Agda for the masses: agda2hs-based libraries in real-world programs

For the Scientific Students' Circle Conference of May 2024
at the Faculty of Informatics

Viktor Csimma

Supervised by Ambrus Kaposi

Eötvös Loránd University
Faculty of Informatics
Department of Programming Languages and Compilers

Eötvös József College

Budapest, 2024

Abstract

Applications with Agda backends? Agda has always been a promising opportunity to write computer-verified programs with a clean, Haskell-like syntax. Still, its slowness and inflexibility has meant a serious obstacle to real-world usage. With the arrival of the agda2hs compiler, it has finally become possible to use the language in ordinary user-oriented software.

Despite this, there has been no serious attempt to grab this opportunity. My goal is to demonstrate that one can indeed use agda2hs-based libraries as backends (even for a GUI). Moreover, I aim to find the most convenient and practical methods in software technology to compile and deploy these libraries and to integrate them into other environments.

As an example, I provide a Qt calculator based on the exact-real arithmetic of the Acorn library. Through this project, I demonstrate how different package management systems (Cabal and CMake) can be used to compile and deploy a package for usage in Haskell or C++; which options should be used for integration without depending on GHC; how platform-specific code can be used and how some basic tests can be written when proofs are not yet ready. I have also made some improvements to the compiler itself, as its maturity would be essential for industrial usage.

Section 1 provides a brief introduction to the goal of the project, as well as the context in which it has been born. *Section 2* summarises the history and nature of computer-verified programming, as well as the most important properties of Agda and agda2hs. After some technical information in *section 3*, *section 4* describes the essence of this article: the strategies and techniques that can be used when building a full-stack application based on Agda. *Section 5* contains some thoughts on further plans and possible research topics, followed by a short summary in *section 6*.

If a level of trustworthiness is needed while still keeping development simple and of reasonable cost, Agda is a language worthy of consideration. This paper shows that all the tools needed are available for writing Agda software for the masses.

Acknowledgements

First, I express my admiration for the work of all those who have contributed to the relatively underground fields of research mentioned in this article: developers of proof assistants and compilers, as well as the mathematicians and computer scientists having laid down the foundations of exact real arithmetic. Many of them are going to be cited in the article.

Most importantly, I would like to thank my supervisor *Ambrus Kaposi* for guiding me throughout my work, as well as for teaching me Agda in the first place. Without him, I would probably still be lost among the many branches of computer science, without any clue on which one would fit me the most. He was ready to help me whenever I needed while letting me pursue my own ideas[1]. He is also nice to work with both as a supervisor and as the head of the Type Theory course at the university, where I teach Agda under him.

The next people I want to mention are the ones who made this possible and helped me in several technical questions, despite the fact we have never met in person. They are those behind agda2hs; particularly *Jesper Cockx*, the chief developer of the project. They were open for my ideas while helping me to bring out the best of them and to prepare them for release. I sincerely hope that the program they work on has a bright future ahead of it, maybe even brighter than they would expect.

Here, I would like to thank *Levente Lócsi* and *Tamás Kozsik*, the leaders of the *Computer Science Workshop* of *Eötvös József College*, who also have monitored my progress and helped me with advice. In fact, I first met Ambrus on his course held at the workshop; that was where I got to know Agda.

More personally, I express my gratitude to everyone at the college and at the university who has helped me in any way: with a joyful conservation, emotional support or simply by being kind to me. In many of them, I see something I admire and would like to achieve in my later life, either professionally or personally.

And I cannot finish my acknowledgements without thanking my family for all the support they have provided, both materially and in other ways. Without them, I could not have experienced all the adventures I have had in my life.

[1] He always says that innovation is born when one stubbornly works on their own visions, in defiance of others. Hopefully, he will be right.

Contents

1 Introduction

Using computers for proving logical statements is not a recent idea. Proof assistants have also been around since at most 1967. Today, there are several languages and tools made specifically to help write and verify formal mathematical proofs. Even major organisations and corporations use proof assistants, like NASA (PVS) and Intel (HOL Light). Also, there are many existing solutions for extracting trustworthy code from proof assistants; the most established one probably being that developed for the French system *Coq* [59].

