Productively recursing infinitely

Modelling evaluation of lambda calculus with coinduction in Agda

Sarah van de Noort[1]
Supervisors: 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: Sarah van de Noort
Final project course: CSE3000 Research Project
Thesis committee: Jesper Cockx, Bohdan Liesnikov, Diomidis Spinellis

An electronic version of this thesis is available at http://repository.tudelft.nl/.

Abstract

Coinduction is used to model infinite data or cycles in Agda. However, it is not as well explored in Agda as induction. Therefore, support for it might be lacking compared to induction. I explore how this applies for the evaluation of lambda calculus, what the different encodings of lambda calculus using coinduction are, and how they compare to each other and to an inductive evaluator.

The two models I looked at are modelling cycles in variable references and modelling cycles in recursive variables. Cycles in variable references can be modelled coinductively, however, they do not help with evaluation. Since the evaluator is not coinductive, it is not accepted by the termination checker, therefore, it is not safer than an inductive evaluator. Encoding recursion using coinduction does make the evaluator terminate, aiding in creating a correct evaluator. This comes with the downside of sacrificing clarity and ease of reasoning about the code.

1 Introduction

While in Agda, a programming language, finite data structures like lists are modelled using induction, this doesn't work for infinite structures or data with cycles in them. Coinduction is used to encode these structures in Agda anyway. However, coinduction is not as mature as induction is. Research is needed to discover where it is and is not useful.

An example where cyclic structures appear is lambda calculus. One of the ways to model functional programming languages like Haskell is in the from of lambda calculus [Hud+07]. Thus, understanding lambda calculus and its evaluation is beneficial to understanding functional programming languages as a whole.

The research questions I attempt to answer are:

1. What are the different ways to model evaluation of lambda calculus using cyclic data-structures, and thus coinduction?

2. How do the models compare to each other in terms of ease of implementation and their limitations?

3. How suitable are these models to Agda and what are limitations Agda has that got in the way of evaluating lambda expressions?

For the evaluation of lambda calculus there are multiple different places where infinite or cyclic data-structures can be used. When modelling a lambda expression, variables used within the body of a lambda function can refer back to the arguments of the function, creating a cycle. Secondly, lambda expressions can be recursive, possibly infinitely so. Coinduction is needed to represent this infinite computation.

In previous work evaluators for lambda calculus have been made, like the evaluator by Gomard and Jones in Lisp [GJ91]. An inductive evaluator was made in Agda by Asai [Asa19] and a coinductive evaluator for a typed lambda calculus was made by Abel and Chapman [AC14]. In this paper I focus on an untyped lambda calculus with a language construct for recursion.

The structure of the paper is as follows: First I outline the background of the research in section 3. Then in section 4 I explain the different encodings and what the differences are between them. In the next section, section 5, I describe the evaluation of induction and coinduction for recursion. And finally I point out drawbacks Agda currently has in section 6.

2 Responsible Research

The research conducted consisted of programming in Agda. The code has been made publicly available which can be used to reproduce my results[1]. This code comes with installation instructions aiding other people with setting up the environment needed to look at, and work with, the code themselves. Additionally, in this report I outline the decisions I made with regards to what and how to program and what experiences made me come to the conclusions outlined in section 7.

Because of time constraints I was not able to explore all encodings equally; I implemented one coinductive encoding almost fully while the other one I considered but did not implement. This could have introduced a bias: if the results of the model using coinduction for variable references were to be different than expected, my conclusions could be different too.

In terms of ethical concerns, I do not see big issues with this research. This project does not involve sensitive or personal data. Additionally, lambda calculus itself is a theoretical model, with other languages that can be used in practise built on top of it. While functional languages could be used for things like discrimination, this is not the goal of these languages. Evaluators for these languages already exist, therefore I do not think this research would make it easier to do possibly illegal or unethical things.

While generative AI can be a useful tool in drafting a report or coming up with ideas while programming, I prefer to not use it. I have used autocompletion tools like the autocomplete in Overleaf and the autofill functionality Agda provides, but no generative AI.

3 Background

3.1 Agda & Coinduction

Agda is a functional, total programming language intended to be used as a proof assistant. A total language is a language that guarantees any expression in that language terminates and will not give a runtime error. Constructs that would infinitely loop are not allowed in Agda. This totality, together with its dependent type system, make Agda lend itself well for aiding in writing proofs.

