Practical Verification of Infinite Structures in AGDA2HS

Remco Schrijver
Supervisors: Jesper Cockx, Lucas Escot
EEMCS, Delft University of Technology, The Netherlands

June 19, 2022

A Dissertation Submitted to EEMCS faculty Delft University of Technology,
In Partial Fulfilment of the Requirements
For the Bachelor of Computer Science and Engineering

Practical Verification of Infinite Structures in AGDA2HS

Author: Remco Schrijver
Delft University of Technology
Netherlands

Supervisor: Jesper Cockx
Delft University of Technology
Netherlands

Supervisor: Lucas Escot
Delft University of Technology
Netherlands

ABSTRACT

Agda allows for writing code that can be mathematically proven and verified to be correct, this type of languages is generally known as a proof assistant. The AGDA2HS library makes an effort to translate Agda to readable Haskell, in a way the Haskell is still consistent. In previous work it is shown that with the current AGDA2HS implementation, rudimentary structures can be translated to Haskell from Agda with AGDA2HS. In this paper the translation and verification of infinite structures to readable Haskell code is researched. This allows for future work to be done on verification of more complex libraries because the concept of infinite structures is used often in Haskell. The results of the research were that translation of rudimentary infinite structures is possible, but functions creating infinite structures cannot be translated at this point in time.

KEYWORDS

functional programming, Agda, Haskell, AGDA2HS, infinity, coinduction, copatterns

1 INTRODUCTION

Using unit and/or integration tests is important to improve stability and verify correct behaviour of software. However, in safety critical systems there is a more formal method of verification and testing of systems compared to normal unit/integration testing. This takes the shape of verification and validations (V&V) plans laid out during testing, and contingency/safety planning for after deployment of the system[Joung et al. 2009].
These guidelines for V&V ensure the logic of the system is sound, making it possible to translate that logic to concrete test cases in unit testing. However, there are limitations to unit/integration testing, it is impossible to exhaust the set of inputs and outputs for most systems under test. This and other limitations can be negated in the form of static analysis, code coverage, and mutation testing. But on top of that engineers can use languages with proof assistants to aid them in verifying systems.

One of these languages with a proof assistant is Agda. A proof assistant, in contrast to unit testing, allows you to be certain that the system is correct for the whole space of possible inputs and outputs. The V&V guidelines can be translated to a mathematical proof and if those are correct, the system is also correct. Resulting in safer systems and allows developers to be more confident in their work.
However, the small number of libraries and frameworks implemented in Agda holds it back in adoption of real-world systems. One way this weak spot could be relieved is translating the code to readable Haskell. This allows engineers to take advantage of the broad ecosystem of frameworks and libraries provided to Haskell programmers. But with the added benefit of the guaranteed correctness provided by proofs written in Agda. This is where AGDA2HS comes in, it aims to find a subset between Agda and Haskell, which translates Agda code to readable Haskell.
AGDA2HS is not a complete subset yet, one of many areas that is not researched yet is infinite structures. Common usage of these in Haskell are in list generators, list comprehension, GUIs, and Streams. If AGDA2HS allows for the use infinite structures, it would be possible to verify more complex libraries. The goal in this paper is to find the limitations of AGDA2HS regarding translation of infinite structures. And if it is possible to translate these structures and their operations from Agda to Haskell, that the properties remain unaffected by the translation.

Infinite structures are used often in Haskell, without this support AGDA2HS would be less intuitive to use by Haskell programmers. With this in mind I have researched the possibility of implementing infinite structures in AGDA2HS and came up with the following research question: *Is it possible for agda2hs to translate infinite and cyclic structures that rely on co-induction in Agda to the concept of infinite structures used in Haskell, and if not can agda2hs be extended to be able to allow for this translation?*

And produced the following sub questions:

- Is it possible to translate the type of infinite lists with agda2hs from Agda to Haskell?
- Is it possible to translate operations on the type of infinite lists with agda2hs from Agda to Haskell?
- Is there a difference in implementation between the type of cyclic and infinite structures in the translation with agda2hs from Agda to Haskell?
- Does the verification of properties on the type of infinite lists stemming from co-induction remain consistent after the translation with agda2hs from Agda to Haskell?
- Does the verification of properties on operations on the type of infinite lists stemming from co-induction remain consistent after the translation with agda2hs from Agda to Haskell?