The language this paper concentrates on, however, is slightly different. *Agda* is a dependently typed programming language that "can be used as a proof assistant" [45]. This formulation already emphasises the main focus of Agda: writing programs. It better supports an approach of first writing a program and later adding explanation and proofs (which the library described below will be based on [21]). However, its default backend generates slow, obfuscated Haskell code, being much more ineffective than if written by hand in plain Haskell.

Then came agda2hs. *agda2hs*[2] is a tool that translates a specially annotated subset of Agda to readable Haskell [18]. While it is not able to compile arbitrary Agda code to Haskell, it can generate fast [21], simple (even human-readable) Haskell code from lightly annotated Agda programs written following certain rules. agda2hs means a new opportunity to find real-life significance for Agda.

But there are several real-life usages of Coq in industrial environments; even ones that are actively developed and well-established. Why do we need another language for exactly the same purpose?

Well, the purpose is not exactly the same. As I have noted, Coq is much older, stricter and more trusted; therefore, Agda will probably not replace it in the most safety-critical applications (e.g. software run in hospitals or on spacecraft). But I can well imagine a situation in which someone wants to verify a small critical part of the program while still writing the rest of the code relatively quickly. For long, Agda had not been fast enough for execution; with the advent of agda2hs, this changed.

In an agda2hs-based approach, an Agda backend can contain as many proofs as one may need, while still being:

- fast enough (almost as fast as Haskell written by hand);
- easy to learn (with a clean, Haskell-like syntax save for some annotations);
- able to interact with a C or C++ frontend;

[2]The documentation [16] consistently spells the name with lowercase letters. I follow this convention throughout the article.

- built using traditional package management systems of the Haskell and C++ world;
- based on a healthy balance between trustworthiness, speed and simplicity.

Such a competitor to already-established systems would also foster development of all of them, therefore bringing momentum into this relatively underground area of computing.

This paper is going to demonstrate that besides the compiler, all the tools and knowledge are available to create production-grade Agda-based programs; including the architecture, package managers, the GUI and test frameworks.

2 Background

In this section, I aim to provide some context via a short overview of what computer-verified coding means, along with an introduction to Agda and agda2hs.

2.1 Computer-verified programming

One of the first instance of proving a mathematical theorem with the help of a computer occurred in 1954, when Martin Davis programmed an already-published algorithm for deciding the truth value of every sentence in the first-order theory of addition in the arithmetic of integers – in fact, proving that the theory is decidable [27]. This is also a so-called *constructive proof*; that is, the existence of an object (in this case, the truth value) is proven by actually calculating it and showing it satisfies the conditions.

A famous example for proving a previously unproven conjecture was the four colour theorem in 1976. Here, Kenneth Appel and Wolfgang Haken reduced the number of cases to be checked to 1936 and then checked these with a program written for this purpose [67].

Both of the previous proofs involved the help of a computer; however, they were not computer-verified, which means that unless someone proves these algorithms to be correct manually, no one guarantees them to be free of errors. *Computer-verified proofs* are those checked by a special software tool called a *proof assistant* to be mathematically correct. This does *not* mean the assistant writes the proof (programs that do so are called automated theorem provers); rather, it parses a proof written by a human in a given language and certifies its validity. Having said that, modern proof assistants can automate some steps [28].

Proof assistants were already under development as early as 1967, when de Bruijn initiated the Automath project [28]. Some proofs of great symbolic importance have been verified with them, such as the aforementioned four colour theorem in Coq [32].

