Practical Verification of a Free Monad Instance

Luka Janjić
Supervisor(s): 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

Abstract

Formal verification of software is a largely underrepresented discipline in practice. While it is not the most accessible topic, efforts are made to bridge the gap between theory and practice. One tool conceived for this exact purpose is AGDA2HS, a tool intended to allow developers to create their programs correct-by-design. A program written a proof assistant language Agda, along with the proof of its correctness, can be translated to the readable Haskell equivalent, retaining only the functionally relevant aspects and leaving the proof related aspects of the code behind. This paper describes research done into the current abilities of AGDA2HS on the use case of verifying properties of free monads, a higher order type with potential for use in implementation of domain specific languages and purely functional and modular handling of effects. In order to represent an aspect of the type in a general way I used containers, a uniform way of representing types that store data. This led me to a limitation of AGDA2HS: while the tool in its current state can only handle direct translations from a common subset of the two languages, in order to translate my definition of the data type I needed a more fine grained control of the translation process which the tool could not provide.

1 Introduction

A naive approach to software verification usually comes down to verifying that a certain combination of inputs to the program results in the expected outputs. A vast majority of modern programs, however, have such a large amount of possible input-output combinations that the traditional software testing methods are unable to provide any strong guarantees on their correctness. For this reason, when more concrete guarantees are required, another approach might become more feasible: formally proving properties of programs. Indeed this approach might require a greater extent of intellectual effort, yet given the right tool set it could prove itself paramount in use cases such as compiler construction [1], where software failure is not an option. While the formal methods show a lot of promise, there are still some challenges on their way to widespread use.

A range of different tools providing formal verification exist [2] and this project is focused on the Agda language along with a tool for translating Agda code into Haskell called AGDA2HS[1]. The main goal of this research is investigating how and what properties can be stated and formally verified for a small subset of the Haskell ecosystem relating to free monads [3], using Agda and AGDA2H. The first step is identifying and selecting instances of free monads as starting point of the research. These instances are then formalized in Agda in such a way that AGDA2HS translates them back to an equivalent of Haskell code that we started with. Next, in order to provide formal verification, invariants of these structures are identified and stated in Agda. Once the code under investigation and its invariants are stated in Agda, the formal verification is completed by providing relevant proofs.

In the context of functional languages, monads are a versatile and powerful tool for handling *impure computations* in a purely functional way. A canonical example in Haskell is its IO monad, which allows programs to interact with the outside world and handle related effects without jeopardizing the functional purity of the language, thereby retaining the benefits that this provides. Nevertheless, since the IO monad wraps such a wide range of effects, it becomes impossible to determine what exactly a specific instance is capturing. Free monads have a potential to allow for a more fine grained structuring of such monolithic monads on the type level [3]. Furthermore, due to their tree-like structure, they can be a natural way to embed domain specific languages into the underlying language which can conveniently be written using Haskell's do-notation. Since constructed expressions would in fact be just data, this allows for separation of syntax and semantics definitions, as the interpretation would be defined independently of the structure.

In practice, however, vast majority of programs depend on existing libraries. The code in these libraries, assuming it is correct, adheres to a certain specification that may be expressed as a series of claims involving the functions and the data types defined within it. If a developer using certain features of these libraries wanted to prove correctness of their own code, they would preferably do so by

[1] `https://github.com/agda/agda2hs`

assuming the expected properties of the code used. However if these properties were merely conjectured to hold without a proper proof, the resulting proof of correctness would provide little guarantee that the desired properties hold. Hence, to facilitate formal verification of software in practice, it is useful to have proofs of such known properties for existing library code at disposal. These proofs can then be used as lemmas in proving correctness of programs using those libraries. Since I am working with a monadic structure, a natural choice of properties to prove are the *monad laws*, which must hold for any valid monad.

