BRINGING FORMAL VERIFICATION INTO WIDESPREAD PROGRAMMING LANGUAGE ECOSYSTEMS

Sára Juhošová

Bringing Formal Verification into Widespread Programming Language Ecosystems

Sára Juhošová

to obtain the degree of Master of Science
in Computer Science
at the Delft University of Technology

to be defended publicly on
Thursday 29 June 2023 at 9:00

Student number:	4906330	
Thesis committee:	Dr. Christoph Lofi	*thesis advisor*
	Dr. Jesper Cockx	*daily supervisor*
	Lucas Escot	*daily co-supervisor*

Preface

This thesis is the result of five years of studies at the TU Delft. I have had a wonderful time throughout the entire journey and I cannot imagine a better-suited study for myself. The topic of this thesis combines many of the things I love about computer science, including writing beautiful code, diving into the confusingly interesting world of programming languages, and helping others understand how to use a tool that I find interesting.

I would like to thank my supervisors, Jesper Cockx and Lucas Escot, who always found the time to help me and who motivated and encouraged me to share the work I did. I would also like to thank Christoph Lofi, who took the time to lead small discussions with me whenever I knocked on his door. Those discussions greatly influenced the design and flow of this thesis and led me to think more deeply about the purpose of this thesis.

I would also like to thank Jelco Köhlenberg, Wouter Polet, and Marko Matušovič for being there throughout my thesis journey and calling coffee breaks when I needed to clear my mind. My thanks also goes to Cédric Willekens and Jonathan Brouwer, who were great study companions and kept me on track throughout all my master's courses, to Hanka Jirovská, who was the best project teammate, and to Ruben Backx, who was always ready to help me with anything. Finally, and perhaps most importantly, I would like to thank my family for all the support they gave me throughout my educational career.

Sára Juhošová
Delft, June 2023

Abstract

Formal verification is a powerful tool for ensuring program correctness but is often hard to learn to use and has not yet spread into the commercial world. This thesis focuses on finding an easy-to-use solution to make formal verification available in popular programming language ecosystems. We propose a solution where users can write code in an interactive theorem prover and then transpile it into a more popular programming language. We use AGDA2HS as a case study to determine what challenges users find in using such a tool, improve selected features, and then conduct a user study to evaluate the usability. We find that detailed documentation, support for commonly-used features in the target programming language, features that facilitate verification, integration of the tool into the target ecosystem, and user studies are necessary for the accessibility of such a tool.

keywords: formal verification, theorem proving, programming language ecosystems, transpilation, usability

Contents

Chapter 1

Introduction

As software in all forms creeps into every part of the modern world, "the developer community no longer needs to argue about the importance of software testing" [1]. Making sure that your software is *correct* has become increasingly more important with every financial, personal, and societal catastrophe caused by a tiny bug in someone's code. Many techniques, tools, and strategies for software testing have been developed and are widely used in the industry, allowing us to have higher confidence in the programs we write and use. Unfortunately, as Dijkstra once famously said, "program testing can be used to show the presence of bugs, but never to show their absence".

Luckily, there is something we can use to prove that our programs are 100% bug-free, i.e. *correct*: formal verification. Formal verification is the process of mathematically *proving* that our programs behave the way they were specified to. There are many methods and techniques of formal verification, including model-checking [2] and abstract interpretation [3], but in this thesis, we focus on interactive theorem provers (or "proof assistants"). This is an approach where computers and human users "work together interactively to produce a formal proof" [4], essentially allowing users to reason about their programs and use the computer to verify their specifications. Examples of existing interactive theorem provers include the Agda [5], Coq [6], and Idris [7] programming languages.

While interactive theorem provers seem like the perfect solution, "the relative complexity of the theories underlying theorem provers makes them inaccessible to a wide range of software engineers who are not experienced mathematicians" [8]. This is naturally also reflected in the theorem provers themselves, creating steep learning curves for programmers trying to use them. Because theorem provers are so inaccessible, their respective user communities are not big enough to create an ecosystem that eases their use. This means that there are no beginner-friendly libraries, frameworks, or even question forums

to ease the learning curve of the developers. If a language does not allow developers to create the things they want, the language becomes a lot less attractive.

The truth is, not every part of the code base needs to be so rigorously verified. Tests are much cheaper and easier to write and can often achieve the desired level of confidence. This means that maybe the answer is not to create an ecosystem for a complex formal verification language. After all, that kind of complex thinking is not needed for the entire code base. Maybe it is enough to bring formal verification into the existing ecosystems and make it just another tool in the arsenal.

1.1 Problem Statement

To understand the problem addressed in this thesis, we recall and refine three concepts introduced earlier:

1. Formal verification is a powerful tool for ensuring program correctness and it allows us to have a high confidence in the code we write.
2. Existing interactive theorem provers are often hard (to learn) to use and thus do not have big communities contributing to their ecosystems.
3. Developers want to use languages that have all the tools they might need available in their ecosystems.

All of these together create the unfortunate situation in which we have powerful software verification tools already available, yet unable to spread into the commercial world. This is the problem that we focus on in this thesis, by finding a way to bring formal verification into existing, widespread ecosystems. We use the following research question to address it:

How can we make formal program verification accessible within ecosystems of widespread programming languages?

To better understand the choice of this question, let us look at the word "accessible". In its simplest form, it tells us that we want to make formal verification *available* within widespread programming languages. There are already many ways in which this has been done and we can draw from previous work. However, we also aim to interpret accessible in its other form: we want to make the solution easy to understand and use.

1.2 Methodology & Contributions

We use the following steps to answer the research question defined above:

1. We examine previous work in this field and identify a tool that allows us to reap the benefits of both formal verification and a large programming language ecosystem: AGDA2HS [9]. We compare the suitability of the idea behind the tool with other approaches in the field.
2. We explore the challenges that come with using AGDA2HS by analysing previous work and identifying issues in the tool's repository. This step also includes analysing the situations in which these challenges occur and studying the solutions and work-arounds that have been used to get past them.
3. We form a hypothesis for our research question based on the challenges identified earlier and select several features we want to implement in the hopes of making AGDA2HS more usable.
4. We identify and implement solutions for the selected features.
5. We evaluate the validity of our hypothesis by conducting a small-scale user study where users are asked to use and reflect on the improved version of AGDA2HS.

In the process of answering the research question, we provide the following contributions:

- We present solutions to and implementations for selected challenges as contributions to the online public repository of AGDA2HS. We also create and improve the documentation of these features on the AGDA2HS documentation website.
- We describe the design and execution of a user study focused on the usability of AGDA2HS, and provide recommendations for any consecutive user studies.
- We present the findings from the user study and identify usability requirements for formal verification tools.

1.3 Thesis Overview

To explain the storyline of this thesis, we provide an overview of its chapters. In Chapter 2, we look into related work and search for existing solutions to

making formal verification available in widespread programming language ecosystems. During Chapter 3, we explore the current challenges with using AGDA2HS, a tool chosen as a case study, and from them formulate a hypothesis for our research question. We continue by explaining the selection of certain AGDA2HS features to improve based on the hypothesis in Chapter 4 and present the solutions and implementations of those improvements. We take Chapter 5 to focus on the purpose and setup of the user study, defining helper research questions and explaining the goal of each component. In Chapter 6, we present the results of the user study and answer the helper research questions. During Chapter 7, we discuss the results and provide recommendations for future work. Finally, we conclude the thesis in Chapter 8 by reflecting on what we learned in the previous chapters to answer the main research question.

Chapter 2

Related Work

In order to find a way to make formal verification *available* in widespread programming language ecosystems, we look towards previous work and existing solutions. Most of the tools discussed here have the approach of transpiling[1] between a language with a widespread ecosystem and a proof assistant, thereby giving to formal verification in the form of a theorem prover. We discuss them below based on which direction they transpile in and whether the transpilation is automated or manual. We also discuss Liquid Haskell [10], a framework which allows developers to verify Haskell code *in Haskell*.

Manual Transpilation to a Proof Assistant

In the past decades, there have been multiple works whose aim was to verify a library in a widespread programming language by porting its source code to a proof assistant and proving certain properties about it. One of the earliest works was done by Dybjer et al. [11], who ported Haskell programs to Agda/Alfa and used a combination of techniques to verify their correctness. They recognised the value of other tools in the Haskell ecosystem, and they implemented a version of QuickCheck [12] in Agda/Alfa to be the first step of verification before any expensive proofs were written. Similarly, in a recent paper by Carr et al. [13], the authors focus on verifying their Haskell library for Byzantine Fault Tolerant (BFT) consensus protocols by also porting it to Agda. There has also been work on using other proof assistants to verify Haskell code, such as by Christiansen et al. [14], who reason about effectful Haskell programs in monadic Coq.

A benefit of this approach is that it might seem more natural to developers

[1] Transpilation is the process of translating a program from one language into another language with a similar level of abstraction.

working in the widespread ecosystems, since they start out by writing their programs in exactly the same way as they always do. Even the manual transpilation to a new language might become easier due to this, especially if we are dealing with languages such as Haskell and Agda, which share many design decisions and features.

On the other hand, these approaches all transpile the programs manually, so the two code bases need to be maintained side-by-side. To address this issue, Carr et al. write that in the majority of cases, "[the Agda code] mirrors the Haskell code so closely that side-by-side review requires virtually no mental overhead in our experience" [13]. Despite that, this is still a major drawback, since the code being verified is not actually linked to the code being run and errors will appear on a case-by-case basis.

Additionally, verifying code *after* it has been written severely limits the capabilities of interactive theorem provers since powerful techniques such as intrinsic verification, where proofs become part of the code itself and thus make the programs *inherently* safe, are not accessible to the developer. As Allais noted in his recent work, using only a subset of such powerful languages as Agda or Coq can lead to "awkward encodings when the unrefined inputs force the user to consider cases which ought to be impossible" [15]. He claims that such an approach "completely misses the point of type-driven development", since the extra information available during interactive editing can provide many advantages.

Finally, it is important to consider that many of the libraries being verified rely on functions and types from other libraries. Consider, for example, higher-order functions such as `map` and `filter` which are an indispensable part of any functional programming language. This means that the proof assistant used for verification must have equivalent library functions available, in order to make the transpilation even possible. Additionally, there are certain properties and assumptions that should hold about them in Haskell, and those might be important for proving properties of the programs that use them.

Automatic Transpilation to a Proof Assistant

A step that can solve many of the drawbacks of the approaches in the section above is to make the transpilation process automated. This problem was addressed by Abel et al. [16], who present an approach to automatically transpile Haskell programs into Agda. The GHC compiler is utilised to translate the Haskell code into Haskell Core (a System F-like subset of Haskell) and then a "monadic translation" is applied to transpile the code to Agda. The monadic translation allows for a representation of total programs using the

Identity monad as well as partial programs (which are not supported in Agda) using the Maybe monad.

Similarly, `hs-to-coq` is a tool for automatically transpiling Haskell code to the Coq proof assistant. This tool also uses the GHC compiler to parse and rename the written Haskell code, but stops before the type checking and desugaring phase to transpile the program to Coq. Unlike with Agda, Haskell actually has many structural and design differences with Coq, and the bulk of the work on `hs-to-coq` was figuring out how to overcome them.

A significant challenge with automated transpilation is to make sure that the generated code is actually readable by the user. This is important, because while the user can trust that "[the process] does translate programs to a faithful representation of their semantics" [16], they might still have a hard time reasoning about and verifying an unreadable implementation. Readability is also why `hs-to-coq` (unlike the approach by Abel et al.) does the transpilation before any of the Haskell code is desugared, even though "translating [to] GHC's intermediate language, Core, would certainly simplify the translation" [17].

With automated transpilation, we avoid many of the drawbacks seen in the solutions where the translation had to be done manually. Most significantly, it is no longer necessary to maintain two versions of the code side-by-side — a huge advantage, especially if the codebase that needs to be verified is big or often changing. Additionally, since the translation is automatic and thus less expensive, existing libraries that are often reused in code can be obtained for free by simply running them through the transpiler. This also applied to features such as derived instances of type classes (which we address later in this thesis), which are translated by "simply [taking] the instance declarations synthesized by the compiler and [translating] them just as [they] do for user-provided instances" [17].

Unfortunately, an automated approach does not resolve the issue of using only partial capabilities of the interactive theorem provers. The process of verification is still the same once the transpilation is done, and as such, the drawbacks we saw earlier still apply here.

Automatic Transpilation from a Proof Assistant

An alternative approach to the ones discussed above is to allow developers to write and verify their code in a proof assistant and then transpile it to a more widespread programming language. An example of this can be found in the program extraction of Coq [18]–[20], which allows Coq users to extract their programs into ML-like languages (currently supporting OCaml, Haskell and

Scheme). It allows users to export their verified Coq code and use it within projects in the other languages.

More recently, Cockx et al. have introduced AGDA2HS, "a tool that translates an expressive subset of Agda to readable Haskell, erasing dependent types and proofs in the process" [9]. AGDA2HS leans on the syntactic similarity between the two languages to formulate a common subset language which allows for "faithful" translations from Agda to Haskell. The Haskell code is generated by adding compilation pragmas [21] directly into the Agda code. AGDA2HS aims to make formal verification accessible to Haskell programmers — meaning that it also wants to provide access to the libraries and frameworks that are normally available in Haskell.

The most important benefit with this direction of transpilation is that users can write their programs with the unconstrained help of an interactive theorem prover. The tools then simplify their programs by erasing the parts that are only used for verification, and produce the minimal version of the code needed for its execution. This means that developers can benefit from the proof assistant's stronger type system as well as its interactive features (e.g. holes and automatic case splitting [22]). This also means that programs can be *intrinsically* verified and inherently correct instead of doing all the checks post-hoc.

A challenge and drawback that returns with this solution is the necessity for replicating the existing libraries of the target language in the proof assistant. This is an important thing to consider, because these libraries are an essential part of the ecosystems we are aiming to integrate in. There is an option to simply *postulate* the existence of these methods and types within the proof assistant. This will bring it into scope for development and type checking, which are the relevant phases for the proof assistant in this setup. Unfortunately, a function without an implementation is hard to reason about and would thus make verification of the programs using it more challenging. As a result, AGDA2HS contains an Agda version of the Haskell Prelude, a "trusted codebase" with implementations matching those in Haskell. In the rest of this thesis, we will refer to this library as the "AGDA2HS Prelude".

Liquid Haskell

While the solutions above do have their merits, there is always the question of the correctness of the transpilation to consider. The automated tools can aim to form a "trusted" translation process, but a simpler solution requiring no transpilation has less space for unwanted bugs. Liquid Haskell is a tool being developed by Vazou et al. [10], [23] which extends Haskell with refinement

types to allow formal reasoning about Haskell *in Haskell*. It uses an external SMT solver to check the correctness of the refinements meaning that some simple proofs can be done automatically, while others require users to supply the full reasoning themselves.