Many proof assistants have options for code generation; i.e. turning mathematical proofs into algorithms that can be run and used, with the guarantee that they are verified to be correct. For this, however, it is useful (and sometimes essential) to only use *constructive logic*. Simply put, constructive logic means that all existence proofs must rely on the already-mentioned technique of first producing the object itself and then proving that it has the expected properties [4]. This also means that one has to reject the so-called *law of the excluded middle* (which states that all statements are either true or false). Without omitting that, one could prove existence by contradiction, which gives no clue on what the object actually is, and therefore no algorithm for calculating it [21]. This clearly brings some limitations with itself. For instance, since no constructive proof of Bolzano's theorem is known[3], neither is a general algorithm calculating a root of a function satisfying the conditions. A weaker version

[3] "If a continuous function defined on an interval is sometimes positive and sometimes negative, it must be 0 at some point." [65]

of the theorem, however, which only states the existence of a function value *arbitrarily close* to zero, can be proven constructively [2]. For an engineer or a computer scientist, this might be enough.

Even with this limitation, there are existing solutions for extracting trustworthy code from proof assistants. Perhaps the most significant program made using such tools is the CompCert optimising compiler for the C language which is written and verified in Coq [58]. There have been kernel interpreters for Linux written in Coq [64], and there are examples for verified arithmetic libraries which can also be run, such as the CoRN project (also written in Coq) [52].

2.2 Agda

Development of the current instance of Agda, originally known as "Agda 2", began in 2005 [33]. It was a successor to the original Agda project presented in 1999 [19, 20], with a focus on creating "a practical programming language" [44].

The syntax of Agda is highly similar to that of Haskell (on which the language itself is based). Like Coq, Agda supports *dependent types*. These are "families of types indexed by objects in another type" [45]. For instance, the vector type constructor `Vec` has the type signature `Nat -> Set -> Set`. Its usage is simple: `Vec 1 Nat` is the type of vectors containing 1 natural number; `Vec 2 Nat` is of those containing 2 natural numbers etc. This way, one can play arithmetic on the length of the vectors and can require conditions that could not be required otherwise (e.g. that a vector parameter must have at least length 3).

But dependent types also enable Agda to behave as a proof assistant: logical statements themselves can be represented as types; proofs are elements of these types (this is called the Curry–Howard correspondence [45]). For example, the statement that one natural number is smaller than another one can be formalised as `_<_ : Nat -> Nat -> Set`, where n is smaller than m exactly if the type `n < m` has an element. From here, conjunctions become tuples, disjunctions become sum types, implications become functions and so on. And those functions can even be run, if they are written for this purpose.

An important difference between Agda and Coq is that Agda primarily focuses on writing programs. It is much more comfortable for programming; however, it has much fewer features and automation for proving (meaning that it is significantly harder to write complex proofs in it), and is generally more experimental [55].

To summarise, Agda could be a nice solution for programs that need to be written quickly but also verified for some critical parts. Unfortunately, the default backend compiling Agda to GHC Haskell is not capable of producing binaries with performance at least close to that of

normal Haskell code. There is another language called Idris [3] which emphasises performance while keeping dependent types; however, it is quite uncomfortable for writing proofs at its current state (for example, it does not have an Emacs mode yet). [38]

2.2.1 agda2hs

The cited article and the documentation does not call agda2hs a *compiler*; still, since it provides the essential step for writing executables and can be used almost like the built-in backends, I am going to refer to it as such.

agda2hs utilises a feature already present in Agda, called *run-time irrelevance* [46]: parameters and definitions annotated with an `@0` marker can be used by the type checker, but disappear from the final program compiled by the backend (e.g. because they will not have an effect on the end result). Of course, this also means that they cannot be used at places where a non-erased parameter would be required. This was originally meant to increase performance and to ensure that these objects really cannot influence computation.

For agda2hs, however, it is much more significant; as this is the tool with which dependent types are "eradicated" from the program. After all dependently-typed parameters and definitions are erased, the final code can safely be translated to Haskell. The proofs can still remain, they simply will not get into the program.

This also means that agda2hs is *not* a general-purpose Agda compiler. Its goal is rather to create a "common sublanguage" of Agda and Haskell which can be translated to readable Haskell while containing various proofs on the Agda side [14].

3 Technical information

In this section, I am going to briefly describe the environment I have worked in, as well as the code I am going to use as an example.

