009, pp. 375–390. ISBN: 978-3-642-03358-2 978-3-642-03359-9. DOI: 10.1007/978-3-642-03359-9_26. URL: http://link.springer.com/10.1007/978-3-642-03359-9_26 (visited on 25/04/2025).

[San11] Davide Sangiorgi. *Introduction to Bisimulation and Coinduction*. 1st ed. Cambridge University Press, 13th Oct. 2011. ISBN: 978-1-107-00363-7 978-0-511-77711-0. DOI: 10.1017/CBO9780511777110. URL: https://www.cambridge.org/core/product/identifier/9780511777110/type/book (visited on 23/04/2025).

[Sch98] David A. Schmidt. "Trace-Based Abstract Interpretation of Operational Semantics". In: *LISP and Symbolic Computation* 10.3 (1998), pp. 237–271. ISSN: 08924635. DOI: 10.1023/A:1007734417713. URL: http://link.springer.com/10.1023/A:1007734417713 (visited on 25/04/2025).

[Sun+24] Yujie Sun et al. "AI hallucination: towards a comprehensive classification of distorted information in artificial intelligence-generated content". In: *Humanities and Social Sciences Communications* 11.1 (27th Sept. 2024), p. 1278. ISSN: 2662-9992. DOI: 10.1057/s41599-024-03811-x. URL: https://www.nature.com/articles/s41599-024-03811-x (visited on 29/05/2025).

[The10] The Yale Law School Roundtable on Data and Code Sharing. "Reproducible Research". In: *Computing in Science & Engineering* 12.5 (Sept. 2010), pp. 8–13. ISSN: 1521-9615. DOI: 10.1109/MCSE.2010.113. URL: http://ieeexplore.ieee.org/document/5562471/ (visited on 15/05/2025).

[Tur04] D. A. Turner. "Total Functional Programming". In: *JUCS - Journal of Universal Computer Science* 10.7 (28th July 2004). Publisher: Journal of Universal Computer Science, pp. 751–768. ISSN: 0948-695X. DOI: 10.3217/jucs-010-07-0751. URL: https://doi.org/10.3217/jucs-010-07-0751.

[VW19] Niccolò Veltri and Niels van der Weide. "Guarded Recursion in Agda via Sized Types". In: *LIPIcs, Volume 131, FSCD 2019* 131 (2019). Ed. by Herman Geuvers. Artwork Size: 19 pages, 527018 bytes ISBN: 9783959771078 Medium: application/pdf Publisher: Schloss Dagstuhl – Leibniz-Zentrum für Informatik Version Number: 1.0, 32:1–32:19. ISSN: 1868-8969. DOI: 10.4230/LIPICS.FSCD.2019.3

2. URL: https://drops.dagstuhl.de/entities/document/10.4230/LIPIcs.FSCD.2019.32 (visited on 23/05/2025).

A Example: Trace$_2$ Stuck Proof

This section describes a concrete example of a proof which I believe was made infeasible due to the combination of the restrictions described in sections 6.3 and 6.5.1. The proof of determinism for Trace$_2$ became stuck at the helper lemma execseqDeterministic$_0$.

I begin by considering the effects of the limitation described by section 6.5.1. Consider the example depicted in figure 13. Here, *tnil?* has type out tr_1 r≈ out tr_2. Additionally, res_1 has type $tr_1 \approx$ mkTr (tnil st) and res_2 has type $tr_2 \approx$ mkTr (tnil st_1). Given this, and the hypothesis $tr_1 \approx tr_2$, it should be obvious that the only shape *tnil?* can take is tnil. However, attempting to case split on *tnil?* results on an error arising from an unsolved unification. The problem can be worked around through manual application of symmetry and transitivity of the known bisimilarities such that *tnil?* has a shape with tnil on both sides.

```
execseqDeterministic₀ : {s : Stmt} {tr₁ tr₂ tr₃ tr₄ : Trace₂}
  → tr₁ ≈ tr₂ → execseq s tr₁ tr₃ → execseq s tr₂ tr₄ → tr₃ ≈ tr₄
execseqDeterministic₀ tr₁≈tr₂ exs₁ exs₂ ._≈_.p with (execseq.p exs₁) | execseq.p (exs₂)
... | rexecseqNil res₁ ex₁ | rexecseqNil res₂ ex₂ with _≈_.p tr₁≈tr₂
... | tnil? = {! !}
-- Remaining cases omitted for brevity
```

I'm not sure if there should be a case for the constructor tnil, because I get stuck when trying to solve the following unification problems (inferred index $\stackrel{?}{=}$ expected index):

tnil st$_2$ $\stackrel{?}{=}$ Trace$_2$.out tr$_5$

tnil st$_2$ $\stackrel{?}{=}$ Trace$_2$.out tr$_5$

when checking that the expression ? has type Trace$_2$.out tr$_3$ r≈ Trace$_2$.out tr$_4$

Figure 13: Example showing a case where Agda is unable to unify under funciton application and resulting error message. Attempting to case split on *tnil?* results in the shown error message, even though it is "clear" that tnil is the only possible case for *tnil?*.