Unlike the solutions we discussed above, Liquid Haskell does not require users to learn a new language — everything is written in Haskell. This solution removes all the complications with unifying language features and making equivalent libraries available in both. The tool has recently been extended to include Liquid proof macros [24], a DSL that uses Template Haskell to generate additional proof terms in order to automate inductive proofs. However, this DSL is still embedded in Haskell and is works as part of the language.

Liquid Haskell does, however, come with its own drawbacks. Haskell was not designed to be a proof assistant and as such, does not include the more complex features found in languages such as Agda or Coq. These include strategies such as intrinsic verification, but also interactive editing features which help guide users through writing proofs (e.g. by providing the context and scope of the missing proof). Additionally, tools like Agda already have large proof libraries, whereas this is something that still needs to be developed for Liquid Haskell.

Chapter 3

AGDA2HS

For the remainder of this thesis, we focus on AGDA2HS as a case study. We chose to work with AGDA2HS for a variety of reasons, among which access to the implementors team and previous student work play a significant role. However, we believe that AGDA2HS is a good candidate, since its target audience are Haskell developers wanting to use verification in their projects and its design allows them to take full advantage of the capabilities of an interactive theorem prover.

To better understand the challenges with accessibility of AGDA2HS and similar tools, we take a look at previous work and conduct our own exploration to determine the current challenges with using it. At the end of this chapter, we form a hypothesis about what is necessary to make such a tool accessible and easy to use.

Since this thesis contains code snippets from both Agda and Haskell code, coloured backgrounds are used to create clarity and distinction. Snippets of Agda code are displayed on a blue background and snippets of Haskell code are displayed on a red background:

```
Agda
```

```
Haskell
```

3.1 Current Challenges in using AGDA2HS

As with all new tools that wish to make it in the real world, there are challenges to using AGDA2HS that need to be overcome. However similar Agda might be to Haskell, there are still core differences that require developers to learn new paradigms and forget old habits. Furthermore, AGDA2HS is still in its infancy, and thus has many features that might be incomplete, missing, or somewhat awkwardly designed.

Since AGDA2HS was first created, there have been multiple bachelor student projects exploring the capabilities and limitations of AGDA2HS. Posters and papers explainingd these projects can be found on the website of the TU Delft Research Project[1] under "Practical Verification of Functional Libraries" (2021) and "Practical Verification of Functional Programs" (2022). The theses are also available on the Educational Repository of the TU Delft[2].

During these projects, as well as during the design process of AGDA2HS, the various limitations and challenges of working with this tool have been discovered and rediscovered. They are listed and explained in the following sections.

3.1.1 Incomplete Standard Library

AGDA2HS is a project in its early stages, and there are a lot of functions and types missing in the "trusted codebase" of the AGDA2HS Prelude. This includes things from the Haskell Prelude (which does not need any explicit imports), but also from the `base` package[3], which is an essential part of Haskell development.

Additionally, an Agda implementation of a Haskell library needs something more: the proofs and properties that the exported methods should satisfy, available to the user to use in their own verification. Take the simple proofs about the `&&` construct displayed in Figure 3.1. While these are very intuitive rules and look concise when written out, Agda cannot infer them on its own. As you can imagine, though, they will most likely be needed for quite a few proofs by many potential users. Therefore, they should be included in the standard AGDA2HS library.

```
1 &&-leftAssoc : ∀ (a b c : Bool) → (a && b && c) ≡ ((a && b)
    && c)
2 &&-leftAssoc False b c = refl
3 &&-leftAssoc True b c = refl
4
5 &&-leftTrue : ∀ (a b : Bool) → (a && b) ≡ True → a ≡ True
6 &&-leftTrue True True h = refl
```

Figure 3.1: Proofs about the `&&` construct

[1] https://cse3000-research-project.github.io/

[2] https://repository.tudelft.nl/islandora/search/cockx?collection=education&f%5B0%5D=mods_genre_s%3A%22bachelor%5C%20thesis%22

[3] https://hackage.haskell.org/package/base

3.1.2 Missing Haskell Constructs

While AGDA2HS has support for many Haskell constructs, there are still a few missing that Haskell programmers would most likely find indispensable, especially when considering readability of the generated Haskell code. The most notable constructs are listed and explained here.

Guards

Guards in Haskell are an extension to pattern matching function parameters. They establish properties about the parameters and act as "guards" to which cases can pass into the implementation. Figure 3.2 contains a code snippet with guards which determines whether a student has received a passing grade.

```
1 passed :: Double -> Either String Bool
2 passed grade
3     | grade >= 5.75 && grade <= 10   = Right True
4     | grade >= 1    && grade <  5.75 = Right False
5     | otherwise                      = Left "Not␣a␣valid␣
        grade!"
```

Figure 3.2: Haskell function with guards

Guards make Haskell code readable and clear but, unfortunately, there is no Agda equivalent to this construct. Therefore, just like in Figure 3.3, they have to be replaced with `if_then_else_` expressions, resulting in more complex code. These are still manageable on a small scale, but can quickly clutter the code if the logic is more complex.

Type Class Deriving

Another important construct missing in AGDA2HS is type class `deriving`. This construct allows Haskell programmers to skip writing boilerplate code and generates straightforward implementations for them (such as for the `Eq` and `Show` type classes). An example of its use can be seen in Figure 3.4. It also extends the type class instance automatically when a new constructor is added to the type.

In the current version of AGDA2HS, the programmer needs to define every instance by hand in order to make the instance available within the Agda code. Figure 3.5 illustrates just how much more verbose our Haskell code ends up looking.

```
passed : Double → Either String Bool
passed grade =
    if grade >= 5.75 && grade <= 10 then
        Right True
    else
        if grade >= 1 && grade < 5.75 then
            Right False
        else
            Left "Not a valid grade!"

{-# COMPILE AGDA2HS passed #-}
```

```
passed :: Double -> Either String Bool
passed grade
  = if grade >= 5.75 && grade <= 10 then Right True else
      if grade >= 1 && grade < 5.75 then Right False else
        Left "Not␣a␣valid␣grade!"
```

Figure 3.3: Function implementation without guards

```
data Tree = Leaf | Branch Int Tree Tree
    deriving ( Eq )
```

Figure 3.4: Haskell datastructure `deriving` the `Eq` type class

Despite Agda not having an explicit notion of type classes, it is actually possible to mimic the `deriving` construct in Agda using data-type generic programming [25]. While this might be a viable approach in the future, it is currently under development and still missing support for important constructs in Haskell (e.g. nested inductive types). Since we do not actually have to run the generated code on the Agda side, it should be enough to find a solution that brings the instances into scope. The Agda code will, after all, not be able to influence the generated Haskell code - the verification of that generation should be done within the implementation of the generator.

Single-Constructor Types

Finally, there is no support for the `newtype` declaration in AGDA2HS. This is an alternative to the `data` declaration with some extra restrictions which allow for runtime optimisations. Though this feature might not be crucial, it is one of the more often-used Haskell features and as such should be included in AGDA2HS.

```
data Tree : Set where
    Leaf : Tree
    Branch : Int → Tree → Tree → Tree

eqTree : Tree → Tree → Bool
eqTree Leaf Leaf = True
eqTree (Branch x ll lr) (Branch y rl rr)
    = x == y && eqTree ll rl && eqTree lr rr
eqTree _ _ = False

instance
  iEqTree : Eq Tree
  iEqTree ._==_ = eqTree

{-# COMPILE AGDA2HS Tree #-}
{-# COMPILE AGDA2HS eqTree #-}
{-# COMPILE AGDA2HS iEqTree #-}
```

```
data Tree = Leaf
          | Branch Int Tree Tree

eqTree :: Tree -> Tree -> Bool
eqTree Leaf Leaf = True
eqTree (Branch x ll lr) (Branch y rl rr)
  = x == y && eqTree ll rl && eqTree lr rr
eqTree _ _ = False

instance Eq Tree where
    (==) = eqTree
```

Figure 3.5: Datastructure with implementation of `Eq` type class instance

3.1.3 Witnesses in Flow Control Constructs

As of the current state of AGDA2HS, the flow control constructs (`case_of_` and `if_then_else_`) do not contain witnesses of their branching condition. Consider, for example, the code snippet in Figure 3.6. If we were to navigate to either of the two holes and ask Agda to provide us the context, we would get the following:

```
Int (Goal)      i :  Int      j :  Int
```

In neither of the holes do we get any information about whether `i` and `j` are equal, even though we have just established it with the case match. This is usually not a problem when writing production code, but can make certain types of proofs as well as intrinsic verification impossible.

```
1 flow : Int → Int → Int
2 flow i j = case (i == j) of λ where
3     True → {!   !}
4     False → {!   !}
```

Figure 3.6: Case match with no witnesses

3.1.4 Lawful Type Classes

Many type classes being commonly used in Haskell carry with them an assumption of some laws which should hold for their instances. A prime example is the `Monad` type class, with its three monad laws [26]. Being able to safely assume that an instance follows the laws of its type class can allow for simplifications and optimisations in the code being written. Unfortunately, these laws are not enforced in Haskell - it is the developer's responsibility to design their instances correctly.

AGDA2HS is a very natural tool to help us enforce (and prove) these laws about the instances we define. Currently, lawful type classes are not included in the AGDA2HS standard library, though there is an issue on GitHub[4] regarding their addition. If the instances defined in the standard library were to have proven, lawful implementations, user-defined instances could benefit from these proofs.

3.1.5 Haskell Booleans vs. Agda Propositions

Boolean equality and other operations are an important part of development in almost every language: they control the program flow and they assert truths about the variables being used. However, this equality does not automatically translate into the propositional equality that is such a significant part of Agda. This means that, unfortunately, all the library methods and automatic things Agda can infer using the standard propositional library are not directly applicable to the boolean expressions of AGDA2HS.

As an example, take the implementation of the `transitivity` law of `Eq` in Figure 3.7. The `transitivityBool'` proof (lines 1-3) is defined in terms of the built-in equivalence. To finish the proof, Agda can resolve the only valid case match for you and the proof can easily be concluded with a simple `refl`. On the other hand, the `transitivityBool` proof (lines 6-10) is defined in terms of our Haskell boolean equality. Agda has to match on the constructors of `Bool` to be able to use `refl`. While that might seem like a small increase in lines of

[4] `https://github.com/agda/agda2hs/issues/108`

```
1  transitivityBool' : ∀ (x y z : Bool)
2              → (x ≡ y) → (y ≡ z)
3              → (x ≡ z)
4  transitivityBool' x .x .x refl refl = refl
5
6  transitivityBool : ∀ (x y z : Bool)
7              → (x == y) ≡ True → (y == z) ≡ True
8              → (x == z) ≡ True
9  transitivityBool False False False _ _ = refl
10 transitivityBool True True True _ _ = refl
```

Figure 3.7: An implementation for the `transitivity` law of `Eq`

code, `Bool` only has two possible constructors with no parameters. With larger data definitions, these can quickly explode into a large case match that needs to be repeated for every proof.

3.1.6 The Idiosyncrasies of Agda

While these are out of the scope of this thesis, there were multiple complaints within the bachelor projects about some things that are idiosyncratic to Agda. A significant one which was mentioned often was the termination checker. While the problem of terminating programs is undecidable [27] and thus has no existing general solution, many of the students struggled with how strict its implementation is in Agda. Agda is, of course, a crucial part of AGDA2HS, meaning that any usability issues with Agda will be reflected in AGDA2HS. However, this thesis aims to tackle the accessibility of these kinds of tools in general, and so we leave this for future work.

3.2 The Hypothesis

Based on the challenges identified in the previous sections, we hypothesise over what is necessary for a tool such as AGDA2HS to be accessible and easy to use:

1. **The tool has to support the usage of all commonly-used language features of the language being transpiled to.** This is to ensure that the tool remains accessible to programmers used to working in the popular language.
2. **The tool has to have all methods and types of the popular language implemented and accessible within the formal verification lan-**

guage. The standard library is a core feature of each language, and having implementations of them allows users to write proofs more easily but also to verify that they work in the expected way.

3. **The tool has to have basic proofs for all methods and types of the popular programming language accessible within the formal verification language.** Many proofs of new code require proofs about the existing code used within. To make the usage of the tool easier, it is necessary to provide these basic proofs about the existing code base. This also has the added benefit of making the expected properties of the code clearer to the developer.

Another very important point to consider is the validity of the translations and transpilations. For such a tool to be accessible, users need to trust that (a) the standard functions of the popular language are *correctly* re-implemented in the formal verification language and that (b) the transpilation is done correctly. After all, an incorrectly transpiled piece of code is useless no matter how confident we are in the correctness of the original implementation.

However, this is something that will not be considered within this thesis since trust in the codebase is necessary in every tool. We hope that by increasing accessibility, we increase the community able to contribute to the quality of AGDA2HS and develop trust in the codebase.

Chapter 4

Implementation

Based on the identified challenges, we settled on four features to improve, attempting to cover as many issues as possible while still being able to test the hypothesis. In the following sections, we go into more detail about the challenge with each of them and present the solution and its implications. For each solution, we also improved and updated the relevant documentation. We link to it throughout the following chapters.

4.1 Supporting the `newtype` Definition

Implemented in Pull Requests #141 and #167 of the AGDA2HS repository.

According to the Haskell Wiki, "the syntax and usage of newtypes is virtually identical to that of data declarations" [28]. There are, however, two extra restrictions:

1. A `newtype` declaration must have exactly one constructor.
2. The `newtype` construtor must have exactly one field.

Additionally, there is one crucial benefit that a `newtype` definition has over a `data` one: the new type and the type of its field can be treated as the same *at runtime*. In mathematical terms, they are *isomorphic*. Using `newtype` essentially allows for less overhead during runtime. What this means for practical matters is that undefined values will cause different behaviour for a `data` and `newtype` definition at runtime (examples can be viewed on the Haskell Wiki). However, since AGDA2HS currently does not support dealing with `undefined` values [9], we leave the behaviour for future work.