An issue with a total language is that infinite computations cannot be represented without explicit support. Infinite computations and infinite data structures are not accepted by Agda's termination checker. Coinduction is a way to model infinite data while still abiding by the requirements of the termination checker. Certain coinductive data is valid as long as any single step can be derived in a finite amount of processing. For example, computing the 10th element of an infinite stream, should not take an infinite amount of time. This is called productivity [Coq94; VW19]. Productivity is ensured by enforcing guardedness, this means that between the recursive call and the respective representation's way of encoding coinduction there should not be any non-constructor functions [DA10].

There are multiple styles of coinduction in Agda: Guarded, Musical and Sized types. My implementation mainly follows guarded style, but I explain what would need to be changed to use musical style instead. I do not explore sized types in my research, therefore I do not explain it as broadly in this section as the other styles.

[1]The code is available at `https://github.com/Banaantje04/CSE-RP-Code-Repo`

3.1.1 Guarded Coinduction

With guarded coinduction, infinite data is defined via copatterns. Instead of pattern matching constructors where the data is used, the data is defined by the possible observations on the data by its destructors [Abe+13].

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

repeat : A → Stream A
repeat f .head = f
repeat f .tail = repeat f

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

Figure 1: An infinite stream modelled as a guarded coinductive datastructure

In Agda guarded coinduction uses coinductive records, where each field in the record signifies a certain observation on the data, with the destructor named the same way as the field. As seen in figure 1, when defining an infinite stream of the same repeating value, the head of the stream is defined separately from the tail using copattern matching. This encoding is productive as the finite element, head, has a finite definition. When observing the tail the infinite stream is only computed one step further, requiring another observation to continue further in the stream. Any subsequent elements can be accessed by observing the tail a certain amount of times until the respective head is reached. At no point is the infinite tail fully evaluated, attempting to fully observe the tail would be prevented by the termination checker. It is not possible to pattern match normally on coinductive records [Tea24b]. Instead, accessing specific fields of a record and then matching on those with, for example, a with-abstraction can be done to distinguish cases based on the data.

Another possibility is two functions recursing to each other instead of to themselves like in figure 2. Productivity is still ensured as both functions are guarded by the other while not calling other functions. This structure is useful when behaviour is different between different levels of recursion.

```
mutual
  repeat : {A : Set} → A → Stream A
  repeat f .head = f
  repeat f .tail = other f

  other : {A : Set} → A → Stream A
  other f .head = f
  other f .tail = repeat f
```

Figure 2: Mutually recursive coinductive functions

3.1.2 Musical Coinduction

Instead of making a coinductive record, musical coinduction works by marking recursive references as being coinductive. It makes use of 'delay' and 'force' operators to convert between a concrete value and a 'delayed' coinductive value [DA10].

```
postulate
  ∞ : (A : Set) → Set
  ♯_ : A → ∞ A
  ♭ : ∞ A → A

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

repeat : A → Stream A
repeat x = x :: ♯ repeat x

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

Figure 3: An infinite stream modelled using musical coinduction

Stream, is now defined as a normal inductive data type, but its second field, the tail of the list, is marked as coinductive with the ∞ symbol. ♭, also known as 'force', converts this coinductive value back to a concrete value when you actually want to use the tail, while ♯, also know as 'delay', converts it to a coinductive value. Delay guards recursive calls, like with repeat and map in figure 3. Guardedness and productivity is ensured similar to guarded coinduction, each successive call can be computed in a finite amount of steps given that the infinite computation is guarded, this time behind the 'delay' operator.

An added benefit of musical coinduction is that, because the data is now modelled as a normal inductive data type, functions can pattern match on it. This makes it easier to write code that behaves differently depending on the data. It is, however, recommended to use guarded coinduction instead musical as musical coinduction is considered to be the old way to do coinduction in Agda [Tea24b].

3.1.3 Sized Types

Encoding a 'size' modelling the depth of a data structure can convince Agda a definition of a data structure is productive. If some data has size or depth n, then the end of the data is reached at least when recursing on the data n times [AP16; VW19]. Agda contains two types called Size and Size< i where i : Size. A j : Size< i means that j is strictly smaller than i. As Size< i evaluates to a Size that is smaller than i, this can be used to set up size relations between different Size types. If a data structure has a i : Size, and its recursive reference a j : Size< i then this ensures that the containing data is always smaller. If then a structure has i = ∞ and j : Size< ∞, then the data can be infinite. A problem with sized types is that they cause consistency issues. Since ∞ : Size< ∞, infinity would have to be smaller than infinity, which is not true.

3.2 Lambda Calculus

Lambda calculus is a mathematical system of functions and their application. It is the foundation of functional languages like Haskell [Hud+07].

Simple untyped lambda calculus, the calculus I looked at, exists of three things [Chu36]:

1. **Functions**: $(\lambda x.E)$ defines a function that takes in a parameter x and has a body expression in which this parameter can be referenced.
2. **Function application**: $L(A)$ applies a function L with the result of A as the value of the function parameter.
3. **Variable references**: x references a variable or parameter with name x, effectively replacing itself with the expression that x was bound to.

An example of an expression is the function const. It takes two parameters and always returns the first one: $\lambda x.\lambda y.x$. Applying this function once: $(\lambda x.\lambda y.x)(10)$ will result in $\lambda y.10$, an expression that will always return 10 no matter the input.

A common addition to lambda calculus is letrecs [Pie02]. The way they work is as follows, they contain a recursive variable and a body. Inside of the body, this variable is available to call and use. The powerful part is that inside of the recursive variable's expression, it is available to itself. This makes making recursive expressions quite straightforward. A simple example is the following:

$$letrec\ b\ r$$

$$b = \lambda y.r(y)$$

$$r = r(r)$$

Where b is the body and r is the recursive variable. The recursive variable calls itself, infinitely looping, and the body calls the recursive variable using an extra parameter it gets. Evaluating this letrec expression would evaluate the body and subsequently the recursive variable once it's referenced.

Another way to do recursion in lambda calculus is using fixed-point operators [Pie02], however, these are built using simple untyped lambda calculus, therefore there are no language structures that I could use coinduction to specifically model those. Therefore I opted for letrecs.

3.3 Evaluation Strategies

When evaluating a language that includes functions, there are different approaches that can be used to evaluate function applications. The most straightforward is call-by-value. With call-by-value, the arguments to the function are evaluated before the rest of the function is evaluated. This means that whenever the arguments to the function are used, they can immediately be substituted in as seen in figure 4.