2 The Gap

The main obstacle in providing practical formal verification is the gap between the languages used in practice for development and the tools used for verification. This research project is part of the effort towards bridging this gap. On one hand we have Haskell, a general purpose, purely functional programming language with a rich ecosystem of libraries ready to be used by developers in their endeavor of solving whatever problem they might be facing. This language, in all of its expressive power, is simply not suited for addressing the issue of formally verifying the code that is produced within it. On the other hand we have Agda, a dependently typed language with a type system powerful enough to facilitate formal proofs of arbitrary statements, yet still providing the functionalities of a typical functional programming language. Now one might ask why don't we simply develop our software in a language such as Agda? While we in fact could do so, that would not be very practical since Agda's capabilites as a theorem prover introduce significant sytactic and performance overhead that is not necessarily desired in a language intended for general purpose development. The proposed solution to bridging this gap is using a tool, called AGDA2HS, that can take an Agda file and translate specified parts of the Agda code into the equivalent Haskell. The idea is as follows: a code base is being developed using Haskell. There are some critical segments of the program that require formal verification. Such a segment can in turn be written in Agda, along with formal proofs of its functional correctness. Once the critical part has been expressed and verified, AGDA2HS can be used to automatically convert only the code relevant to the execution of the program to readable Haskell.

2.1 Free monad

In this project I explore such a work flow on an existing part of the Haskell library, specifically that involving the free monad data type. Generally speaking, a monad [4] is a datatype that defines two operations: `return` (also called unit) and `>>=` (called bind), such that they obey three invariants – the monad laws.

```
        # Right Identity #
return a >>= f  ===  f a

        # Left Identity #
 m >>= return  ===  m

        # Associativity #
(m >>= f) >>= g  ===  m >>= (\x -> f x >>= g)
```

Figure 1: The monad laws

A *free monad* is a higher order datatype parameterized by two type variables such that when applied, the resulting type has a valid monadic structure. The first parameter is a single variable type constructor `f` endowed with a functor structure [2] and the second one some concrete type `a`. In Haskell

[2] `https://wiki.haskell.org/Typeclassopedia#Functor`

terms, these two parameters are said to have a kind `f :: * -> *` and `a :: *` respectively. The specific Haskell data type that I have chosen to work with is `Free f a`, which is defined for some functor `f` and type `a` as follows:

```
data Free f a = Pure a
              | Free (f (Free f a))
```

Figure 2: Haskell definition of the Free datatype

Namely, a value of type `Free f a` can either be instantiated using a `Pure` constructor with a value of the type `a`, or using the `Free` constructor with a value of the type `f (Free f a)`. This can be seen as defining a tree structure, where `Pure` constructors instantiate leaves, and `Free` constructors instantiate the internal nodes. It should be noted, however, that the internal nodes do not necessarily represent only the branching but can have additional structure determined by the functor with which they are constructed.

2.2 Agda

Agda is the input language to AGDA2HS. While purely functional, Agda has certain further restrictions and features that enable it to operate as a proof assistant: it is a *total*, *purely functional* and *dependently typed* programming language.

The **purely functional** aspect is the most straight forward of the three. It means that in Agda there is no distinction between values and functions and, moreover, all functions are pure: given the same input a function will always yield the same output. These two statements capture the essence of purely functional languages. A useful consequence of this is the fact that equational reasoning about programs becomes natural and transparent; any function call can be replaced by its definition with the arguments substituted in and the behavior of the program is guaranteed to remain unchanged.

While most programming languages allow for defining nonterminating computations fairly easily, Agda restricts valid definitions exclusively to those that it can prove to be terminating, making it a **total** language. It does so by checking that every recursive call is made on a *syntactically smaller* value compared to the one passed as an argument to the function: given some arguments to a function, it may only recurse on composite elements of the provided arguments, thereby guaranteeing that any recursive program written in Agda will reach a base case and yield a value in finite time.

```
valid : Nat → Nat
valid zero     = zero
valid (suc n-1) = valid n-1
```

Figure 3: A valid recursive call

```
invalid : Nat → Nat
invalid zero = zero
invalid n    = invalid n
```

Figure 4: An invalid recursive call

The third important aspect lies in Agda's type system, which is modelled after an intuitionistic type theory developed by Per Martin-Löf [5]. In addition to defining regular algebraic data types (ADT's) encountered across many functional languages, Agda's type system allows for **dependent types**. These are parameterized types, similar to polymorphic ones, with the key difference being the parametrization over *values* of some type instead of only types themselves. As a simple example, consider the type `Fin n` of natural numbers up to some n. This type depends on a value of type `Nat`, the natural numbers, and each instance of it is populated with n values, corresponding to all the naturals up to the given n.

```
data Nat : Set where
  zero : Nat
  suc  : Nat → Nat
```

Figure 5: Defintion of Nat

```
data Fin : Nat → Set where
    fzero : {n : Nat} → Fin (suc n)
    fsuc  : {n : Nat} → Fin n → Fin (suc n)
```