The Solution

Because the `newtype` definition is so similar to `data`, we can implement this construct very easily: by adding a `newtype` compilation pragma. This pragma can be used on data and record definitions which adhere to the single constructor and single field rules, throwing a relevant error in case they are violated. To compile a definition to a `newtype`, the developer needs to explicitly append the `newtype` keyword to the compilation pragma.

```
data Indexed (a : Set) : Set where
    MkIndexed : Int × a → Indexed a

{-# COMPILE AGDA2HS Indexed newtype #-}

record Identity (a : Set) : Set where
    constructor MkIdentity
    field
        runIdentity : a
open Identity public

{-# COMPILE AGDA2HS Identity newtype #-}

record Equal (a : Set) : Set where
    constructor MkEqual
    field
        pair : a × a
        @0 proof : fst pair ≡ snd pair
open Equal public

{-# COMPILE AGDA2HS Equal newtype #-}
```

```
newtype Indexed a = MkIndexed (Int, a)

newtype Identity a = MkIdentity{runIdentity :: a}

newtype Equal a = MkEqual{pair :: (a, a)}
```

Figure 4.1: The definition of single-field data types compiled to `newtype`

Figure 4.1 demonstrates the definition and compilation of `newtypes` using AGDA2HS, as simple data, as a simple record, and as a record with all but one field erased. These will all successfully compile when the `newtype` pragma is used. The three incorrect definitions in Figure 4.2 will yield the following error messages:

`Choice:`	“Newtype must have exactly one constructor in definition: Choice”
`Duo:`	“Newtype must have exactly one field in constructor: MkDuo”
`RecordDuo:`	“Newtype must have exactly one field in constructor: MkRecordDuo”

```
1  -- does not have exactly one constructor
2  data Choice (a b  : Set) : Set where
3    A : a → Choice a b
4    B : b → Choice a b
5
6  {-# COMPILE AGDA2HS Choice newtype #-}
7
8  -- constructor does not have exactly one field
9  data Duo (a b : Set) : Set where
10   MkDuo : a → b → Duo a b
11
12 {-# COMPILE AGDA2HS Duo newtype #-}
13
14 -- constructor does not have exactly one field
15 record RecordDuo (a b : Set) : Set where
16     constructor MkRecordDuo
17     field
18         left : a
19         right : b
20 open RecordDuo public
21
22 {-# COMPILE AGDA2HS RecordDuo newtype #-}
```

Figure 4.2: Wrong definitions of `newtype` data types

Constructor Naming

The one drawback of this solution is that the constructor name cannot be the same as the name of the new type (which is also the case with `data` declarations). This is something that is possible in Haskell but not allowed by Agda - hence we cannot replicate it directly in the Agda code. However, there is a workaround available in AGDA2HS for record `newtypes`: using the record constructor. This will automatically convert it to the same name constructor in Haskell. Figure 4.3 displays this transpilation.

```
record Identity (a : Set) : Set where
    field
        runIdentity : a
open Identity public

{-# COMPILE AGDA2HS Identity newtype #-}

makeIdentity : a → Identity a
makeIdentity a = record { runIdentity = a }

{-# COMPILE AGDA2HS makeIdentity #-}
```

```
newtype Identity a = Identity{runIdentity :: a}

makeIdentity :: a -> Identity a
makeIdentity a = Identity a
```

Figure 4.3: Obtaining the same type and constructor name

4.2 Supporting Standalone `deriving`

Implemented in Pull Request #161 of the AGDA2HS repository.

The existing solution to instance derivation in AGDA2HS (displayed in Figure 4.4) was to add the `deriving` expression to the pragma. This approach, however, has a very unfortunate limitation: Agda is not aware that these instances exist. This means that using the methods of the instances in any of the Agda code is not possible. While there are some scenarios in which that might not be necessary (such as instances only needed at runtime), there is actually a very good alternative which does not have this problem: using postulates.

The Solution

According to the Agda docs, "a postulate is a declaration of an element of some type without an accompanying definition" [29]. It essentially makes Agda's type checker believe that the element in the postulate exists, even though we have not defined it. Since we never execute the code in Agda, this is a reasonable solutions to make derived type classes available in both versions of the code.

To be able to achieve the exact Haskell syntax as in Figure 4.4, AGDA2HS would need an awareness of the entire program being compiled (which is currently not the case). This is because while the `deriving` clause is part of the data definition in Haskell, postulates can be located anywhere in the Agda

```
data Direction : Set where
  North : Direction
  South : Direction
  East  : Direction
  West  : Direction

{-# COMPILE AGDA2HS Direction deriving (Eq, Show) #-}
```

```
data Direction = North
               | South
               | East
               | West
                   deriving (Eq, Show)
```

Figure 4.4: Adding a `deriving` expression to the AGDA2HS pragma

code. Luckily, standalone `deriving` declarations are also possible in Haskell. Thus, Figure 4.5 demonstrates how AGDA2HS is now able to deal with deriving type class instances.

```
postulate instance
  iEqDirection : Eq Direction
  iShowDirection : Show Direction

{-# COMPILE AGDA2HS iEqDirection #-}
{-# COMPILE AGDA2HS iShowDirection #-}
```

```
{-# LANGUAGE StandaloneDeriving #-}

deriving instance Eq Direction

deriving instance Show Direction
```

Figure 4.5: Using postulates to derive type class instances

There are two things to note about this solution:

1. There is no need to add anything to the pragma, since AGDA2HS can automatically discover whether the instance is implemented or only a postulate. However, there might be some cases in which the developer might want to implement the instance on the Agda side (e.g. to make a proof simpler) but turn it into a derivation on the Haskell side. For this case, there also exists a `derive` pragma, discussed in the following subsection.

2. Since standalone deriving is only available since `GHC2021`, a language flag is automatically added on top of the Haskell file to enable the feature for older GHC versions (seen in Line 1 of the Haskell code in Figure 4.5).

```
1  instance
2    iEqDirection : Eq Direction
3    iEqDirection ._==_ North North = True
4    iEqDirection ._==_ South South = True
5    iEqDirection ._==_ East  East  = True
6    iEqDirection ._==_ West  West  = True
7    iEqDirection ._==_ _     _     = False
8
9  {-# COMPILE AGDA2HS iEqDirection derive #-}
10
11 record Clazz (a : Set) : Set where
12   field
13     foo : a → Int
14     bar : a → Bool
15
16 open Clazz {{...}} public
17
18 {-# COMPILE AGDA2HS Clazz class #-}
19
20 postulate instance iDirectionClazz : Clazz Direction
21
22 {-# COMPILE AGDA2HS iDirectionClazz derive anyclass #-}
```

```
1  {-# LANGUAGE StandaloneDeriving, DerivingStrategies,
       DeriveAnyClass
2    #-}
3
4  deriving instance Eq Direction
5
6  class Clazz a where
7      foo :: a -> Int
8      bar :: a -> Bool
9
10 deriving anyclass instance Clazz Direction
```

Figure 4.6: Using the `derive` pragma

The `derive` Pragma

While a simple postulate is easy for AGDA2HS to recognise and compile, there are some cases in which the pragma might need more configurations to achieve

the desired runtime behaviour on the Haskell side. One such scenario (already mentioned above) is the ability to compile implemented instances into derivations. The other important scenario is the use of *deriving strategies* [30] which tell the Haskell compiler how to generate the derived instances.

Consider the code in Figure 4.6. Without the `derive` keyword, the pragma on Line 17 would have compiled the actual instance implementation to Haskell. Now, the Haskell code is clean and readable, while the Agda code has a specific implementation for proofs to rely on. The pragma on Line 30 tells AGDA2HS to compile the postulate to a derivation which uses the `anyclass` strategy. Since `Clazz` is not one of the type classes for which GHC can generate a derivation natively, the Haskell code would not compile without this specification.

Since deriving strategies are not included in the GHC compiler by default, AGDA2HS sets the necessary language flags. They are included if and only if they are needed.

4.3 Adding Witnesses to Flow Control

Implemented in Pull Request #156 of the AGDA2HS repository.

Unlike in the previous two improvements, this feature mostly required changes to the AGDA2HS Prelude instead of to the actual compilation logic of AGDA2HS. This is also the point where it started to become apparent that while a Haskell library contains data definitions and functions, an equivalent in Agda should also contain the relevant proofs. The original definitions of the control flow constructs in Agda are shown in Figure 4.7.

```
1 infix -1 case_of_
2 case_of_ : a → (a → b) → b
3 case x of f = f x
4
5 infix -2 if_then_else_
6 if_then_else_ : {@0 a : Set ℓ} → Bool → a → a → a
7 if False then x else y = y
8 if True  then x else y = x
```

Figure 4.7: The original definitions of the control flow constructs

The Solution

We present the two possible options of solving the problem of witnesses in flow control. Both of these allow for a "witnessed" usage as well as a simple one, meaning that branching operations which do **not** require a witness of their condition retain the same simple syntax as they did with their original implementation.

Option 1: Using a `Witness` datatype within the simple construct.

The key idea here would be to match on the "witnessed" cases of the condition. For this, we require the definition of a `Witness` data type (also known as a "singleton" type [31]), which would be able to transparently wrap the condition and the proof of its result. Figure 4.8 displays the definition of such a type.

```
1  record Witness (A : Set) (a : A) : Set where
2    constructor _<>
3    field
4      el : A
5      @0 {{ pf }} : a ≡ el
6  open Witness public
7
8  {-# COMPILE AGDA2HS Witness unboxed #-}
9
10 pattern _<_> el pf = _<> el {{ pf }}
11
12 witness : {A : Set} → (a : A) → Witness A a
13 witness a = a <>
14
15 {-# COMPILE AGDA2HS witness transparent #-}
```

Figure 4.8: The definition of the helper `Witness` data type

The `Witness` data type is compiled using the `unboxed` pragma, which tells AGDA2HS that it is *isomorphic* to its single visible field. This means that in Haskell, we would see any `Witness` replaced by its `el` field. Similarly, the accompanying `witness` function is compiled using the `transparent` pragma, letting AGDA2HS know that the function should not appear in the Haskell code (it is "transparent").

Figure 4.9 contains a simple branching example where a `Witness` is used to obtain the necessary equality proof. On the Agda side, the proofs are available whenever needed and allow for the intrinsic verification of the `EqPair` data type, which is meant to store two elements for which we can provide a proof

of equality. On the Haskell side, we see that it compiles to a simple case match with no complicated leftover constructs.

```
data EqPair (a : Set) : Set where
  MkEqPair : ((a' , a'') : a × a) → @0 {{ a' ≡ a'' }} →
     EqPair a

{-# COMPILE AGDA2HS EqPair newtype #-}

postulate bool2prop : ∀ (i j : Int) → (i == j) ≡ True → i ≡ j

makeEqPair : Int → Int → Maybe (EqPair Int)
makeEqPair i j =
  case (witness (i == j)) of λ where
    (True < h >) → Just (MkEqPair (i , j) {{ bool2prop i j h
       }})
    (False <>) → Nothing

{-# COMPILE AGDA2HS makeEqPair #-}
```

```
newtype EqPair a = MkEqPair (a, a)

makeEqPair :: Int -> Int -> Maybe (EqPair Int)
makeEqPair i j
  = case i == j of
       True -> Just (MkEqPair (i, j))
       False -> Nothing
```

Figure 4.9: A simple branching function requiring the use of a `Witness`

While this solution works, there are two big problems with it. Firstly, the Agda code is wordy and does not resemble standard Haskell code at all. Keeping in mind that AGDA2HS is meant to be a tool for Haskell programmers, this might not be an ideal solution to providing a *usable* feature. Secondly, it is not applicable to `if`-statements. The type of the condition becomes a `Witness` of the actual condition and while this is no problem for case matches, an `if`-statement explicitly requires a boolean. We could, of course, add a version of the `if_then_else_` definition into the standard library which decomposes the `Witness` for us. By then, though, we might as well make it simpler since we are adjusting the code in AGDA2HS anyway.

Option 2: Building the constructs with intrinsic proofs.

The alternative to Option 1 is to actually change the signatures of the flow constructs in the AGDA2HS Prelude. The original definitions have a simple,

Haskell-like signature with very straightforward implementations. Figure 4.10 displays the *new* type signatures of the flow control constructs (the implementations stay the same).

```
1  case_of_ : (a' : a) → ((a'' : a) → @0 {{ a' ≡ a'' }} → b) → b
2
3  if_then_else_ : {@0 a : Set ℓ} → (flg : Bool)
4     → (@0 {{ flg ≡ True }} → a) → (@0 {{ flg ≡ False }} → a)
5     → a
```

Figure 4.10: The new type signatures of the control flow constructs

With this redesign, it is possible to carry the witness of the control flow as an *instance* [32] - meaning Agda can sometimes use and resolve them without the developer needing to explicitly worry about them. An example of this can be seen in Figure 4.11: the `MkRange` constructor needs an *instance* of an ordering proof to construct the `Range`. The `then` branch of the expression on Line 18 automatically brings such an instance into scope - meaning that there is no need to explicitly pass it, and the code looks almost identical to the generated Haskell.

```
1  data Range : Set where
2      MkRange : (low high : Int)
3              → {{ @0 h : ((low <= high) ≡ True) }}
4              → Range
5
6  {-# COMPILE AGDA2HS Range #-}
7
8  createRange : Int → Int → Maybe Range
9  createRange low high = if low <= high then Just (MkRange low
       high) else Nothing
10
11 {-# COMPILE AGDA2HS createRange #-}
```

```
1  data Range = MkRange Int Int
2
3  createRange :: Int -> Int -> Maybe Range
4  createRange low high
5    = if low <= high then Just (MkRange low high) else Nothing
```

Figure 4.11: `if` with the proof instance automatically brought into scope

If the automatic passing of the proof instance is not enough, it is also possible to explicitly bring the witnesses into scope of the branch. This can be

useful when, for example, the witness needs to be transformed or combined with another proof before it can be used in the branch. An example can be found in Figure 4.12

```
ifEqPair : Int → Int → Maybe (EqPair Int)
ifEqPair i j =
  if (i == j) then
    (λ {{ h }} →
      Just (MkEqPair (i , j) {{ bool2prop i j h }}))
  else Nothing

{-# COMPILE AGDA2HS ifEqPair #-}

caseEqPair : Int → Int → Maybe (EqPair Int)
caseEqPair i j =
  case (i == j) of λ where
    True {{ h }} →
      Just (MkEqPair (i , j) {{ bool2prop i j h }})
    False → Nothing

{-# COMPILE AGDA2HS caseEqPair #-}
```

```
ifEqPair :: Int -> Int -> Maybe (EqPair Int)
ifEqPair i j
  = if i == j then Just (MkEqPair (i, j)) else Nothing

caseEqPair :: Int -> Int -> Maybe (EqPair Int)
caseEqPair i j
  = case i == j of
      True -> Just (MkEqPair (i, j))
      False -> Nothing
```

Figure 4.12: Flows with explicit usage of the witness proofs (using the `EqPair` type from Figure 4.9)

The Consequences

The change of type signature of the two flow control constructs had some design implications on AGDA2HS. Both changes were due to the change in type signature affecting the correct behaviour of these features. However, both were deemed redundant and have been removed to make room for the new implementation. Detailed explanations can be seen in the description of the Pull Request.

Inline functions in `case_of_` are no longer supported.

It is no longer possible to use simple inline functions such as `length` as a second parameter in `case_of_`. This feature was arguably redundant though, since the generated Haskell can be achieved (much more straightforwardly) without using `case_of_` (i.e. `case xs of length` in Agda compiles to `length xs` in Haskell - which can be achieved by writing `length xs` on the Agda side).

Support for partially applied `if_then_else_` and `case_of_` is removed.