3.1 Environment

The machine used for writing and building the code was an Asus UM431-DA notebook with the following specifications:

- CPU: AMD Ryzen 7 3700U
- System memory: 16 GiB
- GPU: Radeon RX Vega 10
- Storage: SK Hynix 256 GB NVMe-M.2 SSD

The notebook has a dual-boot configuration, with Ubuntu 20.04 and Windows 10 22H2 installed. The version numbers of software used were different on the two machines; mainly for reasons explained in section 4.3.

- Ubuntu:
  - GHC 9.2.8
    - with the GCC compiler already installed
  - GCC 9.4.0
  - Cabal 3.6.2.0
  - Qt 6.7.0
- Windows:
  - GHC 9.4.8
    - with the Clang 14.0.6 compiler built in
  - llvm-mingw 20220323 based on LLVM 14.0.0 (the `msvcrt` version) [53]
    - with Clang 14.0.0
  - Cabal 3.10.2.1
  - Qt 6.5.3

For agda2hs, it is important to note that the library used does **not** get successfully compiled with the vanilla version: it works with the `have-it-both-ways` branch of my fork. This is because of some pull requests which I am still talking with the main developers about. Hopefully, they will soon get merged. The version needed can be installed by cloning the repository, checking out the commit and using `cabal install`. I also provide binary versions [22, 23] with some instructions in section 4.7.

For testing purposes, I used my machine described above, as well as the Debian-based server *Caesar* [5] of the university (through X11 forwarding in SSH), as well as Windows Sandbox and the Windows notebook of my mother, who kindly supported my work as a tester.

3.2 Example code

The project which I have worked on and I am going to use as an example is the Acorn exact-real library [25] implementing real arithmetic designed by Krebbers and Spitters [40].

My goal was to create a Qt-based, multi-platform graphical calculator application, AcornCalc [24], using the real numbers of the library and capable of printing real results with an arbitrary precision chosen by the user. This sounds simple; however, it raised so many questions and problems that it took incomparably more time to write than a simple calculator. Moreover, I had to improve the library itself in many ways. These efforts form the subject of the present paper.

Acorn itself also includes a binary called AcornShell. This provides much of the functionality of the GUI in a console environment; that way, the arithmetic and syntax could be tested without having to build the entire application.

Note, however, that the Acorn library is not part of the topic: I will only cover its exact-real arithmetic as far as it is needed so that the reader can understand the techniques explained. For those interested in details on Acorn, I recommend reading the paper specifically dedicated to it [21].

The versions described in the article are commit `b767587` of Acorn and commit `2f79dff` of AcornCalc. These can be found on the commit lists of the respective repositories [24, 25].

3.2.1 Installation

There are portable binaries which should be ready to run: one for Windows [22] and one for Linux [23]. These have been tested on other machines too; although it is possible that some libraries are missing from the reader's system.

Due to the complexity of the architecture and the huge variety of dependencies, building the two projects from source is particularly difficult. I have already heard many complaints about the installation of the Haskell stack [1], Cabal packages[4] and Qt[5] by themselves; let alone the three combined.

The most important tools needed are:

- GHC 9.2.8 or later (via ghcup).
- The zlib compression library (only on Windows, via *pacman* in MinGW).
- agda2hs. Note, however, that Acorn is **not** compatible with the vanilla version, because of some enhancements that would be needed but have not yet been merged into the official repository. Instead, a modified agda2hs compiler has to be installed, which can be accessed from the `have-it-both-ways` branch of my fork [17].
- Qt 6.6 or later. Here, the source code should be downloaded and then compiled by hand as described in the documentation [47, 48]; however, one needs to use the `-static` and `-bundled-xcb-xinput` options, [49, 50] as well as a C++ compiler *compatible* with the GHC version installed (for what this exactly means, see section 4.3.2). The online Qt installer contains an option to install CMake and Ninja below.
- CMake (version 3.29.3 worked for me).
- A backend for CMake. Ninja (version 1.10.0) is the logical choice, as it is the recommended tool for building Qt [48].
- The Catch2 C++ testing framework [35], installed and then imported via CMake.

I have tried to simplify the process by writing README files for each of the repositories; containing the various problems I have faced. For detailed instructions on how to build Acorn and AcornCalc from source, I recommend reading them [24, 25].