In the end in turns out that infinite structures are part of the subset of Agda and Haskell. Where infinite structures with functions that do not produce or alter infinite structures can be translated by the current AGDA2HS version. But complex functions that create or mutate infinite structures cannot be translated due to copattern translation being incorrect.

The structure of the paper is as follow, in section 2 a quick introduction to Agda and AGDA2HS is made in the preliminaries. Then in section 3 the related works to this paper are discussed, this mainly centres around hs-to-coq and LiquidHaskell. After that I get into the method of how this research is done, described in section 4. Section 5, the problem analysis goes into the coinduction, copatterns, and

sized types. Then in section 6 the results from my research are discussed. Section 7 goes into the responsible research, and section 8 is the discussion where I reflect on the process. Then lastly there still are section 9 which makes recommendations for future research and section 10 is the conclusion of this paper.

2 PRELIMINARIES

For those that are not acquainted with Agda or AGDA2HS a small summary of their functioning and their use will follow in this section.

2.1 Agda

Proof assistants allow developers to write proofs for the code they write, one of these is called Agda. Using these proof assistants, you can ensure at compile time that for example a program will terminate or if it is productive. This has the benefit that more errors get caught at compile time instead of at run time, which makes debugging easier and programs more stable. How Agda does this is by using dependent types, it is developed mostly by academics and also sees most of its use in academics.

2.2 AGDA2HS

Agda on its own has a small user base and it being mostly academic, the developer community and library support does not come close to languages like Java, C#, or Haskell. This makes it hard for developers wanting to use the proof assistant capabilities of Agda while making functional products without reinventing the wheel. This is where AGDA2HS comes in, it tries to find the subset of Agda and Haskell and translates this to readable Haskell. This allows for verified code checked by Agda to be used in Haskell projects. But a drawback is that some parts of Agda might not be able to be translated as it is not part of this subset between Agda and Haskell.

3 RELATED WORKS

The proposition of using coinduction for infinite and circular structures (also called non-well-founded sets in set theory) was introduced with the anti-foundation axiom[Aczel 1988]. Or more aptly described in [Sangiorgi and Rutten 2011, p . 17] as "axioms of anti-foundation lead to the largest possible universe, i.e. a 'coinductive universe'". This use of coinduction has further ended up in languages like Agda to denote infinite structures and perform verification on these structures[Agda 2021].
I also researched comparable libraries to AGDA2HS that try to extend Haskell such that it completely or partially can be proven mathematically correct, while still generating readable code. I have found two of these that have significant adaptation and support. Namely hs-to-coq[hs-to coq 2021] that translates Haskell to the proof assistant language Coq[Coq 2022]. And LiquidHaskell[LiquidHaskell 2022b] which uses logical predicates that allow you to ensure properties at compile time[LiquidHaskell 2022a].

Both hs-to-coq and LiquidHaskell have their own way of dealing with infinite (or non-terminating) structures, first hs-to-coq will be explained and then LiquidHaskell.

The way hs-to-coq handles infinite structures is by not allowing them[Spector-Zabusky et al. 2018]. The part of Haskell that was problematic for their translation to coq was "As a consequence of Haskell's lazy evaluation, Haskell data types are inherently coinductive."[Spector-Zabusky et al. 2018, p. 10]
This is the crux of the problem, Haskell has a less stringent structure in the form of always being coinductive, as opposed to Coq with a more stringent handling of coinduction that requires an annotation for coinductive structures. The developers argue that the most used Haskell code either works with finite structures or can be rewritten in a way it is finite. This can be debated, because it might be less intuitive to write Haskell code without infinite structures like list generators or list comprehension. But in the light of the alternative, that all translated types can no longer use functions like length. It was the reasonable decision to make for the project.
However, this is not a problem for AGDA2HS because instead of going from a less stringent way of handling coinduction to a more stringent one, the opposite happens. So, this trade-off between either infinite structures or loosing out on some functions is not on the table.

On the other hand, there is LiquidHaskell, this system uses refinement types[Vazou et al. 2014]. Which is different from the dependent types used by Agda, meaning they are hard to compare. And the implementation LiquidHaskell used for infinite structures is not applicable for a solution for AGDA2HS.
As mentioned LiquidHaskell uses refinement types, which in simple terms can explained as pre-conditions and post-conditions that are required to hold for the functions annotated. This has as an upside that it is easier to write these, but you will lose out on versatility in general when using refinement types. But it is interesting to note that infinity is still usable under refinement types. Although this has no real consequence for AGDA2HS and cannot be used as a lead for further implementation.