While Haskell does offer infix operators for user defined data types, it does not do so for `if_then_else_` and `case_of_`, which are native constructs. This requires a user to explicitly write a lambda if they wish to partially apply it, which is now also the case in AGDA2HS.

4.4 Adding Lawful Type Classes to Prelude

Implemented in Pull Requests #160 and #165 of the AGDA2HS repository.

Just like with flow control witnesses, this is a challenge that had to be addressed at the level of the AGDA2HS Prelude rather than in the actual codebase of AGDA2HS. It is obvious that the relevant laws need to be added to the type classes, but there are multiple decisions to make in the design.

The Solution

The first thing to determine is whether to include the laws in the base type class. In this case, the laws are part of each instance definition and are erased during the transpilation to Haskell. The alternative approach is to create an `IsLawful` definition for each type class which users can choose to implement when necessary.

The solution that we went for in this case is to keep the two type classes separate. If we included the laws within the default type classes, users would always have to provide the proof. With the second approach, the logic is divided and the laws are present only when they are necessary. This decision actually led us to separate the `Haskell.Prelude` library into two packages: the existing `Prim` package, containing the data and method definitions of the AGDA2HS Prelude, and the `Law` package, containing the proofs and laws about the definitions in `Prim`.

The second decision we had to make was about bundling versus parameterising [33] - two ways in which to relate type classes to each other. Consider

the `Eq` and `Ord` type classes. In order to be able to implement an `Ord` instance, there already needs to exist an `Eq` instances for that type. There are two ways to implement this, demonstrated in Figure 4.13: make `Eq` a field in `Ord` (Lines 1-5) or make `Eq` a parameter in `Ord` (Lines 7-10). The same two options are available to relate `Eq` with `IsLawfulEq`.

```
1  -- Eq bundled in Ord
2  record Ord (a : Set) : Set where
3    field
4      overlap {{ super }} : Eq a
5      ...
6
7  -- Eq parameterised in Ord
8  record Ord (a : Set) {{ super : Eq a }} : Set where
9    field
10     ...
```

Figure 4.13: Bundling vs. Parameterising

The design we settled on is illustrated in Figure 4.14: prerequisite type classes are bundled in other type classes (this was already the implementation) and parameterised in their lawful variants. The best way to explain why this was the choice is to demonstrate the results.

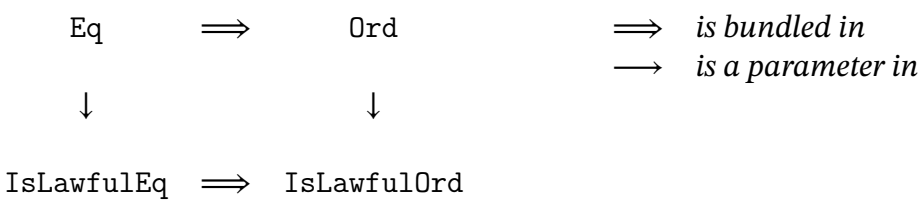

Figure 4.14: Bundling vs. Parameterising of Type Classes

Consider the code snippet in Figure 4.15. There is no need to pass an `Eq` instance to the `minimum` function because `Eq` is *already bundled in* `Ord`. This makes the function look very similar to the Haskell one. On the other hand, the `makeEqual` function needs both an `Eq` and an `IsLawfulEq` instance. While this might seem like unnecessary verbosity, we argue that it brings *clarity*. There is a clear division between the part of the type signature that will be translated to Haskell and the part that will be erased.

Unlike the previous three features we implemented, this one is more of a guideline than an implementation. We have implemented example lawful

```
minimum : {{ Ord a }} → a → a → a
minimum i j = if i < j then i else j

{-# COMPILE AGDA2HS minimum #-}

makeEqual : {{ iEqA : Eq a }} → @0 {{ IsLawfulEq a }}
              → a → a → Maybe (EqPair a)
makeEqual i j =
  if i == j then (λ {{ h }} →
    Just (MkEqPair (i , j) {{ equality i j h }}))
  else Nothing

{-# COMPILE AGDA2HS makeEqual #-}
```

```
minimum :: Ord a => a -> a -> a
minimum i j = if i < j then i else j

makeEqual :: Eq a => a -> a -> Maybe (EqPair a)
makeEqual i j =
  if i == j then Just (MkEqPair (i, j)) else Nothing
```

Figure 4.15: Using Lawful Type Classes

type classes as well as some of their instances for other AGDA2HS developers to follow. While this separation of transpiled code from erased code might not be the way Agda developers are used to working, we believe that it provides clarity for Haskell developers and will hopefully make the transition easier for them.

Lawful Eq

During the implementation of lawful type classes, we also touched upon the problem of boolean and propositional equality. To model the relationship that they have, we used the `Reflects` idiom (used in Line 3 of Figure 4.16) used in the `IsLawfulEq` type class. This allows users to tell Agda that the two equalities "reflect" each other.

With this implementation, users only have to prove the `isEquality` field for type they are working with. Figure 4.17 shows an example of this for `Bool`. With this short implementation, we get the conversion between boolean and propositional equality for free and are even able to prove the laws for boolean equality generically. The boolean equality proofs and the definition of the `Reflects` idiom are included in Appendix A).

```
record IsLawfulEq (e : Set) {{ iEq : Eq e }} : Set₁ where
  field
    isEquality : ∀ (x y : e) → Reflects (x ≡ y) (x == y)

  equality : ∀ (x y : e) → (x == y) ≡ True → x ≡ y
  equality x y h = extractTrue {{ h }} (isEquality x y)

  nequality : ∀ (x y : e) → (x == y) ≡ False → (x ≡ y → ⊥)
  nequality x y h = extractFalse {{ h }} (isEquality x y)

  -- contrapositive of nequality
  equality' : ∀ (x y : e) → x ≡ y → (x == y) ≡ True
  equality' x y h with x == y in eq
  ... | False = magic (nequality x y eq h)
  ... | True = refl

  -- contrapositive of equality
  nequality' : ∀ (x y : e) → (x ≡ y → ⊥) → (x == y) ≡ False
  nequality' x y h with x == y in eq
  ... | True = magic (h (equality x y eq))
  ... | False = refl

open IsLawfulEq {{ ... }} public
```

Figure 4.16: The Definition of `IsLawfulEq`

```
instance
  iLawfulEqBool : IsLawfulEq Bool
  iLawfulEqBool .isEquality False False = ofY refl
  iLawfulEqBool .isEquality False True  = ofN λ()
  iLawfulEqBool .isEquality True  False = ofN λ()
  iLawfulEqBool .isEquality True  True  = ofY refl
```

Figure 4.17: An instance of `IsLawfulEq` for `Bool`

Chapter 5

User Study Setup

To evaluate the usability of the improved version of AGDA2HS, we conducted a user study, where we asked participants to complete two small programming assignments in AGDA2HS and then evaluate their experience by filling in a questionnaire. This chapter describes the setup for the user study, explaining the goals and means of conducting it.

5.1 Sources of Inspiration

As all the user-oriented work in formal verification so repetitively states, there has not been much user-oriented work done in formal verification. Despite these circumstances, there are two works which we used as inspiration for our user study.

Firstly, the work by Gamboa et al. [34] on usability-oriented design of liquid types in Java helped us with the design of the goals and assignments in our user study. Their paper describes the process of designing liquid types based on user feedback and then evaluating their design by conducting a user study. We adopt their approach in defining research questions for the user study and designing the assignments and questionnaire to answer those.

Secondly, we took inspiration for our questionnaire from the paper by Beckert and Grebing [35] where they evaluate the usability of interactive verification systems. They recommend a source of questions to evaluate software and provide tips on how to convert closed questions to more informative open questions. They also warn that many participants create different scales when answering questions (e.g. for evaluating their skill levels with a certain tool), so clarity and specific instructions are important when designing a user study.

5.2 The Goals

The goals of this user study are to, of course, evaluate the work done during this thesis, but also to take a first structured look at the usability of AGDA2HS. Since the resources of a user study done as a small part of a master thesis are quite limited, this study is meant more as a prototype which will hopefully lead to both insights and future user studies.

Taking inspiration from the user study of Gamboa et al. [34], we define research questions which this user study will attempt to answer:

1. What are the challenges with using AGDA2HS?
2. Under which circumstances are developers open to using AGDA2HS? In which cases do they think such a tool is useful?
3. Which constructs and features are necessary to make AGDA2HS accessible and easy-to-use?

5.3 Recruitment

To recruit participants for the study, we posted an announcement with a leaflet (attached in Appendix B) in a channel of the Programming Languages research group meant for discussing verification tools, in the off-topic channel of the teaching assistants, and on the course page of CSE3100 Functional Programming 2022/2023[1]. By these means we hoped to acquire participants with diverse experience with both Haskell and Agda as well as with diverse backgrounds and interests. Obviously, participants from industry were not present, but such recruitment was not possible within the scope of our prototype user study.

Candidates were asked to fill in a sign-up form, where they had to agree with the consent statement (attached in Appendix C), provide their e-mail address, and indicate which operating system they use. E-mail addresses were never exported and they were used only to communicate information regarding the user study prior to its execution. Operating systems were only reviewed and used to set up an easy-to-use programming environment.

5.4 The Programming Assignments

For participants to be able to evaluate the usability and potential of AGDA2HS, they had to receive the opportunity to work with the tool itself. In order to do

[1] `https://studiegids.tudelft.nl/a101_displayCourse.do?course_id=61501`

this in as simple a way as possible, two tailored assignments were designed to let them explore the most relevant parts of the tool. The general idea was that they would be able to complete the assignments on their own with only the help of the AGDA2HS docs, the AGDA2HS Prelude, and the examples available in the AGDA2HS repository.

The Environment Setup

To ensure that participants could directly dive into implementing the programming assignments, we set up a coding environment using a virtual machine (VM) and a prepared project. The virtual machine was running a Debian instance with access to the internet and had all the necessary tools preinstalled: the correct version of Agda, the latest version of AGDA2HS in its build path, and Visual Studio Code.

The project[2] that the participants would be working in during the study was setup in the following way:

`app`: The directory containing the executable Haskell app which, when run, executes QuickCheck tests for the specified assignment.

`lib`: The directory containing the Agda solution files. *This is the only directory that the participants are supposed to edit.*

`lib/Help`: A directory containing helper methods for the exercises.

`src`: A git-ignored directory into which AGDA2HS generates the Haskell code.

`default`: : A helper directory for this particular project setup which contains partial Haskell implementations of all the exercises. They are used to make the Haskell code compile even when the exercise has not been implemented yet. The are **not** meant to serve as inspiration.

A `Makefile` was included to let participants build their solutions and execute the tests, allowing them to verify their solutions. The `README` contained this project structure explanation as well as the assignments that the participants were to complete. It also included an example assignment with an example solution in the `lib` directory. The example defines a `reverse` function for which certain properties are extrinsically verified. To build and run the tests for this example assignment, participants simply had to run `make A=reverse`.

[2]Publicly available at the following GitHub repository: `https://github.com/sarajuhosova/agda2hs-user-study`

After signing up for the user study, the participants received an email (included in Appendix D) explaining the steps they can take *before* they arrive in order to ensure a smooth start. This included explaining how to either install and run the VM or setup the environment on their own machines. The email also linked to an "example" project, which they could use to verify that everything was setup and running correctly. This project was a minimal version of the actual user study project, having the same structure and including the `reverse` example, but excluding the instructions to the actual assignments.

Assignment 1: The `All` Type

The first assignment asked the participants to "implement the Haskell `All` type - the boolean monoid under conjunction". It was divided into eight subassignments, the solutions for which can be found in Figure 5.1[3] (the relevant lines are mentioned below):

1. **Define the `All` type.** [Lines 1 and 8-13]
   The `All` type in Haskell is defined as a record `newtype`. Participants could search the documentation for how to achieve this. The QuickCheck tests also required the type and constructor name to be the same, which is normally not possible in Agda. A workaround for this problem is also explained in the documentation. An `All` type is already defined in the AGDA2HS Prelude, so an extra challenge was to learn how to hide imports (this could have been copied from the `reverse` example assignment).
2. **Create an instance for `Eq`, `Ord`, `Show`, and `Bounded`. These should all be available on the Agda side.** [Lines 16-20 and 38-41]
   This part of the assignment was aimed at postulating instances to achieve standalone derivations, since this is how they are implemented in the source code on Hackage. It is possible to define and compile the `Eq` type class easily, but the rest become very complicated if the participant did not use postulates and deriving.
3. **Create an instance for `Semigroup` that is available on the Agda side.** [Lines 22-24 and 42]
   This instance has to actually be implemented, since there is no way for Haskell to derive that this is a conjunction associativity function. The participant can choose to either implement the `_<>_` function logically (as in our example solution) or attempt to copy the use of an imported `coerce` function, as done in the source code on Hackage.

[3] To view the Haskell version of this code, see Appendix E.

```
1  open import Haskell.Prelude hiding ( All )
2
3  {-# FOREIGN AGDA2HS
4  {-# LANGUAGE DeriveGeneric #-}
5  import GHC.Generics
6  #-}
7
8  record All : Set where
9    field
10     getAll : Bool
11 open All public
12
13 {-# COMPILE AGDA2HS All newtype deriving (Read,  Generic) #-}
14
15 instance
16   postulate
17     iAllEq      : Eq      All
18     iAllOrd     : Ord     All
19     iAllShow    : Show    All
20     iAllBounded : Bounded All
21
22   iAllSemigroup : Semigroup All
23   iAllSemigroup ._<>_ a b =
24     record { getAll = (getAll a && getAll b) }
25
26   iAllLawfulSemigroup : IsLawfulSemigroup All
27   iAllLawfulSemigroup .associativity
28     record { getAll = False } b c = refl
29   iAllLawfulSemigroup .associativity
30     record { getAll = True } b c = refl
31
32   iAllMonoid : Monoid All
33   iAllMonoid .mempty = record { getAll = True }
34   iAllMonoid .mappend = _<>_
35   iAllMonoid .mconcat [] = mempty
36   iAllMonoid .mconcat (x :: xs) = x <> mconcat xs
37
38 {-# COMPILE AGDA2HS iAllEq #-}
39 {-# COMPILE AGDA2HS iAllOrd #-}
40 {-# COMPILE AGDA2HS iAllShow #-}
41 {-# COMPILE AGDA2HS iAllBounded #-}
42 {-# COMPILE AGDA2HS iAllSemigroup #-}
43 {-# COMPILE AGDA2HS iAllMonoid #-}
```

Figure 5.1: The intended solution for Assignment 1 of the user study

4. **Prove that the `Semigroup` instance is lawful (you can see which laws should hold in the Haskell documentation).** [Lines 26-30]
   At the moment of the user study, there was actually no documentation on lawful type classes, which the participants might have used in this sub-assignment. There were examples available in the AGDA2HS repository (which they were encouraged to use). It was also possible to define and prove the associativity themselves in any way they wanted, since this is code that should not be compiled to Haskell.
5. **Create an instance for `Monoid` that is available on the Agda side.** [Lines 32-36 and 43]
   While very similar to the `Semigroup` instance, this instance could actually be defined using a default field implementation. Unfortunately, at the time of the user study, there was a bug with imports when using default field implementations (explained in Issue #169), so the easier solution was to just define all the fields (as in our example solution).
6. **(Optional) Create an instance for `Read`. This instance does not need to be available on the Agda side.** [Line 13]
   Since this instance did not need to be available on the Agda side, it was enough to just add a deriving clause to the compilation pragma. It was optional because it tested an *alternative* to a feature added during this thesis, not the feature itself.
7. **(Optional) Create an instance for `Generic`. This instance does not need to be available on the Agda side.** [Lines 3-6 and 13]
   This sub-assignment was similar to the instance for `Read`, but it required the foreign pragma with an import and a language flag. The error messages should have been sufficient for participants to figure this out.