```
mult : Nat → Nat → Nat
mult x y = x * y
```

```
mult (1 + 2) (3 + 4)
mult 3 (3 + 4)
mult 3 7
3 * 7
21
```

Figure 4: Call-by-value evaluation steps

A downside of this is that arguments are evaluated even if they are not used, leading to wasted work. A solution to that is call-by-name with which arguments are only evaluated when they are used.

```
square : Nat → Nat
square x = x * x
```

```
square (1 + 2)
(1 + 2) * (1 + 2)
3 * (1 + 2)
3 * 3
9
```

Figure 5: Call-by-name evaluation steps

The problem with call-by-name is that because the arguments are evaluated when they are used, if arguments are used multiple times, they are evaluated multiple times as well, like shown in figure 5. This can be solved by call-by-need, lazy evaluation. Call-by-need combines the previous strategies by not evaluating arguments before evaluating the function, but whenever an argument is used, it stores it. This means that subsequent uses of the same argument can immediately use this stored value as seen in figure 6.

```
square : Nat → Nat
square x = x * x
```

```
square (1 + 2)
x * x          -- x = 1 + 2
3 * x          -- x = 3
3 * 3
9
```

Figure 6: Call-by-need valuation steps

The value of x does need to be stored during the execution of square. This is where an environment can be used. It's a list of the variables available to the expression that is currently beingn evaluated. This can either be done by storing the name of the variable, or, in the case of lambda calculus, with de Bruijn indices. With de Bruijn indexing variables indicate which function argument they refer to using a numerical index [WKS22]. This numerical index depends on which function the variable was declared: a lower index means a function that is nested more. For example in the body of the second function in the expression $\lambda x.\lambda y.x$, x would have index 0 while y would have index 1. This can be useful as then variable names do not need to be stored anymore.

4 Encodings

There are three ways I could come up with to model lambda calculus evaluation. One is not using coinduction at all, fully inductive, to use as a reference for the coinductive representations. The other two use coinduction either for modelling variable references or for recursion.

4.1 Inductive