4 METHOD

Using AGDA2HS to try to create infinite structures in Haskell from Agda through translation with AGDA2HS is subdivided into multiple research questions, the method for answering them is as follows:

(1) Implement the most basic coinductive structure in Agda, most likely an infinite list, and verify it translates correctly to Haskell.
(2) On this most basic structure implement some basic functions like take and drop and verify correct translations.
(3) Now on this most basic structure we will implement more involved functions that require copatterns to implement like map. As well as creating constructors for infinite lists like repeat, or the Fibonacci sequence.
(4) Implement a more involved coinductive structure with sized types like the delay monad[Abel and Chapman 2014], and verify the translation.

For the implementation we were provided by Lucas Escot with a basic code base where we could start development off from[1]. I went on to analyse the main facets of the translation of infinite structures

[1]Verification template provided by Lucas Escot: https://github.com/flupe/verification-template

done by AGDA2HS. More on that in the next Section 5, the problem analysis.

5 PROBLEM ANALYSIS

During my research I identified three interesting sub-fields that are critical for translation by AGDA2HS for infinite structures. First off, coinduction which is necessary for the infinite data types in Agda. Secondly, copatterns these are for advanced functions that return infinite structures themselves in Agda. And lastly sized types, this is an alternative to creating structures and handling advanced functions in Agda and aids the productivity checker. In the coming subsections these are explained more in depth.

5.1 Coinduction

To write infinite structures in Agda[2] the developers picked coinduction[Sangiorgi and Rutten 2011]. Coinduction is centred around the idea of a co-space of induction, hence the name. Where in induction we have the base case and the inductive case, coinduction only has a coinductive step. This creates a recursive infinite structure that uses the previous step to define the following and repeats this to infinity. Coinductive structures are defined by their destructors[Kozen 2017], a classic example is the Stream type. In Agda it is notated with the record syntax, and Stream would look something like seen in listing 1. The field of *hd* allows you to access the element at the current index of the stream, the *tl* field allows you to iterate over the structure.

Listing 1: Example of the Stream record type defined in Agda.

```
record Stream (a : Set) : Set where
    coinductive
    field
        hd : a
        tl : Stream a
```

5.2 Copatterns

However, coinduction in Agda comes with the downside that defining functions that create or modify infinite structures, i.e., repeat or map, requires a different approach due to the productivity checker. Copatterns[Andreas et al. 2013] allow for pattern matching on structures defined coinductively within Agda and creating these advanced functions. This works because using copatterns in Agda make it no longer check on productivity but just on termination[Abel and Chapman 2014].
A simple case of copatterns in Agda would be the repeat function, as seen in listing 2, where an input 'a' is given which is then repeated infinitely many times in the structure of Stream. As you can see, we defined for repeat what we should do when we observe the hd, we return x in this case. And when we observe tl we just call repeat x again, in this way continuing the Stream.

Listing 2: Example of the repeat function on the Stream record type in Agda.

```
repeat : {a : Set} (x : a) → Stream a
hd (repeat x) = x
tl (repeat x) = repeat x
```

5.3 Sized Types

Sized types are special data types in Agda that helps in guaranteeing productivity for coinductive structures in Agda[Veltri and van der Weide 2019]. This is done by annotating the types with a sized field, this shows the maximum amount of unfolds that are possible. This notation makes defining functions on coinductive structures less complex and more intuitive.

6 RESULTS

Based on my research on previous works, described in Section 3. I discovered the fact that Haskell is implicitly coinductive[Spector-Zabusky et al. 2018]. And Haskell has support for record types, the way Agda defines coinductive structures. Both of these facts gave me reason to expect the translation of these coinductive record types to be possible. Before getting into the written code, for those that want complete access to files that the examples are taken from, these can be found here[3].

I wrote a rather basic infinite structure called *InfiniteList* that can be seen in listing 3. This translates properly to Haskell and can be used to instantiate infinite structures in Haskell. Some examples would be all list of primes or the Fibonacci sequence. But without some basic functions we cannot do anything with these structures. Also note that right now I cannot proof anything about the instantiated structures because they are not instantiated in Agda.

Listing 3: The definition of *InfiniteList* in Agda.

```
record InfiniteList (a : Set) : Set where
    coinductive
    field
        hd : a
        tl : InfiniteList a
{-# COMPILE AGDA2HS InfiniteList #-}
```

The actual result of conversion done by AGDA2HS can be seen in listing 4. And is a rather natural translation that I expected and can be used properly as an infinite structure.