The general idea of this assignment was to get acquainted with AGDA2HS. The participants were only asked to prove a simple associativity law, and so could mostly get by with only Haskell programming logic. The assignment was designed in such a way, that if the participants opened the Haskell source available on Hackage, they could reverse-engineer the entire implementation by consulting the AGDA2HS documentation.

The QuickCheck tests verified that the following held for the solution:

- That an `Eq` instance exists such that
  `(a == b) == (All a == All b)`.
- That an `Ord` instance exists such that
  `compare a b == compare (All a) (All b)`.

- That a `Show` instance exists such that
  `("All getAll = " ++ show a ++ "") == show (All a)`
  (this is what the standardly derived `Show` instance does).
- That a `Bounded` instance exists such that
  `(minBound == All False) && (maxBound == All True)`.
- That a `Semigroup` instance exists such that
  `All a <> (All b <> All c) == (All a <> All b) <> All c`.
- That a `Monoid` instance exists such that:
  - `mempty == All True`,
  - `mappend (All a) (All b) == All a <> All b`,
  - `mconcat (map All xs) == All (foldl (&&) True xs)`.

Assignment 2: Safe `lookup`

The second assignment asked the participants to "implement a `lookupSafe` function which looks up a value in a list of key-value pairs". The compiled Haskell signature of this method was to be:

```
lookupSafe ::  Eq a => a -> [(a, b)] -> b
```

The participants were asked to include a guarantee in their Agda code of the lookup action happening only if the list contains the key. The QuickCheck tests verify that if a key *is* in a list, the `lookupSafe` function will find it. They were given two helper functions in the `lib/Help` directory (displayed in Figure 5.2). To complete this assignment, it was necessary to use the AGDA2HS features more tailored to Agda itself: flow control constructs with witnesses, (possibly) the `IsLawfulEq` type class, and erasure.

The intended solution can be seen in Figure 5.3[4] It is not possible on Line 13 to recursively call the `lookupSafe` function without providing a proof that that tail of the list contains the key. To construct this proof, the witness of the `fst x == key ≡ False` condition in the `else` branch is needed. This boolean equality then needs to be converted into the Agda propositional equality `fst x == key → ⊥`, which can be done using the `IsLawfulEq` type class.

Unfortunately, lawful type classes were not yet documented during the user study, and so it was only possible to complete this assignment with sufficient exploration of the AGDA2HS Prelude.

[4]To view the Haskell version of this code, see Appendix E.

```
contains : {{ Eq a }} → a → List (a × b) → Bool
contains key [] = False
contains key (x :: xs) =
  if fst x == key then True else contains key xs

{-# COMPILE AGDA2HS contains #-}

postulate
  containsTail : {{ iEqA : Eq a }}
    → ∀ (key : a) (x : a × b) (xs : List (a × b))
    → IsTrue (contains key (x :: xs))
    → (fst x ≡ key → ⊥)
    → IsTrue (contains key xs)
```

```
contains :: Eq a => a -> [(a, b)] -> Bool
contains key [] = False
contains key (x : xs)
  = if fst x == key then True else contains key xs
```

Figure 5.2: The helper functions for Assignment 2

```
open import Haskell.Prelude

open import Help.Contains

lookupSafe : {{ iEqA : Eq a }} → @0 {{ IsLawfulEq a }}
               → (key : a) → (xs : List (a × b))
               → @0 {{ IsTrue (contains key xs) }}
               → b
lookupSafe key (x :: xs) {{ hc }} =
  if fst x == key then
    snd x
  else
    λ {{ neq }} → lookupSafe key xs {{
      containsTail key x xs hc (nequality _ _ neq)
    }}

{-# COMPILE AGDA2HS lookupSafe #-}
```

Figure 5.3: The intended solution for Assignment 2 of the user study

5.5 The Questionnaire

After completing the programming assignments to any degree of completion, participants were asked to complete a questionnaire. In combination with

an inspection of the programming solutions, the aggregated responses to this questionnaire would hopefully answer the research questions defined above - and thus also evaluate the work done during this thesis.

Many questions were drawn from the Software Usability Measurement Inventory (SUMI), "a rigorously tested and proven method of measuring software quality from the end user's point of view" [36]. SUMI is an inventory of 50 questions to which participants can reply with *Agree*, *Undecided*, or *Disagree*. It was decided during the development, that adding *Strongly (dis)agree* options only made it harder for participants to make a judgement. SUMI is meant to evaluate user experience of *any* software, meaning that some questions are not directly relevant. An example of this is the statement "the organisation of the menus seems quite logical" which might be applicable to software with a UI, but is not applicable to AGDA2HS.

To also get some insight into concrete future work, the SUMI questions were complemented with open questions. These will be used to evaluate not only the general opinions, but also to get specific insights and recommendations.

The questionnaire was divided into seven sections whose content and purpose we describe below:

1. **Welcome!** The welcome page was used to kick off the session, giving participants useful links to documentation, as well as a link to download the actual project. It contained instructions on where to find the assignments and how to begin.

2. **Your Solution.** This section was used to sort the submissions. Each participant was given a unique identifier when they started the user study and they were asked to compress their `lib` folder into a ZIP archive, rename it to that identifier, and upload it to the SURF drive linked in the questionnaire. They were then asked to input the identifier into a field in this section, such that we would be able to match solutions to answers (this was the only mandatory field in the entire questionnaire).

   This section also asked participants to mark which assignments they completed fully and to indicate the time spent per assignment. They were also asked whether they had read any AGDA2HS documentation before arriving to the session.

3. **Your Skills.** To be able to understand where the feedback was coming from, we asked users to indicate their experience with Haskell and Agda respectively. This was done using multi-select choice questions, where users were asked to indicate in which scenarios they had used the

given language before and which language features they are comfortably familiar with. The features listed out for Haskell were taken from this comprehensive Haskeller Competency Matrix.

4. **Your Experience.** This section was design to answer the first research question, by asking participants to agree, not decide on, or disagree with statements relating to the complexity of using AGDA2HS. These included statements such as "once I have a clear idea of how I want to implement something, it is easy to achieve it using Agda2HS" as well as "the documentation was very clear" or "error messages provided by Agda2HS are adequate". These statements were complemented with two open questions which asked for participants' opinions about the best aspects and most needed improvements with regard to ease-of-use and understanding of AGDA2HS.

5. **Your Impressions.** This section aimed to answer the second research question, asking participants to react to statements about recommending AGDA2HS to others or liking / disliking working with AGDA2HS. They were also asked to give their thoughts on the cases in which AGDA2HS would or would not be useful in two open questions.

6. **Some Technical Reflection.** The section hopes to provide answers to the third research questions using three different tactics. Firstly, participants were asked to indicate which of the four new features they had used in their code and which they thought are necessary for the accessibility and ease-of-use of AGDA2HS. Next, they had space to indicate which Haskell or Agda features they were missing in the tool by answering two separate open questions. Lastly, they were asked to agree, not decide on, or disagree with the three hypotheses that we determined were necessary for the accessibility and ease-of-use of tools such as AGDA2HS.

7. **Anything Else?** In case there was something else participants wanted to mention about their experience with AGDA2HS, this section had one open text field that let them do so.

A copy of the full questionnaire can be found in Appendix F.

Chapter 6

Results

In the following chapter, we present the aggregated results of the user study based on the code submissions and questionnaire answers[1].

The user study was attended by a total of 9 participants, all students from various study levels (i.e. bachelor, master, and PhD) and with varying degrees of experience in both Haskell and Agda. To evaluate their experience, we asked participants to indicate which features in each language they are "comfortably familiar with". We used this information to determine each participant's level of Haskell and Agda skills. The results can be viewed in Figure 6.1 (for more details on how this was computed, see Appendix G).

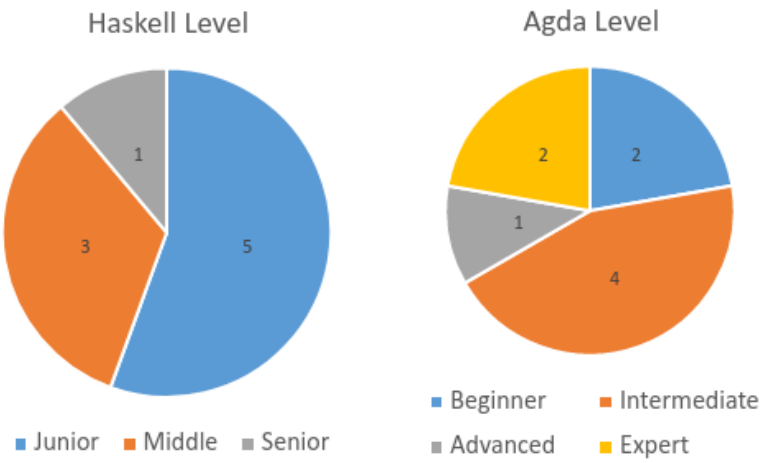

Figure 6.1: The participants' skill levels

[1]The results are available under an MIT License at: `https://doi.org/10.4121/611fd9e6-a7bb-47d7-9afe-34efe890ddd7`

6.1 Lessons Learned

The participants were asked to complete as many assignments to as much completion as they prefer. Figure 6.2 displays an overview of completion for each exercise. We discuss the numbers and common "mistakes" (i.e. where the solutions diverged from the expected solution) for each assignment.

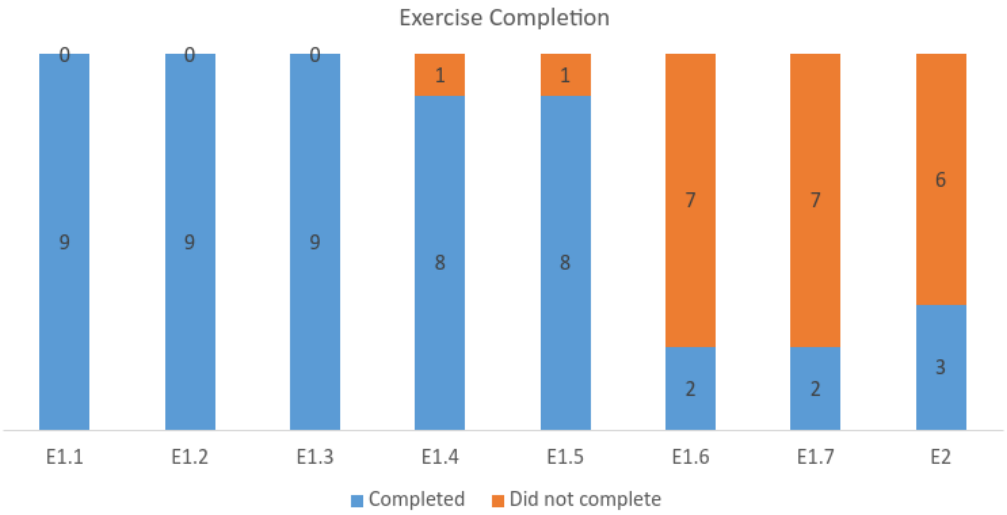

Figure 6.2: The amount of participants that completed each exercise

Assignment 1: The `All` Type

The first assignment (with all sub-assignments included) took participants an average of 80 minutes to complete. Except for the two optional sub-assignments, this assignment was completed by almost every participants in its entirety. The most common mistakes were the following:

- Not compiling `All` to `newtype`. This was not a breaking issue.
- Defining a specific constructor name, different from the type name. This actually caused the Haskell tests to fail, since they expected the same name.
- Implementing the `Eq` instance by hand and only at the more complex type classes realising that `deriving` might be an option.
- Not using the built-in `IsLawfulSemigroup` type class to prove that their `Semigroup` instance was lawful. Many participants just defined and proved this on their own.

Even though the expected solution did not do it (since the feature was buggy at the time of the user study), two participants *did* use the default method type class implementations for defining `Monoid`. Unfortunately, imports were broken and their code did not compile, despite them following the documentation and examples.

The two optional assignments were only completed by two participants (and one implemented Exercise 1.6 accidentally).

Assignment 2: Safe `lookup`

Participants spent an average of 53 minutes trying to implement the second assignment, but only three managed to actually complete it. Among the others, there were several partial solutions which were missing the following:

- Using the `nequality` function of the `IsLawfulEq` type class (or a similar one) to convert between boolean and propositional equality.
- Knowing when and how to use erasure to achieve the correct type signature.
- Attempting to use a helper function but not being able to transpile it to Haskell correctly.
- Not knowing how to include `contains` in the type signature.

There was some concern that this exercise was not representative of how Agda would be used on its own.

6.1.1 The Experience

To understand what challenges come with using AGDA2HS, participants were asked to fill in the best and in-need-of-most improvement aspects that they found about the tool. We present the things that they mentioned below:

The Idea. Multiple participants mentioned that they found the concept of AGDA2HS "great" and "neat".

The Design. There were many comments in relation to the design of AGDA2HS. Participants mentioned the simplicity of the compilation pragmas as well as the clearness of the generated Haskell code and praised them as the best aspects of AGDA2HS. It was noted that Agda knowledge transfers well into working with AGDA2HS. There was also a specific mention of the use of postulates to support instance deriving as a good aspect.

On the other hand, there were complaints about "hacky" solutions when there is no 1-to-1 mapping between an Agda and a Haskell concept. The specific example mentioned was the encoding needed to obtain the same constructor and type name when creating a `newtype` declaration. There were also requests for support for more Prelude methods and concerns about the difficulty of knowing how and when to apply erasure.

The Documentation. Many of the answers mentioned documentation as the most important point of improvement. There were concerns about the completeness as well as comprehensibility. Many participants suggested not only better documentation, but also things such as more detailed explanations, tutorials, and more involved examples.

Only one participant mentioned that the documentation was decent and helpful if the user is already familiar with both programming languages.

The Environment. Although the study was not focused on this, a participant did mention that they would like to see more integration of the tool with other programming environments. The example they provided was propagating errors specific to AGDA2HS to the integrated development environment (IDE) and showing them interactively.