[4]When I teach Agda, I simply provide a link to a pre-made installer for Windows, as students get stuck with Cabal and Stack so often that otherwise, it would take a complete lesson and a half just to help them through the installation. So, for Agda to be really easy to use, a robust installation method is desperately needed.

[5]On Linux, the GUI installer worked for me on ca. the third try, after providing a different download mirror on the command line. My acquaintances teaching Qt at the university also often complain about students being unable to perform installation by themselves. Fortunately, building from source seems to be more stable.

4 Achievements

This section forms the main part of the article: here, I am going to explain what kind of new tools I have created or found for using Agda libraries effectively in C++ programs.

4.1 Enhancing agda2hs

Enhancements of the compiler are special in the sense that they are not specific techniques used in the example code. Still, they belong here.

This entire work is based upon the agda2hs compiler described above. agda2hs is still in an experimental state; thus, its development is essential for making Agda itself mature enough for real-world usage. I have contributed to the process by contributing some features and fixing bugs.

4.1.1 New features

The first significant feature I added to the compiler (in pull request #189 [15]) was the option to use "rewrite rules". This means the user can provide some custom rules in a YAML file on rewriting certain definitions to Haskell counterparts [16]. The feature uses the mechanism originally created to translate definitions of the agda2hs standard library to elementary Haskell functions. It is meant chiefly to work with the original Agda standard library [26]; as I chose not to depend on that for Acorn, I do not actually use rewrite rules in the example code, but it might be useful for other projects.

Enabling the Agda Emacs mode (PR #198 [15]) for agda2hs, however, is something that really quickens my daily work. Since agda2hs actually contains a full Agda binary, it can interact with Emacs provided that the backend is disabled by default. Most of the code added later turned out to be unnecessary and was removed in PR #238 [15]; the important part that has remained (from PR #209 [15]) is the end of `Main.hs` which disables the backend if run in interactive mode.

Another useful feature (added in PR #300 [15]) was the ability to check whether an Agda module has to be recompiled and skip it if not. As this checks the modification date of the `.agdai` file made from an already type-checked module, it also re-compiles if a dependency of an otherwise untouched module has changed. Despite this, there has been some debate about whether this is safe and the feature has not got merged yet, but it can be found on the `have-it-both-ways` branch of my repository.

A relatively new addition (which, for the time being, has not got merged either) is a mechanism for adding Haskell data structures to the standard library. The currently usable example is `Data.Map` from PR #291 [15]. It is not implemented in Agda but rather postulated with its methods and laws and then rewritten to the Haskell counterpart (similarly to other

definitions in the standard library of agda2hs). (The rewriting mechanism is to be added with PR #297 [15].) This is why the vanilla version does not work with Acorn; along with the implementation of `Control.Alternative` needed for the parser (which can be found in PR #298 [15]).

4.1.2 Improving handling of erasure

Erasure is a key concept for agda2hs, as it enables to use language elements not available in Haskell while still compiling to valid Haskell code. As I have explained in the introduction, it marks definitions and parameters that the type checker can use but do not get compiled into Haskell and do not exist in the running program [16, 46].

Apart from fixing a bug because of which local erased definitions would not get filtered out, the most important achievement I made (in PR #195 [15]) was to enable the compiler to skip erased arguments of functions returning a type when they are parameters of a data type constructor. Probably, this is easier to understand by looking at specific examples at the cited pull request. The Σ' type is actually an important data structure in the library.

4.1.3 Other contributions

Besides these, I have made some minor improvements and fixed bugs. For details about them, see the list of pull requests I have opened.

4.2 Architectural decisions

This subsection is about techniques and decisions to organise Agda-based libraries and programs into a layered architecture and create simple mechanisms for communication between the layers.