Figure 6: Definition of a dependent type Fin

A consequence of these properties is the logical consistency of Agda's type system. It encodes a constructive logic system in which proofs of claims are given by construction from smaller terms until trivial truth is reached. Under the Curry-Howard correspondence, we can relate each Agda type to a logical proposition and each program of the given type to a proof of that proposition. Thus a proposition represented by some type is true if and only if this type is populated and, equivalently, if we can show a type to be empty then we have shown the corresponding claim to be false. This allows us to use Agda not only as a programming language but also as a proof assistant. While Agda by default guarantees that the aforementioned properties hold and hence that the typechecked proofs are correct, it allows its user to loosen any of the restrictions for some segment of the code, trading off its rigor for ease of implementation. Indeed in such a case we must be careful not to trick Agda into violating its own rules, for this would allow us to prove contradictory statements and render Agda inconsistent.

Note that in Agda types are themselves also values and, as such, must belong to a type. We call this type `Set`. Hence, the newly defined data types and any type variables appearing within their definitions must be annotated as elements of `Set`, corresponding to the Haskell kind $*$. Since `Set` is also a value it must itself have a type. However, if we are to preserve the logical consistency of the type system, no value can not be its own type. This issue is circumvented by the introduction of an infinite hierarchy of type *levels* (e.g. `Set n` contains level n types), such that `Set` is a synonym for `Set 0` and `Set n : Set (n + 1)` for $n \in \mathbb{N}$. Agda provides a module `Level` that exposes the type of the same name used to represent the level values.

2.3 Strict Positivity and Containers

One of the ways in which Agda's rules could be violated relates to defining inductive data types. Namely, Agda places a restriction on valid definitions of data types: strict positivity. Types of arguments to a constructor of some data type D must either be noninductive (they do not mention D at all), or have D only in the last position (in the type of the return value). In other words, D must not occur to the left of an arrow in the type of any of its constructors' arguments. Without this restriction we could seamlessly violate the totality of Agda and, consequently, the soundness of Agda's type system. Take as an example the following Agda definitions, for an arbitrary type T:

```
data NonPositive : Set where
    bad : (NonPositive → T) → NonPositive

apply-self : NonPositive → NonPositive
apply-self (bad f) = f (bad f)

break-agda : T
break-agda = apply-self (bad apply-self)
```

Figure 7: Non Strictly Positive Definition

Notice that if we attempt to evaluate `apply (bad apply)`, the expression expands into itself; it is a *non-terminating computation*. Worse yet, if we take T to be the empty type then the above example constructs an element of the empty type and thereby *proves a contradiction*.

Now recall the Haskell definition of `Free`. We can easily express the same in Agda:

```
data Free (F : Set → Set) (A : Set) : Set where
    pure : A → Free F A
    free : F (Free F A) → Free F A
```

Figure 8: Naive definition of Free data type in Agda

While it is not as immediate as in the previous example, this definition is not strictly positive. Namely, the issue can arise if `F` evaluates to a function that takes as an argument a value of the type given as an argument to `F`. This would cause the type of `free` to be of the form:

```
(T1 -> ... -> Free F A -> ... -> Tn) -> Free F A
```

violating the strict positvity condition. The verbatim translation of the Haskell definition of `Free` is not valid in Agda. Since the goal is to verify the data type for the most general case I have turned to *containers* [6]: a uniform way to represent data types that store values and, conveniently, all strictly positive types [7]. Generally speaking, a container is a higher order type determined by a shape type `S : Set ℓs` and a family of types `P : Set ℓp` indexed by `S`, denoted as `S ▷ P : Container ℓs ℓp`. These two types determine what structure the container represents. Since I am using the containers for purposes of modeling types in Haskell and in Haskell there are no higher, the two level arguments will be set to zero so I define the corresponding type `Container00`. The *extension* of a container is a type constructor `⟦ S ▷ P ⟧ X = Σ[ s ∈ S ] (P s → X)`, a dependent pair with a shape value `s : S` and a function from possible positions `P s` (its domain *depends* on the shape `s`) to the contained values of some type `X`. The intuition behind this is that the type `S` represents the collection of all possible shapes that a given structure can take and that `P` yields all the possible positions `P s` in a given shape, which we can use to index into the structure with the function in the second position of our dependent pair.