```
data Term : Set where
  TNat : Nat → Term
  TVar : Nat → Term
  Abs : Term → Term
  App : Term → Term → Term
  LRec : Term → Term → Term
```

Figure 7: The inductive datatype modelling lambda calculus expressions

The encoding represents each different term as its own constructor, as can be seen in figure 7. Variable references are modelled using de Bruijn indexing. This means that function abstraction only contains its body. Letrecs contain both the recursive variable they bind and the body in which this variable is available. They do only allow defining a single

variable, however. There are no cycles used in modelling the possible expressions; accessing this recursive variable is done using a 'normal' TVar.

A variation on this type is used for the implemented coinductive encoding as well, with a function to translate from this inductive model to a coinductive one.

4.2 Recursion

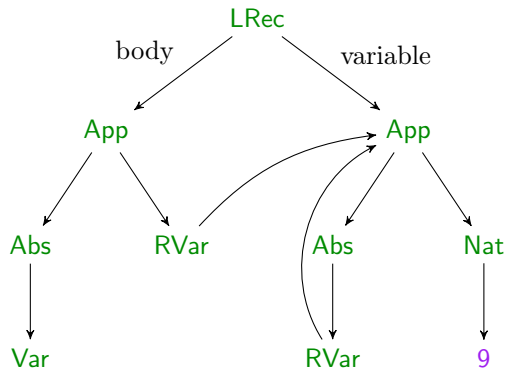

Figure 8: A tree of a lambda expression with a letrec in it where recursive variables are modelled using coinduction

When using coinduction to model recursion, the cycles appear where expressions refer to recursive variables. Both in the body of the letrec and in the actual recursive variable, when a recursive reference is made, this reference points back to the entire variable expression, as can be seen in figure 8.

```
data ITerm : Set where
  ITNat : Nat → ITerm
  ITVar : Nat → ITerm
  ITRVar : RTerm → ITerm
  ITAbs : ITerm → ITerm
  ITApp : ITerm → ITerm → ITerm
  ITLRec : RTerm → ITerm
```

```
record RTerm : Set where
  coinductive
  constructor RTermCtr
  field
    term : ITerm
```

Figure 9: The datatype modelling recursion using coinduction

The difference with this coinductive type and the inductive type is that letrecs (ITLRec in this encoding) only have the body as the parameter. As visible in figure 9, the recursive variable itself is stored and referenced by ITRVar, creating an infinite cycle if the recursive variable contains a recursive reference itself, like in figure 8. Technically, because ITRVar contains the recursion now, ITLRec is not strictly necessary. My implementation does contain it, however. This link is made in the translate function, see appendix A.1. When it encounters a letrec, any recursive references inside of it are constructed with the recursive variable inside. Because this infinite cycle is guarded behind constructors and a copattern, it is deemed productive by Agda so this function terminates.

A small change was made in the inductive Term type compared to the pure inductive encoding to make translating recursion easier. The datatype is a dependent type indexed by booleans indicating whether the current term is inside a recursive expression or not. This makes it easier to link together ITRVar with the variable as it would not allow an unbound variable. This means that it can be assumed that the variable is known wherever an ITRVar appears. This does not allow for nested letrecs though since ITLRec is only defined for Term false. Nested letrecs could possibly be implemented using a list instead as an index on Term but I did not look at that.

While RTerm is written currently using guarded style coinduction, it could also be written using musical coinduction. In that case RTerm would not exist and the parameter to ITRVar and ITLRec would be ∞ ITerm. translate would be using $\sharp$ instead of copatterns to recurse on the variable.

4.3 Variable references

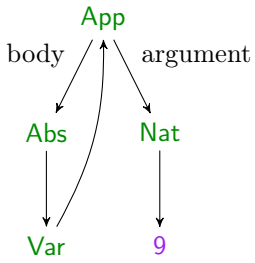

Figure 10: A tree of a lambda expression where variable references are modelled using coinduction

A place where cycles can show up is with variable references. As they refer back to a specific function where they were defined, and subsequently called, this can be used to point to the value of this variable without needing extra structures to keep track of what each variable is supposed to evaluate to. Like visible in figure 10, the variable reference points back to the application where then the argument can be found.

I have not implemented this because of time constraints, however, the way I would build this would be similar to how it was done with recursion: ITVar would contain a coinductive reference using an RTerm in guarded style or ∞ ITerm in musical to refer back to the ITApp where the variable is bound. This way it could get the appropriate variable binding. During translation from inductive to coinductive, the function would keep a list of all the values variables can bind to to choose the appropriate one according to the variable's de Bruijn index, effectively an environment.

I do not expect this encoding to be useful during evaluation compared to inductive evaluation. As letrecs are still defined inductively, evaluation would still not terminate and would therefore not be allowed safely by Agda. It would eliminate the need of an environment storing the values, however. It seems mostly an elegant way to model lambda expressions as is.

5 Evaluating Lambda Expressions

This section outlines the different evaluations for the inductive encoding for lambda expressions and for expressions where recursion is modelled with coinduction.

5.1 Inductive Evaluation

```
eval e (TVar x) = eval e (lookup e x)
eval e (LRec t t₁) = eval (t :: e) t₁
```