4.2.1 Layers

Acorn is written in three languages. The source files can be split into four categories:

- The arithmetical libraries themselves are mostly written in an agda2hs-compatible version of Agda, with some foreign pragmas[6] to use features agda2hs does not support yet.
- In the upper-level modules, I frequently wrote files which, although they are technically `.agda` files, consist almost exclusively of a large foreign pragma containing "native" Haskell code (I will call these **full-foreign files**). The point of this approach was to

[6] A *foreign pragma* is a code block denoted by {`-# FOREIGN AGDA2HS ...#-`} where plain Haskell code can be inserted. This gets into the compiled file without modification.

integrate them into the existing framework while also mitigating the risk of deleting them accidentally along with compiler-generated Haskell files. Moreover, if I discovered that some small parts could still be written in Agda, I could do so easily.

- There are currently three modules which are handwritten `.hs` files. `Main.hs` and `TestMain.hs`, which contain a `main` function, were required by Cabal to be non-computer-generated; therefore, I had no choice. With `Shell/Platform.hs`, the problem was the current incompatibility of agda2hs with the C preprocessor (although this could probably be fixed by a patch in the near future).

- One source file (`acornInterruptEvaluation.c`), as well as the exported header files in the `include` directory, are written in C. These are to be used in C or C++ programs based on the library.

For AcornCalc, I used Qt and an MVVM (model-view-viewmodel) architecture. The code is almost entirely in C++, except for Qt Designer's `.ui` files which, in the background, also get converted to C++ code.

The layered architecture is also illustrated visually in Figure 1.

4.2.2 Acorn's API

C and C++ programs can communicate with the Agda/Haskell backend via:

- the functions of the Haskell RTS itself (see `TinyHsFFI.h`);

- the functions exported from Interaction.agda, related to the management of the CalcState object and execution of commands;

- the `acornInterruptEvaluation` function written into a separate C file, which stops a long calculation on demand by triggering a semaphore or event.

The headers are to be found in the `include` directory. The main header is `Acorn.h`, exporting the functions of the library itself. This is almost entirely based on the `_stub.h` file generated for Interaction.agda; I just added the `acornInterruptEvaluation` function, renamed the parameters and added comments.

However, this uses type aliases defined by the Haskell FFI[7]; e.g. `HsStablePtr` for a StablePtr or `HsInt32` for a CInt. In order to interpret these, GHC automatically includes `HsFFI.h` [36] when generating the stub; however, to be able to export the library for usage with C (without having to use GHC for linking), one needs to create a standalone header file.

[7] Short for *foreign function interface*, the official mechanism of GHC to call Haskell code from C/C++ and vice versa [60].

Therefore, I've written a header called `TinyHsFFI.h`, which only contains the type definitions and FFI control functions needed for Acorn, but also has some other code copied into it from other headers of the FFI which it needed.

Normally, as the Acorn header already includes `TinyHsFFI.h`, one only needs to include `Acorn.h`. Afterwards, one can use, for example, its `hs_init` and `hs_exit` functions, which have to be called in order to initialise and deinitialise the runtime [60].

4.2.3 C++ wrapper class

For effective and easy usage (maybe even for those unfamiliar with functional programming), it is useful to hide the Agda/Haskell side as much as possible on the C++ side – this means that Haskell resources should be dealt with in an RAII[8] way so that one only needs to think about them once and forget them afterwards.

In AcornCalc, class `HsCalcStateWrapper` serves this purpose. Using the functions exported from Acorn's `Shell/Interaction.agda`, it initialises a CalcState object and obtains a StablePtr in the constructor, which it frees in the destructor. In an MVVM architecture, this effectively serves as the model; except that it is only an interface to the Haskell side.

An important problem to keep in mind is the question of C strings returned by Acorn's functions, as they are allocated on the Haskell side but are to be freed on the C++ side (as a `free` call from there is much cheaper than a call into the Haskell runtime). I solved this by copying the result into an `std::string`, freeing the pointer and then returning the string (hoping for a return-value optimisation).

Besides this, there is only one point where the Haskell runtime has to be kept in mind: the C++ `main` function. Here, one has to initialise the runtime with `hs_init(&argc, &argv)` and deinitialise it with `hs_exit()`. For the latter, it is also important to ensure that no Haskell thread exists when it is called; I solved this by putting Acorn's objects into a separate code block, shutting down the runtime only after their destructors have been called.