Listing 4: The definition of *InfiniteList* in Haskell after translation by AGDA2HS.

```
data InfiniteList a = InfiniteList{
    hd :: a,
    tl :: InfiniteList a
}
```

This *InfiniteList* by itself is not very useful, for it to become useful we need functions for it. Basic functions are straightforward to implement. Like *takeInf* which takes the n first values of a list. The function *dropInf* which drops the first n values of a list, or *!!!* which takes index n of an infinite list. The results of implementing these can be seen in listing 5.

[2]Documentation page of Agda for coinduction: https://agda.readthedocs.io/en/latest/language/coinduction.html

[3]Code base for the discussed examples: https://github.com/RemcoSchrijver/verification-of-infinite-structures/tree/paper_reference

Listing 5: Definition of basic functions, *takeInf*, *dropInf*, and *!!!* in Agda.

```
takeInf : {a : Set} -> InfiniteList a →
    Nat → List a
takeInf list Zero = []
takeInf list (Suc n) = (hdInf list) ::
    (takeInf (tlInf list) n)
{-# COMPILE AGDA2HS takeInf #-}

dropInf : {a : Set} → InfiniteList a →
    Nat → InfiniteList a
dropInf list Zero = list
dropInf list (Suc n) =
    dropInf (tlInf list) n
{-# COMPILE AGDA2HS dropInf #-}

_!!!_ : {a : Set} → InfiniteList a →
    Nat → a
list !!! Zero = hdInf list
list !!! Suc n = (tlInf list) !!! n
{-# COMPILE AGDA2HS _!!!_ #-}
```

The translation also works nicely and conform to what I expected, this can be seen in listing 6. The use of *Nat* as opposed to *Integer* is a bit unfortunate but this was the easiest way for now to implement this in Agda. During translation this Nat type is translated with it as well. This means you lose out on the efficiency of the int datatype in Haskell so using *Integer* would be necessary in the future to optimise the translation.

Listing 6: Conversion of basic functions, *takeInf*, *dropInf*, and *!!!* in Haskell after translation by AGDA2HS.

```
takeInf :: InfiniteList a ->
    Nat -> [a]
takeInf list Zero = []
takeInf list (Suc n) =
    hdInf list : takeInf (tlInf list) n

dropInf :: InfiniteList a ->
    Nat -> InfiniteList a
dropInf list Zero = list
dropInf list (Suc n) =
    dropInf (tlInf list) n

(!!!) :: InfiniteList a -> Nat -> a
list !!! Zero = hdInf list
list !!! Suc n = tlInf list !!! n
```

Verifying these basic functions can be seen in listing 7, and shows the proofs that are necessary for proving bisimilarity. This means that using different operations of functions results in the same outcome. Because these proofs do not mean anything in Haskell these are not translated.

Listing 7: Proving of bisimilarity of basic functions

```
drop-head-bissimular-tail-head : ∀ {a}
    (list : InfiniteList a) →
    hdInf(dropInf (Suc Zero) list) ≡
        hdInf(tlInf list)
drop-head-bissimular-tail-head _ = refl

tail-head-bissimular-index :∀
     {a} (list : InfiniteList a) →
    hdInf(tlInf (tlInf (tlInf (tlInf list)))) ≡
     list !!! Suc (Suc (Suc (Suc Zero)))
tail-head-bissimular-index _ = refl
```

As I already explained in the problem analysis, for more advanced functions on this *InfiniteList* copatterns are necessary. You can think of instantiating the structures or the map method that is used a lot in functional programming. Two examples I wrote are the *fibonacci* and *evenInf* functions, these can be seen in listing 8.

Listing 8: Definition of the advanced functions, *fibonacci* and *evenInf* in Agda.

```
fibonacci : Nat → Nat → InfiniteList Nat
hd (fibonacci n1 n2) = n1
tl (fibonacci n1 n2)
  = (fibonacci (n2) (n1 +++ n2))
{-# COMPILE AGDA2HS fibonacci #-}

evenInf : {a : Set} → InfiniteList a →
    InfiniteList a
hd (evenInf xs) = hd xs
tl (evenInf xs) = evenInf (tl (tl xs))
{-# COMPILE AGDA2HS evenInf #-}
```

These do not translate as expected, as can be seen in listing 9. The functions *fibonacci* and *evenInf* translates to invalid Haskell, this is being looked at right now within the AGDA2HS library[4]. For now, it is impossible to use these types of functions that require copatterns.