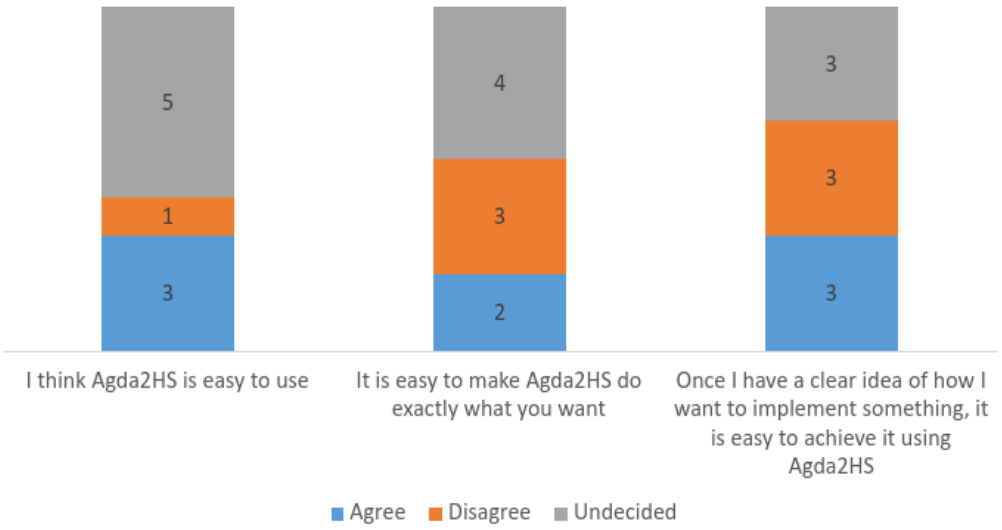

Figure 6.3: How easy participants found AGDA2HS to use

As seen in Figure 6.3, there was a lot of scepticism about the ease of use of AGDA2HS. Many participants were undecided about the issue and the answers

were very evenly distributed. We postulate that this is due to the fact that many people were not familiar with all the Agda concept necessary to complete the programming assignments and so were not able / willing to make an informed judgement.

6.1.2 The Impressions

To answer the second research question, we asked participants questions directed at determining under which circumstances they would find AGDA2HS useful. The primary feeling that appeared within these answers was the concern that in many cases, the cost of using AGDA2HS might not be worth the benefit.

Participants listed two reasons for this concern. Firstly, it takes a long time to actually write proofs that verify your code. This is both because learning to use Agda takes a long time, but also because the proofs themselves can become tedious and long. Secondly, using AGDA2HS requires an understanding of not one, but two programming languages. Not only is this quite a big consideration on its own, but participants also pointed out the combined usage affects developers with background in both programming languages. Haskell developers need to go through the steep learning curve of Agda, and Agda developers have to depart (to a certain extent) from the usual usage of the language.

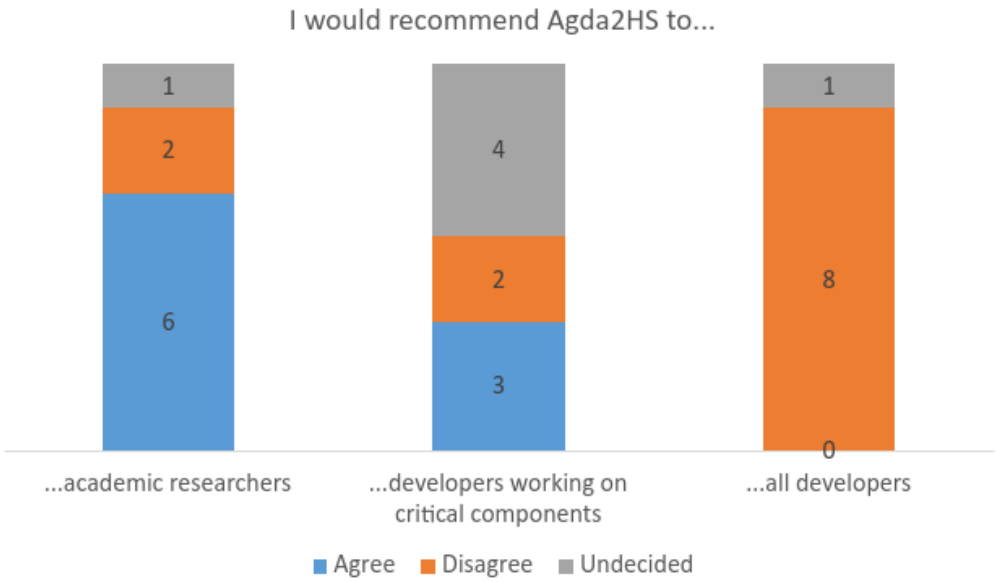

Figure 6.4: Who should use AGDA2HS

When asked specifically who they would recommend AGDA2HS to (visualised in Figure 6.4), the general consensus was that it is great for academic researchers but definitely not a tool for everyone. There was a certain degree of agreement on AGDA2HS as a tool for developers of critical components and standalone libraries, but many of the participants were not sure about their opinion in this matter. Fortunately, as seen in Figure 6.5, no participant disagreed with the fact that AGDA2HS *is* a useful tool. This shows that if some of the big costs of using AGDA2HS can be overcome, the tool might spike interest in more domains than is currently apparent.

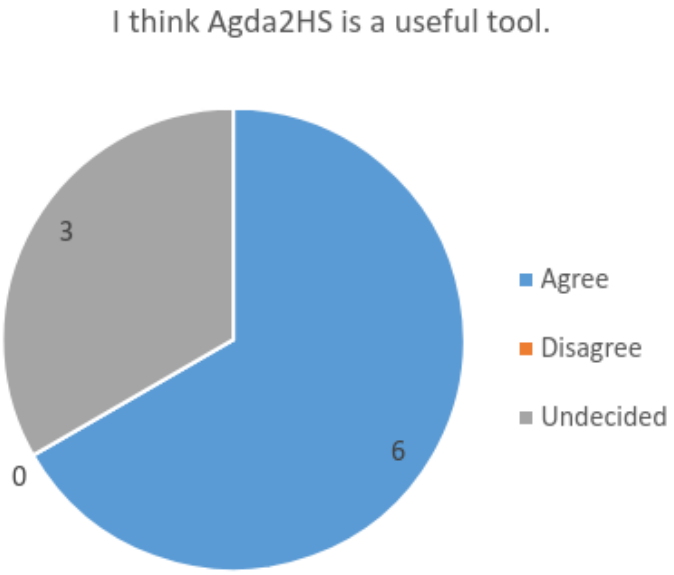

Figure 6.5: Is AGDA2HS useful

6.1.3 The Technical Reflection

To evaluate the validity of our hypothesis as well as the implementation work done in this thesis, participants were asked to give opinions on the features necessary to make AGDA2HS usable and accessible. The results of this are visualised in Figures 6.6 and 6.7.

What we see from these graphs is that most participants agree with the hypothesis on the necessity of the chosen features. They wanted to see commonly-used Haskell features made available, especially those which significantly reduce the workload of the developer (such as deriving type class instances).

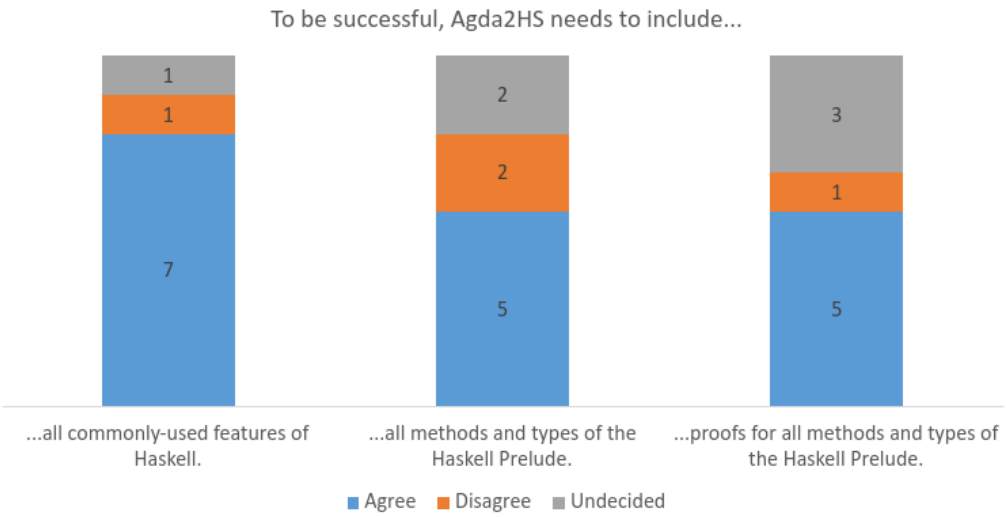

Figure 6.6: General features necessary for the success of AGDA2HS

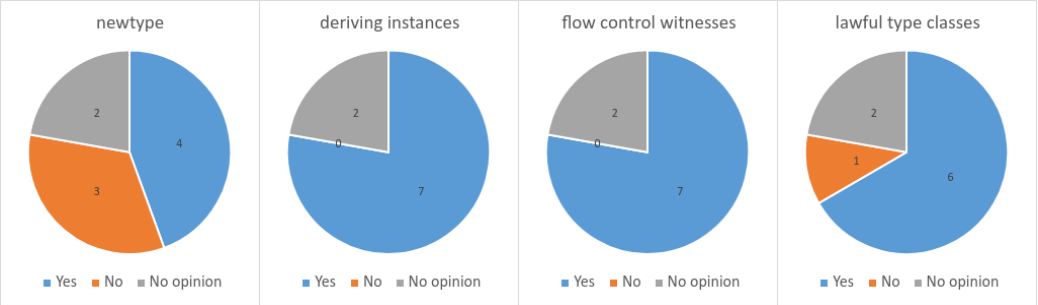

Figure 6.7: Specific features necessary for the success of AGDA2HS

More than half of the participants also agreed that the methods, types, and proofs of the Haskell Prelude are an essential part of AGDA2HS. In the open questions, we did not see any significant Haskell or Agda features missing other than the `with` abstraction which allows you to use intermediate computations to control the flow of your program.

The one outlier is the `newtype` feature, where participants were divided on its necessity. We assume this is because the `newtype` is an alternative to the `data` declaration and is thus not crucial to the success. However, we believe that when efficiency and other metrics start coming into play, `newtype` will become necessary due to its reduction in constructor overhead at runtime.

Chapter 7

Discussion

Based on the results of the user study, we identify three main requirements for the usability of AGDA2HS:

1. **Common Haskell language features and features that facilitate verification should be available.** This was reflected throughout the results of the user study, with participants reacting positively to features that made using AGDA2HS easier - whether by offering default solutions, such as with using postulates for deriving of type class instances, or by offering simple ways to obtain information necessary to complete a proof, such as with flow control witnesses. Making common Haskell language features available allows users to carry over their expertise while features that facilitate verification can help guide them into new territory. Both of these ease the transition to using a new tool.

2. **Documentation should be complete, searchable, and progressive.** Even though we did not hypothesise about documentation, its importance became evident during the user study. Participants performed worse on implementing features that were less documented, and it was clear that they were often unsure of where to look for features they might need to solve a problem. Especially with the complexity of using all the components of AGDA2HS, providing users with documentation, tutorials, and guiding them to good solutions is a crucial part of making this tool usable.

3. **The tool should be integrated into the ecosystem of the target language.** Though this might not be relevant before the previous two points are addressed to a sufficient degree, it is important to start considering how AGDA2HS will work within the ecosystems it wants to be a part of.

This refers to all sorts of development operations, such as how to include the code in a repositories and existing workflows, how to combine code generated by AGDA2HS with pure Haskell code, how to incorporate AGDA2HS into IDEs, etc. It also refers to the ease with which users can install and begin working with the tool. The more we can pave the way for developers to be able to use it out of the box, the more accessible it will become.

7.1 Threats to Validity

Especially because this was a prototype user study, there are some threats to validity to consider. Perhaps the most significant of those is the number and distribution of participants. This study was conducted on a very small scale with a population of students from a single university. Many of these took the same course on both Haskell and Agda programming and have worked with the same tools. This indicates that the results are probably not representative of the general target population, which includes developers in industry as well as academic researchers.

The second thing to consider were the assignments. First of all, they were tailored to the features that have been added and so were largely biased towards evaluating what was already improved rather than the tool as a whole. Secondly, they are small assignments of the like that one might find in course materials with a usage of limited library methods. AGDA2HS is meant to be used within large ecosystems and as such a more valid user study would include assignments where participants are asked to use it within a larger project context with more dependencies.

Finally, since this user study was a sort of "pilot", we were not sure what to expect and so we were present in the room with participants, helping with errors that hindered their progress (e.g. installation issues). This can seriously affect the validity of a large-scale study and so the results here should be taken with a grain of salt, being considered as a first step in the right direction.

7.2 Future Work

As with all research, there is plenty to be explored and solved in the future. We would like to take this section to indicate the most interesting trajectories to explore.

Firstly, we believe conducting research about the best ways to provide documentation for formal verification tools would be a great next step for this field.

As was also noted so often during the user study, not only is documentation important, it can provide support in overcoming the challenge of learning to work with formal verification tools. Therefore, finding the least overwhelming and most helpful form of tool presentation would be a logical next step for accessibility. We imagine that this research could take inspiration from existing research on software engineering practices as well as educational tools.

Secondly, we suggest future work in integration with the existing ecosystems. This is not only applicable to tools such as AGDA2HS that use a combination of programming languages, but also to formal verification languages themselves. The usability of a tool is limited if collaboration within developer teams becomes overly complex due to it. Important things to consider here are installation, versioning, version control, and continuous integration. First steps could include conventions and guidelines for how to work with these tools within their ecosystems.

We already touched upon integrating AGDA2HS into a larger project during the design of the user study, when making the assignments. Creating a project that always compiles *and* works well with version control was quite a challenge, since AGDA2HS generates code. The solution we ended up with required default solution files and a bash script which only worked on Linux-based systems. This is quite problematic, since collaboration among multiple platforms would be impossible. Providing a project structure and default environment for such projects is therefore an import aspect to consider and design.

Lastly, more technical future work could focus on the transpilation process. AGDA2HS does not remember the state of its transpilation, instead resolving each pragma separately. This is why, for example, it is only possible to transpile postulates into *standalone* derivations instead of being able to attach them to the data definition. Preserving the state and remembering the context and scope of the transpilation could offer interesting possibilities for tools such as AGDA2HS.