The necessity of this workaround additionally means that absurd cases must be proven manually. For example, in the case of execseqDeterministic$_0$ where we have one execseq in the rexecseqNil case and one in the rexecseqCons case, we have $tr_1 \approx$ mkTr (tnil st) and $tr_2 \approx$ mkTr (tcons st_2 tr_3). Given $tr_1 \approx tr_2$ it should be clear that this case must be absurd, as there is no way for a trace constructed by tcons to be the same as one constructed by tnil. Indeed, when pattern matching using musical coinduction or sized types, Agda is able to automatically exclude this case. Yet, due to the limitations on unification it is necessary to apply the workaround to "show" Agda that this case must be absurd, increasing proof complexity.

However, this issue goes beyond the nuisance of proving absurd cases. The limitation described above can combine with the strictness of guardedness described in section 6.3 to create an unfillable hole. While the criteria for guardeness is not unique to coinductive records — the same criteria applies to musical coinduction — the need to use function application to reshape variables to help with unification can create problems unique to

guarded coinduction. This occurred in the double rexecseqCons case of execseqDeterministic$_0$. The hole had shape out tr_a r≈ out tr_b. Following the proofs laid out by Nakata and Uustalu, the hole must be filled by an application of tcons containing a corecursive call. However, Agda cannot "see" that the hole can be filled with tcons due to constraints of unifying under function application. Using the knowledge available at this point in the proof, it is possible to again use symmetry and transitivity to define a function reshape that fills the hole and takes an argument tcons st tr_c r≈ tcons st tr_d, which can now be filled with tcons and the recursive call. Yet, the application of reshape means that the call is not guarded and as such does not pass termination checking. I argue that the definition is still productive, only applying symmetry and transitivity to what is otherwise a guarded call, but due to the combination of restrictions to unification and guardedness checking, I was unable to fill this hole in a way accepted by Agda. As such, the proof of determinism for the semantics of Trace$_2$ failed. The attempted implementation is preserved in the repository in `BigRel2.agda`.

B Additional Figures and Technical Discussion

This section contains additional implementation discussion which I consider interesting but is not strictly necessary for understanding of the report body.

B.1 Semantics Example

Figure 14 presents an overview of the semantics as implemented for the encoding of Trace$_1$. As discussed in section 3.1, the semantics consist of two parts, exec, relating statements and starting states to the resulting trace, and execseq, responsible for relating a statement from a starting *trace*. exec and execseq are defined via mutual induction and coinduction. Complete semantics implementations for each encoding (except Trace$_3$, which did not have semantics implemented) may be found in the `BigRel` family of modules in the repository.

```
data exec : Stmt → State → Trace₁ → Set where
  execWhileLoop :
    {c : Expr} {b : Stmt}
    {st : State} (tr tr′ : Trace₁)
    → (isTrue (c st)) ≡ true
    → execseq b (tcons st (♯ tnil st)) tr
    → execseq (Swhile c b) tr tr′
    → exec (Swhile c b) st tr′
```

(a) Encoding of exec with example case for a looping while loop. Intuitively, the definition says that given a proof that the condition of the loop is true in the state, a proof that executing the body after the "step" added to the trace as a result of checking the condition[9] results in a trace tr, and a proof that executing the while loop at the end of this trace results in another trace tr', then executing the while statement from the current state results in this trace tr'.

```
data execseq : Stmt → Trace₁ → Trace₁ → Set where
  execseqNil : {st : State} {s : Stmt} {tr : Trace₁}
    → exec s st tr
    → execseq s (tnil st) tr

  execseqCons : {s : Stmt}
    (st : State) (tr tr′ : ∞ Trace₁)
    → ∞ (execseq s (♭ tr) (♭ tr′))
    → execseq s (tcons st tr) (tcons st tr′)
```

(b) Encoding of execseq, stating that executing a statement from the end of a given trace results in another trace. The case execseqNil states that given a proof (exec) executing s from st results in tr, then sequentially executing s from a state containing only st results in tr. execseqCons encodes that an execseq may always be extended with an additional "step" consisting of a state. This execseq is delayed with ∞ to ensure guardedness.

Figure 14: Musical encoding of exec and execseq.

[9] Justification for guard checking adding to the trace is described by Nakata and Uustalu [NU09].

B.2 Analysis of Trace_3

This section provides an analysis of the challenges encountered when working with Trace_3, the encoding described by section 4.2.2. The encoding is illustrated in figure 8.