4.3 Qt and GHC

Here, I describe some specific problems that needed to be solved while integrating the library into a Qt program.

[8] Short for *resource acquisition is initialisation*: a programming idiom in which one represents a resource by an object on the stack whose constructor automatically connects it to a resource and whose destructor automatically releases the resource [54]. This way, the resource can be easily used in other parts of the code without even having to think about how to release it.

4.3.1 Linking the library with C code

The GHC documentation recommends linking C code using GHC itself as it takes care of the flags one should use. I did not want to follow this path, as it would have required clients to install GHC, even if they wanted to work only on the frontend. Instead, I aimed to enable them to utilise the library with the tools they already have, even if they have no knowledge of Agda or Haskell whatsoever.

This is surprisingly hard. Through CMake, I have compiled a static library (see reasons and methods later, in section 4.3.3) with the name `libAcorn.a`. I then created another CMake project for the calculator (then only consisting of a dummy main function) and tried to link the library to it:

```
target_link_libraries(calc PRIVATE Acorn::Acorn)
```

As I tried to build the program, I immediately got linker errors about unknown symbols. These mostly were in the C++ standard library and system libraries that, for some reason, did not get linked automatically.

Then, my strategy was simply to look up the missing symbols in documentations and check which library contains them. The library sets with which compilation succeeded were:

- on Ubuntu:
  - *dl* (the dynamic loader)
  - *pthread* (POSIX threads)
  - *gmp* (GNU's multiple precision arithmetic library, probably for Haskell types)
- on Windows:
  - *ntdll* (the Windows Native API)
  - *rpcrt4* (the Remote Procedure Call Runtime)
  - *dbghelp* (the debug help library)
  - *ucrt* (the Universal C Runtime)
  - *msvcrt* (the Microsoft Visual C++ Runtime)
  - *ws2_32* (Winsock 2 – only for the test executable)
  - *winmm* (the Windows Multimedia API – also only for the text executable)

This solved the problem on Ubuntu; on Windows, however, I still got linker errors. The missing symbols were the ones exported from Interaction.agda; interestingly, it found the symbols of the Haskell runtime.

I have spent a lot of time trying to come up with an explanation for this, with no actual result. Despite this, I noticed that if I linked Interaction.o separately to the executable, the process succeeded.

My temporary solution was to install the Interaction.o object file from the CMakeLists.txt of Acorn similarly to the library file (libAcorn.a) itself; like this:

```
if("Windows" STREQUAL ${CMAKE_SYSTEM_NAME})
  install(FILES src/Shell/Interaction.o DESTINATION ${CMAKE_INSTALL_LIBDIR})
endif()
```

4.3.2 Compatibility of C compilers

Here, I describe a problem which can cause seemingly inexplicable errors if one is unaware of it. Namely, the *same* C compiler has to be used for Acorn, Qt and the client program.

On Ubuntu, no efforts need to be made to achieve this, since GHC already uses the default C compiler of the system. This can be verified by checking the compiler version in the logs after running

```
ghc -v3 -optc=--version <any_Haskell_file>.hs
```

and comparing the version number to that of the default compiler.

On Windows, however, this becomes a huge issue. As the GHCup version of the Haskell stack already includes a MinGW installation [61], GHC, by default, uses the C compiler bundled with that installation. For my GHC 9.4.8, this was the Clang frontend of LLVM version 14.0.6. Afterwards, when I tried to use GCC to link the library to the Qt frontend, I got mysterious errors; it took a long time to find out what the problem was. Then I attempted to use the Clang binary included in the installation folder of GHC to build Qt, but this failed because of missing libraries.

For me, the solution was to install an LLVM-based MinGW [53] with the version number as close to that of the GHC backend as possible (that meant 14.0.0 for me), recompile the whole Qt library with it and then build the client program.

Another path I can imagine would be to make GHC use a C backend different from its default; flags `-pgmc` and `-pgmcxx` might be capable of doing this [62].