Listing 9: Conversion of the advanced functions, *fibonacci* and *evenInf* in Haskell done by AGDA2HS

```
fibonacci :: Nat -> Nat -> InfiniteList Nat
fibonacci n1 n2
    Data.InfiniteList.InfiniteList.hd
  = n1
fibonacci n1 n2
    Data.InfiniteList.InfiniteList.tl
  = fibonacci n2 (n1 +++ n2)

evenInf :: InfiniteList a -> InfiniteList a
evenInf xs Data.InfiniteList.InfiniteList.hd =
    hd xs
evenInf xs Data.InfiniteList.InfiniteList.tl =
    evenInf (tl (tl xs))
```

[4]GitHub Issue for AGDA2HS on copatterns: https://github.com/agda/agda2hs/issues/98

Using copatterns and the more higher order functions, bisimilarity can be proven on *mergeInf* and *splitInf* functions[5]. These either merge two lists into a singular infinite list by taking first one element from the first and then from the second list. And split does the opposite creating two infinite lists from one by first taking one element of the parent list and then taking the other splitting it in two. So, to prove that using merge on split results in an identical list the following code can be used as seen in listing 10. Please note that helper structures are defined for ≈ and for this proof translation to Haskell is not necessary.

Listing 10: Bisimilarity proof in Agda on mergeInf and splitInf

```
merge-split-id : ∀ {a} (list : InfiniteList a)→
      mergeInf (split list) ≈ list
hd≡- (merge-split-id _) = refl
tl≈- (merge-split-id list)
  = merge-split-id (tl list)
```

Because copatterns do not translate properly with AGDA2HS, I looked to sized types. Sized types are a different way to write coinductive types and help the productivity checker of Agda. Because of this it allows us to write advanced functions differently from normal coinductive types. The definition of a comparable *InfiniteList* with use size types called *CoList* can be seen in listing 11 and the supporting Thunk structure can be seen as well.

Listing 11: *CoList* and *Thunk* in Agda

```
data CoList (a : Set) (@0 i : Size) : Set where
  Nil  : CoList a i
  _:::_ : a → Thunk (CoList a) i → CoList a i
{-# COMPILE AGDA2HS CoList #-}

record Thunk (a : @0 Size → Set)
    (@0 i : Size) : Set where
  coinductive
  field
    force : {@0 j : Size< i} → a j
open Thunk public
{-# COMPILE AGDA2HS Thunk #-}
```

Two important parts that are not implemented in translation, but are being worked on at the moment[6]. Are first, the size field can be removed in all cases. This means that you do not need to use erasure annotations for it anymore and removal is done automatically. Secondly the *Thunk* record defined is not necessary in Haskell, thunks are handled implicitly by Haskell by being a lazy language[7]. The translation that takes place right now can be seen in listing 12

Listing 12: *CoList* and *Thunk* in Haskell translated by AGDA2HS

```
data CoList a = Nil
              | (:::) a (Thunk (CoList a))
data Thunk a = Thunk{force :: a}
```

Using sized types the expectations were that defining more complex functions like *repeatCoList* would translate properly as opposed to copatterns. How repeat would be defined can be seen in listing 13.

Listing 13: *repeatCoList* in Agda

```
repeatCoList : {a : Set} {@0 i : Size} →
    a → CoList a i
repeatCoList x = x ::: λ( where .force →
    repeatCoList x)
{-# COMPILE AGDA2HS repeatCoList #-}
```

However as can be seen in listing 14 the expectations are proven wrong. A somewhat similar problems as seen in listing 9 around copatterns show up. This might have something to do with the fact that when the *force* functions are used on the left-hand side it is still a copattern[Abel and Chapman 2014]. Although this notation is not used in this example it might well be the reason why the same outcome as with normal coinductive types takes place. Trying to construct the record type inside of the *repeatCoList* function also was fruitless because then the productivity checker gave problems. So also sized types do not make it possible to define more advanced functions on infinite structures.