```
evalFuel zero _ _ = nat 999
evalFuel (suc n) e (TVar x) = evalFuel n e (lookup e x)
evalFuel (suc n) e (LRec t t₁) = evalFuel n (t :: e) t₁
```

Figure 11: Inductive evaluation implemented non-terminating and by using fuel

Inductive evaluation is pretty straightforward. When it encounters a letrec, it puts the recursive variable in the environment and then evaluates the body, as outlined in figure 11. Similarly, variables are looked up in the environment, and then evaluated seeing how this implementation is call-by-name. Other terms are implemented this way as well. The full implementation can be found in appendix A.2.

eval does not terminate when an infinite loop is present in the lambda expression, therefore this function has to be annotated with NON_TERMINATING to be accepted by Agda. To get a safer implementation that does not require to be annotated, fuel can be added. As seen with the function evalFuel in figure 11, where the fuel is the first parameter, for every recursive call to evalFuel the fuel parameter is decreased. When the fuel hits zero, the function stops, ensuring that the function terminates even if the expression would infinitely loop. Since this is an error, instead of returning a set value like nat 999, the function could instead return an optional where the case where the fuel hits zero nothing is returned.

5.2 Coinduction in Recursion

```
data IVal : Set where
  concrete : Val → IVal
  delay : RVal → IVal
```

```
record RVal : Set where
  coinductive
  field
    rec : IVal
```

Figure 12: Coinductive value

Coinductively evaluating converts the coinductive ITerm to a coinductive IVal, defined as in figure 12. This makes eval terminate even though it works with possibly infinitely looping lambda expressions. Instead of recursing eval normally, which would not be allowed by the termination checker, the recursive call is guarded by the coinductive RVal, effectively delaying the execution of this recursive evaluation until the value is forced sufficiently far. See figure 13. This makes the function productive, the recursive call and creation of the coinductive record is only behind constructors and nothing else, in this case behind the delay constructor. The rest of eval can be found in appendix A.3.

```
eval e (ITRVar x) = delay (mkRVal e (term x))

mkRVal : Env → ITerm → RVal
mkRVal e r .rec = eval e r
```

Figure 13: Coinductively evaluating recursion

A separate function, runIVal, can then convert the coinductive delayed value to a concrete value, see appendix A.3. Similarly to the inductive encoding is this not terminating; it is converting a possibly infinite structure to a finite value. Once again can there be a separate function that takes in a fuel value as well to prevent the function from running forever, also preventing the need for a NON_TERMINATING annotation.

This way of modelling IVal is very similar to how musical coinduction is used. To use musical instead of guarded style coinduction, delay should have ∞ IVal as parameter instead. Calls to term, the field of RTerm, should be replaced with $\flat$. And calls to mkRVal should be replaced with $\sharp$.

eval currently does not have an implementation that passes the termination checker. This is because the case for ITApp is not productive. Applying a function abstraction involves first evaluating the expression that returns an abstraction, and then evaluating its body. This doesn't work because evaluating to an abstraction could take infinite amount of work, after which more work has to be done to be able to evaluate the body. This involves doing a recursive call, evaluating the abstraction, and then calling a non-constructor function, applyFunction, preparing the environment and evaluating the body.

This could be solved by structuring eval differently. In the ITApp case, instead of attempting to evaluate twice, only recurse on evaluating the abstraction while adding the argument to the function in an extra parameter to eval. This way, when an ITAbs is encountered, instead of returning it, its body is immediately evaluated as well. Now no extra function is needed after the recursive call, ensuring productivity.

5.3 Issues with the Evaluation

As lazy evaluation involves keeping state on variables to keep track of whether they have been evaluated once already, this is more challenging than call-by-name. Agda and functional languages do not lend well to this compared to imperative languages. Together with the time pressure, this means the implementations are currently call-by-name.

Another problem that both encoding currently suffer from is an incorrect Var implementation. As I follow a call-by-name evaluation strategy, expressions in variables are only evaluated when they are referenced. Currently variables are evaluated in the current environment, where different variables are bound to the same de Bruijn indices. And might even lead to unintentional infinite loops, as in the lowest lambda expression x would refer to itself instead of y, like it's supposed to.

$$(\lambda y.(\lambda x.x)(y))(0)$$

A possible solution could be to increase the index that is connected to the currently deepest nested function. As the variable has an environment that is a subset of the current environment, only missing the functions that are nested within the function that this variable is declared in, this would make it line up again.

5.4 Tradeoffs of Modelling with Coinduction

The advantages of coinduction are that being able to create an evaluator that is productive even though the lambda expressions put in could infinitely loop. This makes it great to work with in Agda. This safety could in the future be combined with more dependent types to create an evaluator that is correct by construction. An inductive evaluator will need a NON_TERMINATING or TERMINATING clause or fuel to be accepted by Agda which defeats the purpose of using this language.

A coinductive evaluator is not more powerful than an inductive evaluator for the same language. To get a concrete answer out, even the coinductive evaluator could still loop. Not to mention is coinduction foreign to most programmers. I felt it was harder to reason about than inductive programs. Infinities are difficult to wrap my head around. Induction is something that I am familiar with, making it easier for me to write programs with it compared to coinduction. This makes it more challenging to work with a coinductive evaluator.

In conclusion, the gains are mostly in terms of correctness and elegance, at the cost of clarity.

6 Agda Challenges

There are some things I have run into with Agda that I struggled with or I think could be improved upon.

6.1 Unclear Error Messages

Error messages are lacking in Agda. Error messages often refer to intermediate variables not exposed to the user normally, meaning that it is unclear what these variables refer to. For example in the error in figure 14, `_A_21` and `_i_22` are unknown. It would be nicer if a deduction for these intermediate variables was given to show where they come from. What could help as well is a list of relevant bindings; things that are not necessarily involved in the error, but are adjacent to it, which could help with debugging what the problem is.