2.4 Equational reasoning in Agda

The proofs in this paper are constructed using a simple *equational reasoning* framework in Agda. The syntax is made to allude to typical mathematical proofs by algebraic rewriting so the proofs are relatively easy to read. Each proof begins by the expression `begin` and ends with `end`. Trivial equality is expressed by `=⟨⟩` , representing that Agda can show the two statements to be equal by computing their normal form. In other words both expressions reduce to the same value just by applying all the constituent functions. This, in fact, needs not be stated for a valid proof to be accepted by Agda, but is used for readability. On the other hand `=⟨ some-argument ⟩` is used when an additional argument needs to be used for Agda to accept the equality assertion. In this case the expression can be seen as a transitivity statement, where the argument in the brackets is the bridging equality statement between the two surrounding expressions. Finally, `cong` is an often used function in equational reasoning proofs and it represents congruence.

3 Verification for Free?

The final translation of the `Free` data type restricts the first parameter to a container and requires the argument to the `free` constructor to be an extension of the given container, preserving the strict positivity condition and resulting in the following definition: This choice of representation had several

```
data Free (F : Container00) (A : Set) : Set where
    pure : A → Free F A
    free : ⟦ F ⟧ (Free F A) → Free F A
```

Figure 9: Final definition of Free data type

consequences on the outcome of my research. While it allowed me to circumvent the strict positivity issue, it had an additional benefit of enabling me to subsequently write general proofs applying to any kind of positive functor. On the other hand it has pushed me to the verge of capabilities of AGDA2HS, as I have found that this kind of representation, while useful, is beyond what can be handled by the tool. This formulation can not be translated back to equivalent Haskell code. Due to the requirement that any monad must be an applicative and a functor, I had to translate those definitions first before giving the instance of Monad for `Free`. This was a straight forward task that involved nothing more but adapting the original Haskell definitions in accordance the syntactical differences of the two languages.

```
iMonadFree ._>>=_ (pure a) f = f a
iMonadFree ._>>=_ (free m) f = free (fmap (_>>= f) m)
```

Figure 10: Definition of bind for the monad instance of Free in Agda

```
Pure a >>= f = f a
Free m >>= f = Free ((>>= f) <$> m)
```

Figure 11: Definition of bind for the monad instance of Free in Haskell

Despite reaching the limits of AGDA2HS, I have provided proofs of the monad laws for `Free F A` given any positive funtor `F` representable by the container and any contained type `A`. Left identity required the least effort, as it can be shown directly from the definitions of `return` and `>>=` simply by applying the functions.

```
monad-left-id : {F : Container00} {A B : Set}
              → (a : A) → (f : A → Free F B) → (return a >>= f) ≡ f a
monad-left-id a f = refl
```

Figure 12: Proof of the left identity law

For the remaining two proofs mere substitution was not enough. Since the container extension abstracts over representable types using a function to yield the contained values and the monad laws are equality statements, I had to use the axiom of *function extensionality* in order to complete them. This axiom states that, given two functions of the same type, if they evaluate to the same value for any given input then they are equal. While the axiom is not part of the underlying logic system, it is known to be consistent with it, so postulating it does not affect the validity of the proofs.

```
extensionality : {A B : Set} {f g : A → B}
               → (∀ x → f x ≡ g x) → f ≡ g
```

Figure 13: Function extensionality

The remaining two proofs follow a similar inductive structure where the base case holds trivially from the definitions, while the recursive case consists of reducing the binds to `fmap` by substituting the definition, further rewriting the resulting expression within the `free` constructor, then using extensionality to reason about the effect on the contained values, until finally reaching the very expression under investigation involving those very contained values. At this point all that is left to do is tie the recursive knot by referring to the theorem itself as the induction hypothesis.

```
-- the induction hypothesis
monad-right-id : {F : Container00} {A : Set} → (m : Free F A) → m >>= return ≡ m

bind-into-pure-is-id : {F : Container00} {A : Set}
                       ((s , vs) : ⟦ F ⟧ (Free F A)) → (_>>= return)  vs ≡ vs
bind-into-pure-is-id c@(_ , vs) =
    begin
       -- from definition of composition
        (λ p → vs p >>= pure)
       -- by the induction hypothesis applyied to the contained value (vs p)
    =⟨ extensionality (λ p → monad-right-id (vs p)) ⟩
        vs
    end

fmap-bind-into-return-is-id : {F : Container00} {A : Set}
                              (fa : ⟦ F ⟧ (Free F A)) → fmap (_>>= return) fa ≡ fa
fmap-bind-into-return-is-id fa@(s , vs) =
    begin
       -- since return is defined as pure I write it direcly
        fmap (_>>= pure) fa
    =⟨⟩ -- applying the definition of fmap for containers
        (s , (_>>= pure)  vs)
       -- by the above lemma applied to the second position of the pair
    =⟨ cong (s ,_) (bind-into-pure-is-id fa) ⟩
        fa
    end

monad-right-id (pure x)  = refl
monad-right-id (free fa) =
    begin
       -- from applying the bind and return
        free (fmap (_>>= pure) fa)
       -- now referring to the above lemma w.r.t. the expression within the constructor
    =⟨ cong free $ fmap-bind-into-return-is-id fa ⟩
        free fa
    end
```