While the definition of Trace_3 itself is relatively readable, bisimilarity is where the problems inherent to the encoding begin to present themselves. The type of bisimilarity is now significantly more complex, as compared to other encodings. The head hd is a simple proof that the heads of the two traces are equal, but the tail tl is more difficult to decipher. Arising from the choice of the trace either continuing or not, tl is a sum type (denoted by $\uplus$) indicating that terms of the type can be one of two options. The first option is a pair (product type, $\times$) of proofs that the tail of each trace is empty by being equal to the nothing constructor of Maybe. The second option is a dependent product type corresponding to existential quantification, $\exists$, encoding that two traces exist which are the tails of the traces we are comparing and are themselves bisimilar.

The definition of bisimilarity already begins to provide a hint as to the challenges associated with this encoding. The "choice" introduced by a trace potentially ending, and the resulting $\uplus$ type, makes the type of bisimilarity and reasoning about bisimilarity cumbersome. Working with an instance of bisimilarity involves projecting on tl and then matching on its subtypes, resulting on deep pattern matching on the constructors of the sum and product types in order to get access to each subfield. This makes it difficult to keep track of all the necessary cases and what each case represents. As an example of this challenge, in proving that bisimilarity is a setoid, proof of transitivity is required. That is, given $tr_1 \approx tr_2$ and $tr_2 \approx tr_3$ it can be concluded $tr_1 \approx tr_3$. In other encodings, this proof is quite simple, taking less than ten simple lines of Agda. However, under this encoding the proof took 34 lines with deep pattern matching, manual proof of absurd cases, and manual application of injectivity of just.[10] My results reflect those of Ciccone, who concluded that colists represented as coinductive records scale poorly [Cic20]. Due to these difficulties encountered before even the semantics were implemented, I elected to not pursue this encoding further and focus efforts elsewhere.

[10] Setoid instances for all encodings can be found in `MusicalTraces.agda`, `GuardedTraces.agda`, and `SizedTraces.agda` in the repository.

B.3 Example Proof of Trace Satisfaction

Figure 15 shows a proof of the trace satisfaction property for an program describing an infinitely incrementing trace, using musical coinduction. This property is discussed in section 5.2.

```
exloopincrementing : exec (Swhile (λ _ → 1) (Sassign 0 add1)) startState incrementingtrace
exloopincrementing = forever startState
  where
    forever : (st : State) → exec (Swhile (λ _ → 1) (Sassign 0 add1)) st (incrementingFrom st)
    forever st = execWhileLoop
      (tcons st (♯ (tcons st (♯ (tnil (update 0 (add1 st) st))))))
      _
      _≡_.refl
      (execseqCons st _ _ (♯ execseqNil (execAssign (tcons (♯ tnil)))))
      (execseqCons st _ _ (♯ (execseqCons st _ _ (♯ (execseqNil (forever (next st))))))))
```

Figure 15: Example showing satisfaction of an infinite trace for a program which loops forever, adding one to the variable at position 0. incrementingtrace starts with 0 having value zero, then increasing by one every two states. The helper function forever creates the execution path coinductively, incrementally proving that each successive sub-trace is satisfied by the next application of execWhileLoop.[11]

[11] Not all helper functions are shown here. The proof, complete with all helper definitions, is present in the repository.

C Use of Generative AI

As discussed in section 2, my use of generative AI was limited to gaining an understanding of concepts in Agda, comprehending proofs written in Coq, asking abstract questions about LaTeX, and brainstorming ideas for creative visualizations. In table 2 I provide a listing of prompts used in the project to illustrate my use of AI.

Table 2: Listing of prompts given to ChatGPT

Prompt
How can I visualize the idea of a non-terminating while loop in an attention-grabbing way? I'm making an academic poster about program traces for potentially non-terminating computations.
How can I read this proof written in Coq using the ssreflect library? [excerpt of proof from Nakata and Uustalu]
In agda, is there a way to shorten a lambda definition of the form x -> x + 1?
I'm working with Agda. I have x1 == x2, x3 == x4 (propositional equality) and x1 REL x3, how can I then write x2 REL x4?
How can I have subfigures in latex? Follow-up prompts: • What are the [b] and [htbp] optoins in the example you showed? • Can I have the subfigures automatically figure out their width? • How can I get the "append" symbol in latex, represented in ascii by "::"?
Please provide an explanation of this proof in Coq. I find the syntax difficult to understand, coming from Agda. [excerpt of proof from Nakata and Uustalu] Follow-up prompts: • Explain the variable destructions in the helper lemma COINDHYP2. All the different "h3" "H0" are difficult for me to follow. • In the step - by apply bisim_reflexive. (* Tnil st1 ~ Tnil st1 *), how does proving that then prove that tr3 ~ tr4? Why does this work without requiring a recursive call? • Can you provide a more in depth view of the variables for the outer lemma now that I have the inner helper lemma?
What is the citation style that has author initials and a year in square brackets, such as [SPB99] ?
What is the symbol ℓ in Latex? I'm using the amsmath package for math symbols, if that's relevant.