```
/mnt/Data/Study/TUDelft/BSc Computer Science and Engineering/Y3/Q4/Research
Project/agda-code/report-examples/sized.lagda:39,13-42
Stream _A_21 _i_22 !=< StreamA A i
when checking that the inferred type of an application
  Stream _A_21 _i_22
matches the expected type
  StreamA A i
```

Figure 14: Error message with internal variables unknown to the developer

The termination checker error messages are great on the other hand. They clearly outlay which functions are causing issues, and which calls are problematic. One point of improvement with them is that they do not differentiate functions that don't pass the termination checker because they are not productive and functions that don't pass the termination checker because they call functions that don't terminate. This difference would help with finding the root cause of termination failures, especially with functions that recurse onto each other.

6.2 Error Highlighting

Another problem with errors in Agda is the highlighting in text editors. Too much text is highlighted, even when this text is not relevant at all to the error.

```
eval e (ITApp t a) = case eval e t of λ where
  (concrete (abs t)) → delay (mkRVal (a :: e) t)
  (delay x) → {!   !}
  n → n
```

Figure 15: Error highlighting on the pattern match that is not relevant to the error

In figure 15 the pattern matches on concrete and delay are highlighted in red, for example. The only error in this code is that the code does not terminate. This is a problem with the call to eval, mkRVal, and possibly the lambda abstraction (λ where). When the entire code block is highlighted, it becomes really unclear what is causing the problem.

6.3 Productivity Checking

The productivity checker is very strict. If a recursive call is behind any function call that is not a constructor, it is not accepted by Agda. Even in trivial cases where productivity would not be affected this is the case.