7.3 Recommendations for Future User Studies

To conclude this section, we would like to suggest a more user-oriented approach to creating "accessible" designs for tools such as AGDA2HS. Even while we were preparing the assignments for the user study, we noticed crucial documentations and features missing and updated the tool accordingly. We believe that user studies and a user-driven approach are key in making AGDA2HS accessible to a wider audience. To that end, we would like to provide a few recommendations for future user studies:

- **Do not underestimate your own knowledge of the tool.** The assignments that we designed turned out to be far more complex and took much more time than we expected. Even if something might seem simple to you as the developer, it might not be the case for someone who has not implemented the needed features.
- **Make sure the skill level of your participants matches the goal of your study.** Complementary to the point above, having the correct participants can help with the quality of your results. While our user study was only a small prototype, it was already obvious that some questions were too hard for participants with less skill to answer objectively.
- **Assume that people will not read instructions and documentation carefully.** Especially in an environment where you are short on time, make sure that there is not an overload of things for participants to read and understand before they can participate. Create a list of things participants should open and have available for reference while they are working with your tool and make sure they can find everything they need *easily*.
- **Ask open questions with short answers.** While choice questions are quicker to answer, they will always have a bias towards only answering questions about things you have already thought about. Giving participants the option to express some of their own thoughts might bring new insights that you did not even consider. Be sure to keep the questions specific enough that participants do not lose motivation to answer them.
- **If your tool has a complicated installation process, provide your own machine with a correct setup.** Even though we provided users with a virtual machine as well as instructions for a personal installation, many still took a long time to set everything up correctly. To ease the experience, make sure to have a setup where a participant can arrive and start programming immediately. (This is only applicable, of course, if the installation process is *not* part of what you want to study.)

Chapter 8

Conclusion

In this thesis, we aimed to find a way to make formal verification accessible in widespread programming language ecosystems. We studied AGDA2HS, a tool that transpiles Agda to Haskell, and determined what challenges users face when working with it. We implemented four features meant to improve the usability of AGDA2HS, among which the most notable was a solution to witnessing flow control branch conditions when their proof is necessary. To make sure that these features were accessible, we added detailed documentation for them to the AGDA2HS documentation page.

In order to evaluate the usability of the improved AGDA2HS, we designed and conducted a pilot user study - the first of its kind for Agda. This user study is a first step towards user-driven design for tools such as AGDA2HS, providing initial insight, but also recommendations and designs for future user studies. Its results also show that there is still a long way to go and we hope to motivate future work in the area.

To conclude this thesis, we answer the general research questions based on our findings from the AGDA2HS usability study. To make formal verification accessible within ecosystems of widespread programming languages, we studied a tool that allows users to write their code in a formal verification language and then transpiles it into a widespread programming language with a big ecosystem. This allows the user to harness the full powers of formal verification, while still being able to reap the benefits of popular frameworks and libraries. To make such a tool accessible, we find that the following things are necessary:

- detailed documentation, with accompanying tutorials and involved examples,
- support for features commonly used in the widespread programming language,

- a library of proofs for the standard library methods and other tools to facilitate verification,
- the integration of the tool into the ecosystem it aims to benefit from,
- and a user-driven development based on future user studies.

References

[1] M. Aniche, *Effective Software Testing: A developer's guide*. Shelter Island: Manning Publications Co., 2022.

[2] E. M. Clarke, "Model checking," in *Foundations of Software Technology and Theoretical Computer Science*, Springer Berlin Heidelberg, 1997, pp. 54–56, ISBN: 978-3-540-69659-9.

[3] P. Cousot and R. Cousot, "Abstract Interpretation: A Unified Lattice Model for Static Analysis of Programs by Construction or Approximation of Fixpoints," ser. POPL '77, Association for Computing Machinery, 1977, pp. 238–252, ISBN: 9781450373500. DOI: 10.1145/512950.512973.

[4] J. Harrison, J. Urban and F. Wiedijk, "History of Interactive Theorem Proving," in Dec. 2014, vol. 9, pp. 135–214, ISBN: 9780444516244. DOI: 10.1016/B978-0-444-51624-4.50004-6.

[5] A. Bove, P. Dybjer and U. Norell, "A Brief Overview of Agda – A Functional Language with Dependent Types," in *Theorem Proving in Higher Order Logics*, Berlin, Heidelberg: Springer Berlin Heidelberg, 2009, pp. 73–78, ISBN: 978-3-642-03359-9.

[6] The Coq Team. "Early history of Coq." (1995), [Online]. Available: https://coq.inria.fr/refman/history.html#id1.

[7] E. Body, "Idris, a general-purpose dependently typed programming language: Design and implementation," *Journal of Functional Programming*, vol. 23, Sep. 2013. DOI: 10.1017/S095679681300018X.

[8] G. F. Kadoda, R. G. Stone and D. Diaper, "Desirable features of educational theorem provers - a cognitive dimensions viewpoint," in *Annual Workshop of the Psychology of Programming Interest Group*, 1999.

[9] J. Cockx, O. Melkonian, L. Escot, J. Chapman and U. Norell, "Reasonable Agda is Correct Haskell: Writing Verified Haskell Using Agda2HS," in *Proceedings of the 15th ACM SIGPLAN International Haskell Symposium*, ser. Haskell 2022, Association for Computing Machinery, 2022, pp. 108–122, ISBN: 9781450394383. DOI: 10.1145/3546189.3549920.

[10] N. Vazou, J. Breitner, R. Kunkel, D. Van Horn and G. Hutton, "Theorem Proving for All: Equational Reasoning in Liquid Haskell (Functional Pearl)," *SIGPLAN Not.*, vol. 53, no. 7, pp. 132–144, Sep. 2018, ISSN: 0362-1340. DOI: 10.1145/3299711.3242756.

[11] P. Dybjer, Q. Haiyan and M. Takeyama, "Verifying Haskell programs by combining testing, model checking and interactive theorem proving," *Information and Software Technology*, vol. 46, no. 15, pp. 1011–1025, 2004, Third International Conference on Quality Software: QSIC 2003, ISSN: 0950-5849. DOI: https://doi.org/10.1016/j.infsof.2004.07.002.

[12] K. Claessen and J. Hughes, "QuickCheck: A Lightweight Tool for Random Testing of Haskell Programs," *SIGPLAN Not.*, vol. 35, no. 9, pp. 268–279, Sep. 2000, ISSN: 0362-1340. DOI: 10.1145/357766.351266.

[13] H. Carr, C. Jenkins, M. Moir, V. C. Miraldo and L. Silva, *An approach to translating Haskell programs to Agda and reasoning about them*, 2022. DOI: 10.48550/ARXIV.2205.08718.

[14] J. Christiansen, S. Dylus and N. Bunkenburg, "Verifying Effectful Haskell Programs in Coq," ser. Haskell 2019, Association for Computing Machinery, 2019, pp. 125–138, ISBN: 9781450368131. DOI: 10.1145/3331545.3342592.

[15] G. Allais, *Builtin Types viewed as Inductive Families*, 2023. DOI: 10.48550/ARXIV.2301.02194.

[16] A. Abel, M. Benke, A. Bove, J. Hughes and U. Norell, "Verifying Haskell Programs Using Constructive Type Theory," in *Proceedings of the 2005 ACM SIGPLAN Workshop on Haskell*, ser. Haskell '05, Association for Computing Machinery, 2005, pp. 62–73, ISBN: 159593071X. DOI: 10.1145/1088348.1088355.

[17] A. Spector-Zabusky, J. Breitner, C. Rizkallah and S. Weirich, "Total Haskell is Reasonable Coq," in *Proceedings of the 7th ACM SIGPLAN International Conference on Certified Programs and Proofs*, ser. CPP 2018, Association for Computing Machinery, 2018, pp. 14–27, ISBN: 9781450355865. DOI: 10.1145/3167092.

[18] C. Paulin-Mohring and B. Werner, "Synthesis of ML programs in the system Coq," *Journal of Symbolic Computation*, vol. 15, no. 5, pp. 607–640, 1993, ISSN: 0747-7171. DOI: https://doi.org/10.1016/S0747-7171(06)80007-6.

[19] P. Letouzey, "A New Extraction for Coq," in *Types for Proofs and Programs*, Springer Berlin Heidelberg, 2003, pp. 200–219, ISBN: 978-3-540-39185-2.

[20] The Coq Team. "Program extraction." (2002), [Online]. Available: https://coq.inria.fr/refman/addendum/extraction.html.

[21] The Agda Team. "Pragmas." (2019), [Online]. Available: https://agda.readthedocs.io/en/v2.6.0.1/language/pragmas.html.

[22] The Agda Team. "Automatic Proof Search (Auto)¶." (2023), [Online]. Available: https://agda.readthedocs.io/en/v2.6.3/tools/auto.html.

[23] N. Vazou, "Liquid Haskell: Haskell as a Theorem Prover," Ph.D. dissertation, University of California, San Diego, 2016.

[24] H. Blanchette, N. Vazou and L. Lampropoulos, "Liquid Proof Macros," in *Proceedings of the 15th ACM SIGPLAN International Haskell Symposium*, ser. Haskell 2022, Association for Computing Machinery, 2022, pp. 27–38, ISBN: 9781450394383. DOI: 10.1145/3546189.3549921.

[25] L. Escot and J. Cockx, "Practical Generic Programming over a Universe of Native Datatypes," *Proc. ACM Program. Lang.*, vol. 6, no. ICFP, Aug. 2022. DOI: 10.1145/3547644.

[26] P. Wadler, "Monads for functional programming," in *Program Design Calculi*, Berlin, Heidelberg: Springer Berlin Heidelberg, 1993, pp. 233–264, ISBN: 978-3-662-02880-3. DOI: 10.1007/978-3-662-02880-3_8.

[27] A. Abel, "foetus - Termination Checker for Simple Functional Programs," 1998. [Online]. Available: https://www.cse.chalmers.se/~abela/foetus.pdf.

[28] "Newtype." (2016), [Online]. Available: https://wiki.haskell.org/Newtype.

[29] The Agda Team. "Postulates." (2023), [Online]. Available: https://agda.readthedocs.io/en/v2.6.0.1/language/postulates.html.

[30] GHC Team. "6.6.7. Deriving strategies." (2023), [Online]. Available: https://ghc.gitlab.haskell.org/ghc/doc/users_guide/exts/deriving_strategies.html.

[31] D. Aspinall, "Subtyping with Singleton Types," in *Selected Papers from the 8th International Workshop on Computer Science Logic*, ser. CSL '94, Springer-Verlag, 1994, pp. 1–15, ISBN: 3540600175.

[32] D. Devriese and F. Piessens, "On the Bright Side of Type Classes: Instance Arguments in Agda," in *Proceedings of the 16th ACM SIGPLAN International Conference on Functional Programming*, ser. ICFP '11, Association for Computing Machinery, 2011, pp. 143–155, ISBN: 9781450308656. DOI: 10.1145/2034773.2034796.

[33] M. Al-hassy, J. Carette and W. Kahl, "A Language Feature to Unbundle Data at Will (Short Paper)," in *Proceedings of the 18th ACM SIGPLAN International Conference on Generative Programming: Concepts and Experiences*, ser. GPCE 2019, Association for Computing Machinery, 2019, pp. 14–19, ISBN: 9781450369800. DOI: 10.1145/3357765.3359523.

[34] C. S. T. Catarina Gamboa Paulo Santos and A. Fonseca, "Usability-Oriented Design of Liquid Types for Java," ICSE 2023 at Conf Researchr, 2023. [Online]. Available: https://catarinagamboa.github.io/papers/icse-preprint-964.pdf.

[35] B. Beckert and S. Grebing, "Evaluating the usability of interactive verification systems," *CEUR Workshop Proceedings*, vol. 873, pp. 3–17, Jan. 2012.

[36] J. Kirakowski. "The Use of Questionnaire Methods for Usability Assessment." (2021), [Online]. Available: https://sumi.uxp.ie/about/sumipapp.html.

Appendices

A Lawful Eq

```
1  eqReflexivity : {{ iEq : Eq e }} → {{ IsLawfulEq e }}
2    → ∀ (x : e) → (x == x) ≡ True
3  eqReflexivity x = equality' x x refl
4
5  eqSymmetry : {{ iEq : Eq e }} → {{ IsLawfulEq e }}
6    → ∀ (x y : e) → (x == y) ≡ (y == x)
7  eqSymmetry x y with x == y in eq
8  ... | True  = sym (equality' y x (sym (equality x y eq)))
9  ... | False = sym (nequality' y x (λ qe →
10                   (nequality x y eq) (sym qe)))
11
12 eqTransitivity : {{ iEq : Eq e }} → {{ IsLawfulEq e }}
13   → ∀ (x y z : e) → ((x == y) && (y == z)) ≡ True
14   → (x == z) ≡ True
15 eqTransitivity x y z h
16   = equality' x z (trans
17     (equality x y (&&-leftTrue (x == y) (y == z) h))
18     (equality y z (&&-rightTrue (x == y) (y == z) h)))
19
20 eqExtensionality : {{ iEq : Eq e }} → {{ IsLawfulEq e }}
21   → {{ iEq : Eq a }} → {{ iLawfulEq : IsLawfulEq a }}
22   → ∀ ( x y : e ) ( f : e → a ) → (x == y) ≡ True
23   → (f x == f y) ≡ True
24 eqExtensionality x y f h =
25   equality' (f x) (f y) (cong f (equality x y h))
26
27 eqNegation : {{ iEq : Eq e }} → {{ IsLawfulEq e }}
28   → ∀ { x y : e } → (x /= y) ≡ not (x == y)
29 eqNegation = refl
```

Figure A.1: The Laws for the Eq Type Class

Figure A.1 displays the boolean equality proofs for the `Eq` type class. It uses the conversions from boolean to propositional equality that we obtain from the `IsLawfulEq` type class. These are in turn obtained by using the `Reflects` idiom (defined in Figure A.2) to define the relationship between boolean and propositional equality.

```
1  data Reflects {p} (P : Set p) : Bool → Set p where
2    ofY : ( p :   P )      → Reflects P True
3    ofN : ( np : (P → ⊥) ) → Reflects P False
4
5  private
6    variable
7      p : Level
8      P : Set p
9
10 of : ∀ {b} → if b then P else (P → ⊥) → Reflects P b
11 of {b = False} np = ofN np
12 of {b = True }  p = ofY p
13
14 invert : ∀ {b} → Reflects P b → if b then P else (P → ⊥)
15 invert (ofY  p) = p
16 invert (ofN np) = np
17
18 extractTrue : ∀ { b } → {{ true : b ≡ True }}
19   → Reflects P b → P
20 extractTrue (ofY p) = p
21
22 extractFalse : ∀ { b } → {{ true : b ≡ False }}
23   → Reflects P b → (P → ⊥)
24 extractFalse (ofN np) = np
```

Figure A.2: The Definition of the `Reflects` Idiom

B User Study: Recruitment Leaflet

Come *prove* your Haskell skills!

Sign up here:

https://forms.office.com/e/1jM7LkTCnV

What?

We are conducting a (small) user study on how accessible **Agda2HS** is to Haskell programmers.

We will ask you to:
1. Implement a few exercises in Agda,
2. Use Agda2HS to compile the code to Haskell,
3. Fill in a questionnaire to rate your experience.