Listing 14: *repeatCoList* in Haskell translated by AGDA2HS

```
repeatCoList :: a -> CoList a
repeatCoList x
  = x :::
      \case
        Data.Thunk.Thunk.force ->
          repeatCoList x
```

7 RESPONSIBLE RESEARCH

In this paper it is possible for others to follow the same process I followed writing the code to answer my research questions. Besides that, all the code written is hosted on GitHub[8] and can be freely accessed by anyone. Besides that, anyone is free to clone and install it following the steps in the README file provided in the repository to run the examples themselves. The usage of a make file ensures that if the correct dependencies are installed, the code can be run cross-platform. The dependencies and their versions are fully described in the README file, but the important ones are Agda, Cabal, and GHC. These can be installed via their respective installers.

During the research no ethical issues arose, no data of humans was used, and it was fully focused on the theoretical aspects programming languages and mathematics. As far as was possible I tried to write correct code and think soundly when making conclusions following from my research. Therefor this research is free from malicious intent and all faults can be attributed to mistakes made during this research.

[5]Proof of bisimilarity on merge and split: https://agda.readthedocs.io/en/v2.6.2.1/language/coinduction.html
[6]GitHub issue for AGDA2HS on intuitive translation of sized types: https://github.com/agda/agda2hs/issues/99
[7]Haskell documentation on Thunks: https://wiki.haskell.org/Thunk
[8]GitHub Repository for this paper: https://github.com/RemcoSchrijver/verification-of-infinite-structures/tree/paper_reference_1

8 DISCUSSION

Although the first part of my research went rather flawlessly, I stumbled upon a solid wall in the form of non-supported functionality in AGDA2HS. This is the translation of copatterns, this is required to create functions that can create or modify coinductive record types in Agda.
But during my research I felt that my progress was slow. This might be caused by the fact that the topics of coinduction, copatterns, sized types, and infinity are rather intricate, and it took me a long time to grasp them.
But when I finally understood coinduction and its application in Agda, I encountered copatterns and sized types which threw me off again. So, I think I progressed slowly but, in the end, achieved some useful results. I was able to clear up what is and is not possible with AGDA2HS regarding infinite structures. This allowed for clear targets on what to implement for AGDA2HS to actual allow for programming with these structures.

Another struggle I encountered was the fact that most knowledge is distributed within papers, examples on GitHub, and the developers of AGDA2HS, but not defined in documentation. This made my progress rather slow, it required me to ask a lot of question about basic functions. A user manual as defined for Agda would be a great improvement. On the other hand AGDA2HS, when comfortable with using Agda is intuitive to use. But it does expose a weakness in AGDA2HS, the fact that using it is not well documented or supported for now. But if not for the missing translation in AGDA2HS the project feels mature.

The verification template provided by Lucas Escot was unreliable when used on Windows, but future users should have less problems due to the fixes introduced by me and Lucas[9]. In the end it did not take a lot of time to make improvements and have the template working reliably. However, if I were a developer, I would expect that downloading AGDA2HS would also come with an environment in which I could create my projects. Similar to the role that the verification template plays in this case.

So, in summary AGDA2HS is a project that still has work to do, but the features that are in there are stable and very usable.

9 FUTURE RESEARCH

There are still quite some topics that can be researched in this part of the AGDA2HS library on the topic of infinite structures, and these are the following:

- One sub question I asked was, if there would be any difference between infinite and circular structures Because although circular structures are not strictly infinite, it is possible to infinitely traverse the structure. The termination checker of Agda might complain about this, but the translation of the structure to Haskell should be rather natural. Although functions could be more involved as seen with the problems with copatterns.
- Right now the infinite structures I defined uses the record type with a coinduction step in Agda. However, in Haskell list comprehension and list generation is an important part of Haskell that uses infinite structures. And it would not be intuitive for Haskell developers to use coinductive structures for these features.
  Now using the *map*, *filter*, and *bind* functions list comprehension could be constructed. However, the research also should go into if filter can actually be used because it is not productive. For list generation the *iterate* function will need to be defined, this to me on the surface seems easier to implement. Now using the fact that Agda allows for almost all characters to be function names, both list generation and comprehension could even be defined the same way as in Haskell.

[9]Fixed windows issues with template in PR: https://github.com/flupe/verification-template/pull/1

10 CONCLUSION

The results I have found from my research is that, due to the implicit nature of coinduction in Haskell and the support of record syntax. It is possible to translate coinductive records in Agda to Haskell with AGDA2HS as seen in listing 4. But support of functions that can create or mutate infinite structures is not available with AGDA2HS yet. This is because copatterns are not translated properly and creates invalid Haskell code as seen in listing 9. Support for this has to be added in AGDA2HS to create meaningful infinite structures in Agda. And using sized types to try and achieve the same behaviour as these functions with copatterns failed, as seen in listing 14. This translation of these functions with sized types generates similar faults in the translation as with copatterns. This alludes to the possibility that it fails because it actually does use a similar structure as copatterns do.