```
id : {A : Set} → A → A
id x = x

repeat : {A : Set} → A → Stream A
repeat f .head = f
repeat f .tail = id (repeat f)
```

Figure 16: Code that fails the productivity checker

Take for example the code in figure 16, repeat is recursively called and then passed through id. Even though id does nothing to the input, and would intuitively not affect productivity, this code is not accepted. While it makes sense that all functions are rejected, it would be nicer in terms of readability in some cases if trivial cases were to be accepted.

Especially because in some cases multiple functions interacting with each other when coinductive recursion is involved is fine. For example the functions in appendix A.1 translateLRec, and mkRTerm recurse to each other, but they are still accepted. I suspect this is the case because mkRTerm is a single copattern match immediately calling translateLRec again which only calls constructors. However, the rules on what is and is not accepted are not clearly documented.

Sized types do not suffer from this as they do not use guardedness for productivity, but they have other issues.

6.4 Lacking Documentation

Most areas of Agda are not documented. There is a documentation website [Tea24a], however, this is only surface level information. Further details are missing; I would like to see perhaps some implementation details, or information about the standard library similar to Haskell's documentation or Java's Javadoc. Often the best way to learn about a certain topic is by reading relevant source code. For details about the Agda standard library this is the only official source of information, even its documentation is a set of Agda source files. This is not a great way to read as the markup is limited to syntax highlighting and monospace font.

7 Conclusion

Coming back to the research questions, coinduction is powerful in ensuring the evaluation function is productive even though it is handling potentially infinite computation. It does however, do this by delaying the problem to a later stage. If an actual value is wanted, the coinductive result eval generates needs to be forced until a concrete value is computed. This process does not terminate. The way this is better than an inductive evaluator is that eval no longer needs any unsafe Agda features like NON_TERMINATING . This comes at the drawback of coinduction generally being harder to reason about.

While the coinductive encoding I looked at but did not implement, cycles in variable references, would probably not help with making the evaluation function terminate, it could help with correctness besides termination as it no longer needs a lookup function to retrieve variables from the environment. Implementing this encoding could be an interesting avenue to explore further with more time.

Future improvements to the existing code could be to implement lazy evaluation, which is currently lacking. This would involve reworking how the environment works and how it is used when evaluating Var. This would fix the current problem with the call-by-name implementation as outlined in section 5.3 as well, seeing how it would not be present if the evaluation strategy is different.

Additionally the eval function for coinductive recursion could be fixed by rewriting the ITApp case so it no longer recurses twice. Instead only recursing on the abstraction and evaluating function bodies when abstraction terms are encountered.

Another thing that could be looked at is implementing letrecs with multiple variables that can reference each other and nesting letrecs. These are things that my implementation currently are not able to do and it could be something that could be interesting to look at.

A potential different encoding that could be looked at is an encoding that is a combination of the current coinductive and inductive models. It would involve an inductive datastructure for the lambda expressions but the evaluation function itself would be coinductive, creating a coinductive value from inductive data. This would keep the advantages of coinduction where this version of eval can be terminating while decreasing the amount of coinductive datastructures involved, making it easier to reason about the code by hand.

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.

[AC14] Andreas Abel and James Chapman. 'Normalization by Evaluation in the Delay Monad: A Case Study for Coinduction via Copatterns and Sized Types'. In: *Electronic Proceedings in Theoretical Computer Science* 153 (5th June 2014). Publisher: Open Publishing Association, pp. 51–67. ISSN: 2075-2180. DOI: 10.4204/eptcs.153.4. URL: http://arxiv.org/abs/1406.2059v1.

[AP16] Andreas Abel and Brigitte Pientka. 'Well-founded recursion with copatterns and sized types'. In: *Journal of Functional Programming* 26 (2016). Publisher: Cambridge University Press (CUP). ISSN: 0956-7968, 1469-7653. DOI: 10.1017/s0956796816000022. URL: https://www.cambridge.org/core/product/identifier/S0956796816000022/type/journal_article.

[Asa19] Kenichi Asai. 'Extracting a call-by-name partial evaluator from a proof of termination'. In: *Proceedings of the 2019 ACM SIGPLAN Workshop on Partial Evaluation and Program Manipulation*. POPL '19: 46th Annual ACM SIGPLAN Symposium on Principles of Programming Languages. Cascais Portugal: ACM, 14th Jan. 2019, pp. 61–67. ISBN: 978-1-4503-6226-9. DOI: 10.1145/3294032.3294084. URL: https://dl.acm.org/doi/10.1145/3294032.3294084.

[Chu36] Alonzo Church. 'An Unsolvable Problem of Elementary Number Theory'. In: *American Journal of Mathematics* 58.2 (Apr. 1936), p. 345. ISSN: 00029327. DOI: 10.2307/2371045. URL: https://www.jstor.org/stable/2371045?origin=crossref (visited on 21/06/2025).

[Coq94] Thierry Coquand. 'Infinite objects in type theory'. In: *Lecture Notes in Computer Science*. ISSN: 0302-9743, 1611-3349. Berlin, Heidelberg: Springer Berlin Heidelberg, 1994, pp. 62–78. ISBN: 978-3-540-58085-0. DOI: 10.1007/3-540-58085-9_72. URL: http://link.springer.com/10.1007/3-540-58085-9_72.

[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: 978-3-642-13321-3. DOI: 10.1007/978-3-642-13321-3_8. URL: http://link.springer.com/10.1007/978-3-642-13321-3_8.

[GJ91] Carsten K. Gomard and Neil D. Jones. 'A partial evaluator for the untyped lambda-calculus'. In: *Journal of Functional Programming* 1.1 (Jan. 1991), pp. 21–69. ISSN: 0956-7968, 1469-7653. DOI: 10.1017/S0956796800000058. URL: https://www.cambridge.org/core/product/identifier/S0956796800000058/type/journal_article.

[Hud+07] Paul Hudak et al. 'A history of Haskell: being lazy with class'. In: *Proceedings of the third ACM SIGPLAN conference on History of programming languages*. HOPL-III '07: ACM SIGPLAN History of Programming Languages Conference III. San Diego California: ACM, 9th June 2007. ISBN: 978-1-59593-766-7. DOI: 10.1145/1238844.1238856. URL: https://dl.acm.org/doi/10.1145/1238844.1238856.

[Pie02] Benjamin C. Pierce. *Types and programming languages*. Cambridge, Mass: MIT Press, 2002. 623 pp. ISBN: 978-0-262-16209-8.

[Tea24a] The Agda Team. *Agda's documentation*. 12th Sept. 2024. URL: https://agda.readthedocs.io/en/v2.7.0.1/index.html (visited on 10/06/2025).

[Tea24b] The Agda Team. *Coinduction*. 12th Sept. 2024. URL: https://agda.readthedocs.io/en/v2.7.0.1/language/coinduction.html (visited on 23/05/2025).

[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.2019.32. URL: https://drops.dagstuhl.de/entities/document/10.4230/LIPIcs.FSCD.2019.32.

[WKS22] Philip Wadler, Wen Kokke and Jeremy G. Siek. *Programming Language Foundations in Agda*. Aug. 2022. URL: https://plfa.inf.ed.ac.uk/22.08/.

A Relevant code fragments

A.1 Translation function

This function translates an inductive Term to a coinductive ITerm modelling recursion using coinduction.