You can check out Agda2HS at here:
https://github.com/agda/agda2hs

When? April 25
for 1-3 hours
between 9:00 – 17:00

Where? 4.W950 Shannon
Building 28
Van Mourik Broekmanweg 6

What skills do I need?

- Some experience in Haskell programming
- Limited experience with Agda

If you've taken the CSE3100 Functional Programming course, you are the perfect candidate.

Contact: s.juhosova@student.tudelft.nl

C User Study: Informed Consent Statement

You are being invited to participate in a research study titled "A Usability Study of Agda2HS". This study is being done by Sára Juhošová (MSc student) from the TU Delft under the leadership of Dr. Jesper Cockx.

The purpose of this research study is to determine how easy-to-use and useful potential users find tools such as Agda2HS, and will take you approximately 120 minutes to complete. The data will be used for the evaluation section of a Master's thesis, and to further improve Agda2HS. We will be asking you to complete a few small programming exercises in Agda2HS and then rate your experience in an anonymous survey.

As with any online activity the risk of a breach is always possible. To the best of our ability your answers in this study will remain confidential. We will minimize any risks by (a) not asking for any personal information in the questionnaire where you will rate your experience and (b) using MS Office Forms for the collection, a tool recommended and approved by the TU Delft. The sign-up form will ask you for your e-mail address, but this data will only be used to communicate information closer to the date of the study and will not be linked to the answers given during the study. The e-mail addresses will be deleted after the study is complete.

Your participation in this study is entirely voluntary and you can withdraw at any time. You are free to omit any questions. Since the questionnaires will be anonymous, it will not be possible to remove your answers within a given timescale.

For more information, please contact Sára Juhošová (<e-mail>).

By agreeing to the informed consent statement, you are willing to proceed with the user study and allow for the anonymised data to be archived and shared publicly under an MIT License.

You will now be asked to fill in your contact information. **This information is only collected for contacting you with the details of the user study and setting up a proper programming environment.**

D User Study: Participation Email

Dear participant,

Thank you for participating in the Agda2HS user study!

Date: 25 April 2023
Time: Arrival anytime between 9:30 and 15:00 (the entire study will take about 1-2 hours)
Building: Building 28 (Van Mourik Broekmanweg 6)
Room: 4.W950 Shannon

In case you would like to already read something about Agda2HS, you can check out the following documents:

1. The original paper about Agda2HS: `https://dl.acm.org/doi/10.1145/3546189.3549920`
2. The Agda2HS GitHub repository: `https://github.com/agda/agda2hs`
3. The Agda2HS docs page: `https://agda.github.io/agda2hs/`

To speed up the process, you can already prepare the coding environment. You may also do so once you arrive on location. We have prepared a Virtual Machine (running Debian) which has everything necessary pre-installed. This is to avoid any problems with installation, especially since installing and compiling Agda can take a long time. Instructions on how to prepare the virtual machine are below.

In case you wish to use your own machine, there are instructions included on how you can set everything up. We recommend doing this only with Linux-based operating systems, since we make use of make files.

Using the Virtual Machine

1. Download the virtual machine prepared for this study here: `https://surfdrive.surf.nl/files/index.php/s/8OXGrqv8100rlKP`
2. Import it into your virtualisation software (probably VirtualBox)
3. Start the machine and login to the Agda2HS profile (password: `a2hs`)
4. In case your resolution is weird, you can try changing your VM graphics controller. In VirtualBox, you can do this by navigating to `Settings > Display` and selecting “VBoxVGA”.

5. Open Visual Studio code. This should already open into the example project. If not, you can open the `agda2hs-user-study-minimal` project in the `Documents` directory.
6. Open the `lib/Reverse.agda` file and press `Ctrl + c, Ctrl + l` - this should successfully type-check the Agda file.
7. Run `make` - this should successfully execute the QuickCheck tests.

Preparing the environment on your own machine

1. Follow the instructions on the Agda website to install the latest Agda version: `https://agda.readthedocs.io/en/v2.6.3/getting-started/installation.html`
2. Follow the installation procedure for Agda2HS: `https://agda.github.io/agda2hs/introduction.html#installation` (install it using the master branch of the GitHub repository)
3. Run `make` in the Agda2HS repository.
4. Make sure you have an editor ready to allow you to work with Agda2HS during the user study. The recommendation is to use Visual Studio Code with the `agda-mode` extension.
5. Verify that everything is working by downloading the example project of this study: `https://tinyurl.com/a2hs-study`
   (a) Run `cabal new-install -overwrite-policy=always` in the repository Open the project in your editor. Open the `lib/Reverse.agda` file and press `Ctrl + c, Ctrl + l` - this should successfully type-check the Agda file.
   (b) Run `make` - this should successfully execute the QuickCheck tests.

Looking forward to seeing you!

E User Study: Solutions Compiled to Haskell

This appendix contains the intended solutions of the user study compiled to Haskell. Figure E.3 displays the compiled version of Figure 5.1 and Figure E.4 displays the compiled version of Figure 5.3.

```haskell
{-# LANGUAGE StandaloneDeriving #-}
{-# LANGUAGE DeriveGeneric #-}

module All where

import GHC.Generics

newtype All = All{getAll :: Bool}
                deriving (Read, Generic)

deriving instance Eq All
deriving instance Ord All
deriving instance Show All
deriving instance Bounded All

instance Semigroup All where
    a <> b = All (getAll a && getAll b)

instance Monoid All where
    mempty = All True
    mappend = (<>)
    mconcat [] = mempty
    mconcat (x : xs) = x <> mconcat xs
```

Figure E.3: User Study Assignment 1 Haskell Solution

```haskell
module Lookup where

lookupSafe :: Eq a => a -> [(a, b)] -> b
lookupSafe key (x : xs)
  = if fst x == key then snd x else lookupSafe key xs
```

Figure E.4: User Study Assignment 2 Haskell Solution

F User Study: Questionnaire

This appendix contains a copy of the questions asked in the "Agda2HS Usability Study" questionnaire, used on April 25, 2023.

Welcome

Welcome to the Agda2HS User Study. Thank you for participating!

To begin the user study, download the actual project here: `https://tinyurl.com/a2hs-us`

Once the project is downloaded, open it in your favourite Agda editor and carefully read the README.

There are two exercises, you may finish as many of them to as much completeness as you wish. Please time how long each exercise takes you, you will be asked to fill it in on the next page of this survey.

Below are some useful links:

- The original paper about Agda2HS: `https://dl.acm.org/doi/10.1145/3546189.3549920`
- The Agda2HS GitHub repository: `https://github.com/agda/agda2hs`
- The Agda2HS docs page: `https://agda.github.io/agda2hs/`

Your Solution

1. You should have received an identifier before you started filling in this questionnaire.

   To upload your solution, please compress your lib directory into an archive (ZIP) and name the resulting file `<YOUR_IDENTIFIER>.zip`.

   Upload your archive to: `INSERT LINK HERE`

   Enter the identifier below:

   [ ]

2. Which exercises did you complete?
   - ☐ Exercise 1.1 Define the `All` type
   - ☐ Exercise 1.2 Create an instance for `Eq`, `Ord`, `Show`, and `Bounded`
   - ☐ Exercise 1.3 Create an instance for `Semigroup`
   - ☐ Exercise 1.4 Prove that the `Semigroup` instance is lawful
   - ☐ Exercise 1.5 Create an instance for `Monoid`
   - ☐ Exercise 1.6 (Optional) Create an instance for `Read`
   - ☐ Exercise 1.7 (Optional) Create an instance for `Generic`
   - ☐ Exercise 2 Implement a `lookupSafe` function

3. How long did (the entire) Exercise 1 take you (in minutes)?

   3. _____

4. How long did Exercise 2 take you (in minutes)?

   4. _____

5. Did you look at any Agda2HS documentation before the study (e.g. the docs page, the repository, or the published paper)? ☐ Yes ☐ No

Your Skills

6. What kind of experience do you have with **Haskell**?
   - ☐ I've taken a course in Haskell programming
   - ☐ I've used Haskell in small personal projects
   - ☐ I've used Haskell in academic research
   - ☐ I've used Haskell for a big project (e.g. in industry)
   - ☐ I've used various Haskell frameworks and libraries
   - ☐ Other: _____
7. Which **Haskell** features are you comfortably familiar with?
   - ☐ Lambdas, functions, compositions
   - ☐ Base types (String, Int, Float...)
   - ☐ Main ADTs (Maybe, Bool, Either...)
   - ☐ Basic IO in the IO monad
   - ☐ Lists, tuples
   - ☐ Algebraic Data Types (BinTree, Expr, ...)
   - ☐ Pattern Matching
   - ☐ Basic type classes (Show, Eq, Ord)
   - ☐ Recursion
   - ☐ Base high-order functions (map, filter...)
   - ☐ Base collections (List, Map...)
   - ☐ Advanced composition (point free, applicatives, functors, monads)
   - ☐ General types (Text, ByteString...)
   - ☐ Basic lenses
   - ☐ Main monads (State, Reader, Writer, List, Either, Maybe)
   - ☐ Monad transformers & monad stacks
   - ☐ Advanced IO Advanced type classes (Foldable, Traversable...)
   - ☐ Higher-kinded types
   - ☐ Some type-level features usage (not implementation): Generics, GADTs, FunDeps, Type Families, HKDs...
   - ☐ Template Haskell (TH), Foreign Function Interface (FFI)
   - ☐ Parametrized types
   - ☐ Functional Reactive Programming (FRP)
   - ☐ Concurrency (STM, MVar, threads)
   - ☐ Laziness & its implications
8. What kind of experience do you have with **Agda**?
   - ☐ I've taken a course in Agda programming
   - ☐ I've used Agda for a bigger school project

- ☐ I've used Agda in academic research
- ☐ I've used Agda in various projects
- ☐ I have never used Agda, but I have experience with other formal verification tools
- ☐ Other: _____

9. Which **Agda** features are you comfortably familiar with?
   - ☐ Functions (incl. type definitions and pattern matching)
   - ☐ Built-in basic types
   - ☐ Algebraic Data Types
   - ☐ Record Types
   - ☐ Mix-fix operators
   - ☐ Co-patterns
   - ☐ Postulates
   - ☐ Implicit arguments
   - ☐ Instance arguments
   - ☐ Irrelevance
   - ☐ Erasure
   - ☐ Pragmas
   - ☐ Equational Reasoning

Your Experience

Is it hard to use Agda2HS?

10. Please answer all of the questions below.

    In checking the left or right box you are not necessarily indicating strong agreement or disagreement but just your general feeling most of the time.

	Agree	Undecided	Disagree
I didn't know how to start using Agda2HS.	☐	☐	☐
There is too much to read before you can use Agda2HS.	☐	☐	☐
I keep having to go back to look at the documentation and examples.	☐	☐	☐
The documentation included everything I needed.	☐	☐	☐
The documentation was very clear.	☐	☐	☐
Error messages provided by Agda2HS are not adequate.	☐	☐	☐
Agda2HS allows me to do everything that I would want to do if I used only Haskell.	☐	☐	☐
I think Agda2HS is easy to use.	☐	☐	☐
It is easy to make Agda2HS do exactly what you want.	☐	☐	☐
Once I have a clear idea of how I want to implement something, it is easy to achieve it using Agda2HS.	☐	☐	☐

11. With respect to easy-of-use and understanding, what do you think is the best aspect of Agda2HS, and why?

12. With respect to easy-of-use and understanding, what do you think needs most improvement in Agda2HS, and why?

Your Impressions

Are developers open to using (tools such as) Agda2HS? Do they think it is useful?

13. Please answer all of the questions below.

In checking the left or right box you are not necessarily indicating strong agreement or disagreement but just your general feeling most of the time.

	Agree	Undecided	Disagree
I would recommend Agda2HS to academic researchers.	☐	☐	☐
I would recommend Agda2HS to developers working on critical components.	☐	☐	☐
I would recommend Agda2HS to all developers.	☐	☐	☐
I enjoy using Agda2HS.	☐	☐	☐
Working with Agda2HS is satisfying.	☐	☐	☐
Working with Agda2HS is mentally stimulating.	☐	☐	☐
I would not like to use Agda2HS every day.	☐	☐	☐
Using Agda2HS is frustrating.	☐	☐	☐
Agda2HS is really very awkward.	☐	☐	☐
I think Agda2HS is a useful tool.	☐	☐	☐
I would be open to using Agda2HS for my projects.	☐	☐	☐
I think being able to use intrinsic verification is a great benefit of Agda2HS.	☐	☐	☐

14. In which cases do you think Agda2HS would be useful? For whom would it be useful?

15. Are there any reasons why using Agda2HS would not be useful?

Some Technical Reflection

Which constructs and features are necessary to make Agda2HS usable?

16. Which of the following Agda2HS language features did you use?
    - ☐ deriving type class instances using postulates or the derive pragma
    - ☐ lawful type classes
    - ☐ the `newtype` pragma
    - ☐ witnesses of `if_then_else_` and `case_of_` branches
    - ☐ None of the above
17. Which of the following Agda2HS do you think are necessary for the success of the tool?
    - ☐ deriving type class instances using postulates or the derive pragma
    - ☐ lawful type classes
    - ☐ the `newtype` pragma
    - ☐ witnesses of `if_then_else_` and `case_of_` branches
    - ☐ None of the above
18. Are there any **Haskell** language features you were missing in Agda2HS?

19. Are there any **Agda** features you were missing when writing the code compiled by Agda2HS?

20. Please answer all of the questions below.

    In checking the left or right box you are not necessarily indicating strong agreement or disagreement but just your general feeling most of the time.

	Agree	Undecided	Disagree
Supporting all commonly-used features of Haskell is necessary for the success of Agda2HS.	☐	☐	☐
Including all the methods and types of the Haskell Prelude in Agda2HS is necessary for the success of Agda2HS.	☐	☐	☐
Including proofs for all the methods and types of the Haskell Prelude in Agda2HS is necessary for the success of Agda2HS.	☐	☐	☐

Anything else?

21. Is there anything else you would like to mention about your experience with Agda2HS?

Thank you!

G Results: Participants' Skill Levels

To evaluate the participants' skill levels in Haskell, we used the Haskell Competency Matrix available online. We created a multi-select question with all the features from the matrix and asked participants to indicate which features they were "comfortably familiar with".

We assigned a certain level to the participant if they were familiar with at least 50% of the features in that level as well as at least 80% familiar with all features from the lower levels.

For Agda, we designed our own classification based on known features. A competency matrix such as the one for Haskell was not available, so we sorted participants into four categories based on the total percentage of features they were comfortably familiar with. Participants that were familiar with more than 85% of all features were marked as *Expert*, participants familiar with more than 50% of all features were marked as *Advanced*, and participants familiar with more than 30% of all features were marked as *Intermediate*. Participants familiar with less than 30% of all features were marked as *Beginner*.