Figure 14: Proof of the right identity law

```
-- the induction hypothesis
monad-assoc : {F : Container00} {A B C : Set}
              (m : Free F A) (g : A → Free F B) (h : B → Free F C)
            → (m >>= g) >>= h ≡ m >>= (λ x → g x >>= h)

ext-lemma : {F : Container00} {A B C : Set}
            (g : A → Free F B) (h : B → Free F C) ((_ , vs) : ⟦ F ⟧ (Free F A))
          → ∀ x → (λ p → ((vs p) >>= g) >>= h) x ≡ (λ p → (vs p) >>= (λ x → (g x) >>= h)) x
ext-lemma g h fa@(_ , vs) p =
    begin
        ((vs p) >>= g) >>= h
       -- by induction hypothesis on the constituent terms
    =⟨ monad-assoc (vs p) g h ⟩
        (vs p) >>= (λ x → (g x) >>= h)
    end

fmap-bind-lemma : {F : Container00} {A B C : Set}
                  (fa : ⟦ F ⟧ (Free F A)) (g : A → Free F B) (h : B → Free F C)
                → fmap (_>>= h) (fmap (_>>= g) fa) ≡ fmap (_>>= (λ x → (g x) >>= h)) fa
fmap-bind-lemma fa@(s , vs) g h =
    begin
       -- directly from the definition of fmap
        (s , λ p → (vs p >>= g) >>= h)
       -- by the extensionality argument applied to the value yielding function
    =⟨ cong (s ,_) (extensionality (ext-lemma g h fa)) ⟩
        (s , λ p → (vs p) >>= (λ x → g x >>= h))
    =⟨⟩ --by unapplying the definition of fmap
        fmap (_>>= (λ x → g x >>= h)) fa
    end

monad-assoc (pure x) g h = refl
monad-assoc m@(free fa) g h =
    begin
        (free (fmap (_>>= g) fa)) >>= h
    =⟨⟩ -- reducing the fmap
        free (fmap (_>>= h) (fmap (_>>= g) fa))
       -- applying the above lemma to the argument expression to free
    =⟨ cong free (fmap-bind-lemma fa g h) ⟩
        free (fmap (_>>= (λ x → g x >>= h)) fa)
    =⟨⟩ -- unapplying the >>=
        (free fa) >>= (λ x → g x >>= h)
    end
```

Figure 15: Proof of the associativity law

4 Responsible Research

The work done in this research project was entirely based on Agda language which, as explained in the second chapter, is in itself a consistent logical system providing strong guarantees for the typechecked code. The results outlined by this paper can be easily reproduced by accessing the

public repository containing the final version of the code[3]. Most of the relevant code is presented by the figures throughout the paper, which is given without any semantically relevant changes (some comments and small rearrangements were added for readability) and it can be compared with the code from the repository to assure that it was presented faithfully. The proofs that I have created were typechecked, guaranteeing their correctness. This can be easily verified using an open source Agda distribution accessible from the official GitHub repository. There are no other relevant ethical issues directly relating to the topic of this research.

5 Related Work

A related effort towards writing proofs involving Free monads using containers was done by Dylus et. al. [8]. In their research they used Coq proof assistant, which is based on a similar type theory as Agda's type system. They faced the same issue regarding strict positivity and had solved it using containers. Their research, however, was of a more general nature, as they explored the possibilities of establishing a framework for reasoning more generally about monadic structures using free monads as a representation of a wide range of different monads.