```
mutual
  translateLRec : Term true → Term true → ITerm
  translateLRec r (TNat n) = ITNat n
  translateLRec r (TVar x) = ITVar x
  translateLRec r TRVar = ITRVar (mkRTerm r r)
  translateLRec r (Abs n) = ITAbs (translateLRec r n)
  translateLRec r (App n b) = ITApp (translateLRec r n) (translateLRec r b)

  mkRTerm : Term true → Term true → RTerm
  mkRTerm r b .term = translateLRec r b

translate : Term false → ITerm
translate (TNat n) = ITNat n
translate (TVar x) = ITVar x
translate (Abs n) = ITAbs (translate n)
translate (App n b) = ITApp (translate n) (translate b)
translate (LRec r b) = ITLRec (mkRTerm r b)
```

A.2 Inductive evaluation

This function evaluates inductive Term values using NON_TERMINATING to successfully compile even though it does not terminate.

```
{-# NON_TERMINATING #-}
eval : Env → Term → Val
eval e (TNat x) = nat x
eval e (TVar x) = eval e (lookup e x)
eval e (Abs t) = abs t e
eval e (App t t₁) = case eval e t of λ where
  (nat x) → nat 999
  (abs x e₁) → eval (t₁ :: e) x
eval e (LRec t t₁) = eval (t :: e) t₁
```

This function evaluates inductive Term values using fuel to always terminate.

```
evalFuel : Nat → Env → Term → Val
evalFuel zero _ _ = nat 999
evalFuel (suc n) e (TNat x) = nat x
evalFuel (suc n) e (TVar x) = evalFuel n e (lookup e x)
evalFuel (suc n) e (Abs t) = abs t e
evalFuel (suc n) e (App t t₁) = case evalFuel n e t of λ where
```

```
  (nat x) → nat 999
  (abs x e₁) → evalFuel n (t₁ :: e) x
evalFuel (suc n) e (LRec t t₁) = evalFuel n (t :: e) t₁
```

A.3 Coinductive evaluation

This function evaluates coinductive ITerm values to a coinductive IVal. Keep in mind that this function, in its current state, fails the termination checker because of the ITApp case.

```
mutual
  eval : Env → ITerm → IVal
  eval e (ITNat x) = concrete (nat x)
  eval e (ITVar x) = delay (mkRVal e (lookup e x))
  eval e (ITRVar x) = delay (mkRVal e (term x))
  eval e (ITAbs t) = concrete (abs t e)
  eval e (ITApp t a) = delay applyFunR
    where
      applyFunR : RVal
      applyFunR .rec = applyFunction (delay (mkRVal e t)) a
  eval e (ITLRec x) = delay (mkRVal e (term x))

  mkRVal : Env → ITerm → RVal
  mkRVal e r .rec = eval e r

  applyFunction : IVal → ITerm → IVal
  applyFunction (concrete (abs t e₂)) a = delay (mkRVal (a :: e₂) t)
  applyFunction (delay x) a = delay applyFunctionC
    where
      applyFunctionC : RVal
      applyFunctionC .rec = applyFunction (rec x) a
  applyFunction n a = concrete (nat 999)
```

This function turns a coinductive IVal into a concrete Val. It does not terminate therefore it needs to be annotated with NON_TERMINATING .

```
{-# NON_TERMINATING #-}
runIVal : IVal → Val
runIVal (concrete x) = x
runIVal (delay x) = runIVal (rec x)
```

This function turns a coinductive IVal into a concrete Val using fuel to always terminate.

```
runIValFuel : Nat → IVal → Val
runIValFuel zero _ = nat 999
runIValFuel (suc f) (concrete x) = x
runIValFuel (suc f) (delay x) = runIValFuel f (rec x)
```