ACKNOWLEDGMENTS

I would like to thank my supervisors Jesper Cockx and Lucas Escot for all the support during this research project. Especially helping out when I was lost trying to use Agda and offering solutions to problems I encountered during writing and coding. Also, I want to extend my thanks to my fellow students Alex Haršáni, Luka Janjić, Marnix Massar, and Michelle Schifferstein which during this project provided peer feedback, reading my works, and delivered valuable insights into what I could improve. And lastly a big thanks to Andreas Abel for allowing and helping me to add to the Agda documentation, and I hope it will help others researching comparable topics.

REFERENCES

Andreas Abel and James Chapman. 2014. Normalization by Evaluation in the Delay Monad: A Case Study for Coinduction via Copatterns and Sized Types. *Electronic Proceedings in Theoretical Computer Science* 153 (jun 2014), 51–67. https://doi.org/10.4204/eptcs.153.4

Peter Aczel. 1988. *Non-Well-Founded Sets.* Palo Alto, CA, USA: Csli Lecture Notes.

Developers Agda. 2021. Coinduction in the Agda documentation V2.6.2.1. https://agda.readthedocs.io/en/v2.6.2.1/language/coinduction.html [Accessed: 09-05-2022].

Abel Andreas, Pientka Brigitte, Thibodeau David, and Setzer Anton. 2013. Copatterns Programming Infinite Structures by Observations. *Conference Record of the Annual ACM Symposium on Principles of Programming Languages* 48, 27–38. https://doi.org/10.1145/2480359.2429075

Developers Coq. 2022. Coq language website. https://coq.inria.fr/ [Accessed: 09-05-2022].

Developers hs-to coq. 2021. hs-to-coq Github page. https://github.com/plclub/hs-to-coq [Accessed: 09-05-2022].

E. J. Joung, C. M. Lee, H. M. Lee, and G. D. Kim. 2009. Software safety criteria and application procedure for the safety critical railway system. In *2009 Transmission Distribution Conference Exposition: Asia and Pacific*. 1–4. https://doi.org/10.1109/TD-ASIA.2009.5356897

Silva-Alexandra Kozen, Dexter. 2017. Practical coinduction. *Mathematical Structures in Computer Science* 27, 7 (2017), 1132–1152. https://doi.org/10.1017/S0960129515000493

Developers LiquidHaskell. 2022a. LiquidHaskell documentation website. https://ucsd-progsys.github.io/liquidhaskell/ [Accessed: 09-05-2022].

Developers LiquidHaskell. 2022b. LiquidHaskell Github page. https://github.com/ucsd-progsys/liquidhaskell [Accessed: 09-05-2022].

Davide Sangiorgi and Jan Rutten (Eds.). 2011. *Advanced Topics in Bisimulation and Coinduction*. Cambridge University Press. https://doi.org/10.1017/CBO9780511792588

Antal Spector-Zabusky, Joachim Breitner, Christine Rizkallah, and Stephanie Weirich. 2018. Total Haskell is Reasonable Coq. In *Proceedings of the 7th ACM SIGPLAN International Conference on Certified Programs and Proofs* (Los Angeles, CA, USA) *(CPP 2018)*. Association for Computing Machinery, New York, NY, USA, 14–27. https://doi.org/10.1145/3167092

Niki Vazou, Eric L. Seidel, Ranjit Jhala, Dimitrios Vytiniotis, and Simon Peyton-Jones. 2014. Refinement Types for Haskell *(ICFP '14)*. Association for Computing Machinery, New York, NY, USA, 269–282. https://doi.org/10.1145/2628136.2628161

Niccolò Veltri and Niels van der Weide. 2019. Guarded Recursion in Agda via Sized Types. In *4th International Conference on Formal Structures for Computation and Deduction (FSCD 2019) (Leibniz International Proceedings in Informatics (LIPIcs), Vol. 131)*, Herman Geuvers (Ed.). Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik, Dagstuhl, Germany, 32:1–32:19. https://doi.org/10.4230/LIPIcs.FSCD.2019.32