An alternative approach to verification to the one used in this paper takes the opposite direction: starting with an existing Haskell code, using a tool such as HsToCoq [4] to translate it into a proof assistant language such as and then proving properties on the translated code. While this has an advantage of allowing its user to verify existing code without need to port it to the host language first, which is error prone and time consuming, it is less convenient when correct-by-design code is to be produced. The code translated to the proof assistant language often has to be augmented in order to facilitate proof writing so the process becomes cumbersome if the code has to be written in one language, then augmented and verified in another. For such a workflow it is more appropriate to develop and verify everything in the same environment and have automated translation to the deployment language.

6 Conclusion and Future Work

While the verification process demonstrated in this project was successful, the main goal of this project was exploring the possibilities of AGDA2HS. In this regard I have identified a limitation of the tool. The final implementation of the `Free` data type, using containers to represent strictly positive functors, can not be translated to the desired Haskell code. The tool in its current state is only able to perform direct translations from the common subset of the two languages, while my situation required more nuanced behavior. Namely, the container used to represent functors is supposed to be translated to a regular type constructor variable, and the instantiation of its extension in one of the constructors should be replaced by just the variable itself. This could be a useful feature to implement in the future as it could be used in similar verification efforts where some argument could be replaced by a different representation, more suitable for formal verification. For example, an additional annotation could be included for marking such arguments to types or functions, thereby allowing users to provide a segment of Agda code whose translation should replace the marked argument. Since some additional functions might have to be applied to the argument in the definition of the data type or function under verification, such as the container extension function in case of this research, there should also be a way to state the "activation function" that should be ignored during translation, if there is any.

It is interesting to note that, while the proofs given here hold, their reliance on the postulated extensionality axiom could be avoided. In general, postulates are avoided whenever possible because they result in weaker computational properties [10]. Since the postulate is simply given as an axiomatic statement, which corresponds to a term of some type without a reduction rule, any computation relying

[3] `https://github.com/sourceCode4/verification-of-free-monads/releases/tag/v1.0.0`
[4] `https://github.com/antalsz/hs-to-coq`

on it can not be fully evaluated. This could be avoided by reformulating the proofs in terms of Cubical Agda [11], in which function extensionality can be proven without additional postulates.

References

[1] X. Leroy, (2009). Formal verification of a realistic compiler. Communications of the ACM, 52(7), 107-115. https://doi.org/10.1145/1538788.1538814

[2] R. C. Armstrong, et al. "Survey of Existing Tools for Formal Verification," Sandia National Laboratories, Albuquerque, NM, and Livermore, CA, SAND2014-20533, Dec. 2014. Accessed on 11.06.2022. [Online]. Available: `https://web.archive.org/web/20180713134613id_/http://prod.sandia.gov:80/techlib/access-control.cgi/2014/1420533.pdf`

[3] W. Swierstra, 2008. Data types à la carte. Journal of Functional Programming, 18(04).

[4] E. Moggi, 1991. Notions of computation and monads. Information and Computation, 93(1), pp.55-92.

[5] P. Martin-Lof and G. Sambin, Intuitionistic type theory. Napoli: Bibliopolis, 1984.

[6] M. Abbott, T. Altenkirch, and N. Ghani, "Categories of containers," Lecture Notes in Computer Science, pp. 23-38, Feb. 2003.

[7] M. Abbott, T. Altenkirch and N. Ghani, "Containers: Constructing strictly positive types", Theoretical Computer Science, vol. 342, no. 1, pp. 3-27, 2005. Available: 10.1016/j.tcs.2005.06.002.

[8] S. Dylus, J. Christiansen and F. Teegen, "One Monad to Prove Them All", The Art, Science, and Engineering of Programming, vol. 3, no. 3, 2019. Available: 10.22152/programming-journal.org/2019/3/8 [Accessed 15 June 2022].

[9] J. Voigtländer, "Asymptotic Improvement of Computations over Free Monads", in MPC: International Conference on Mathematics of Program Construction, Marseille, France, 2008.

[10] L. Pujet and N. Tabareau, "Observational equality: now for good", Proceedings of the ACM on Programming Languages, vol. 6, no., pp. 1-27, 2022. Available: 10.1145/3498693 [Accessed 19 June 2022].

[11] A. Vezzosi, A. Mortberg and A. Abel, "Cubical Agda: A dependently typed programming language with univalence and higher inductive types", Journal of Functional Programming, vol. 31, 2021. Available: 10.1017/s0956796821000034 [Accessed 19 June 2022].