4.3.3 Building statically

I definitely wanted the final binary to be statically linked in order to avoid depending on GHC- or Qt-specific files and to ease distribution. With a binary built by Cabal (i.e. AcornShell), this automatically works: by default, the program compiled runs on the Caesar server without any libraries included. For Qt, however, the solution is not as easy as one would think.

The first problem is building the library itself statically. Although Cabal packages can contain *foreign libraries* which are meant to be linked to "foreign" languages (e.g. C), the documentation explicitly states that this feature does not support static libraries yet [56]. Therefore, I could not call Cabal from CMake. Instead, I chose to directly call GHC with the `-staticlib` option and all the other necessary options as well.

To do this, I had to tinker with the language options of the build system. I named Agda as the *linker language* (i.e. *"the language whose compiler is used to link the target"* [8]) and defined what *"creating a static library"* and *"linking an executable"* means (that means setting the `CMAKE_Agda_CREATE_STATIC_LIBRARY` and `CMAKE_Agda_LINK_EXECUTABLE` variables). This involves calling agda2hs and then immediately calling GHC with the necessary options.

After this was done, I had to achieve that the library actually can be linked into the C++ executable. For this, see section 4.3.1 above.

Then, I had to ensure I did not depend on Qt's dynamic libraries either: I recompiled Qt itself with the `-static` and `-bundled-xcb-xinput`[9] options when configuring [49, 50] (and using the correct compiler, as mentioned in section 4.3.2).

Still, the executable built this way was not entirely free from dependencies: on my Ubuntu laptop, it ran flawlessly; with X forwarding on the Caesar server, however, I got error messages about missing libraries. After copying each of them into the folder and adding a `calc.sh` script described for dynamic linking [49], I could get it work on the server, too. For the entire list, see the binary distribution [23].

On Windows [22], my strategy was to copy the executable into Windows Sandbox and then also copy every library on which I got error messages. Fortunately, there were only two: `libc++.dll` and `libunwind.dll`.

It is also important to include all the assets (e.g. images) with the executable. For now, this only means `assets/busy_indicator.gif`, a progress circle animation displayed while calculating a high-precision output.

I packed the necessary files into ZIP archives; one for each platform. On Windows, after unpacking, the program should work simply by launching the executable. On Linux, a shell script described in the Qt documentation [49] was needed which added the directory of the application to the `LD_LIBRARY_PATH` variable. Therefore, the program can be started by running `calc.sh`.

[9] After this, Qt will use a built-in version of the XCB-XInput library, instead of depending on that of the system [51], for better portability. This library is related to how the X Window System handles input devices [63].

4.4 Platform-specific code

In order to be able to interrupt calculation on demand, the UI thread needs to communicate with the Haskell thread performing the calculation. Simply sending a signal was not deemed suitable, as that would need some kind of identifier to the thread as a parameter of the interrupting function, which might be difficult to obtain. This is why I chose the concept of some kind of semaphore having a pre-defined name, using which makes it easily accessible. This, however, is implemented differently in POSIX (as *named semaphores*) [41] and the Win32 API (as *events*) [66]: although they are very similar in principle, both the identifiers and the underlying data types differ.

To remain compatible with both systems, I had to write platform-specific code on the Agda/Haskell side. I aimed to keep that on this side by including a C function in the `acornInterruptEvaluation.c` file that provides a universal interface to the interruption mechanism. Therefore, the solution had to involve the effective cooperation of C and Haskell code. I also wished to use as few Haskell calls from C as possible; since these have a large overhead [60].

My solution looks like this:

- In `Shell/Interaction.agda`, the calculation
  `evaluate $ force $ showValue value precision`
  is passed to a wrapper function called `runInterruptibly`. Here, `evaluate . force` is needed so that the thunk[10] hiding the result gets normalised to the simplest possible form immediately [29]. For this, however, an `NFData` instance is needed for `Value (C aq)` [30] (see later).

- `runInterruptibly` is exported by the umbrella module `Shell/Platform.hs`. This uses the C preprocessor to choose between the two platform-specific submodules and then simply reexports its symbols. It had to be written as a pre-made `.hs` file because agda2hs does not support the C preprocessor yet, not even in a foreign pragma.

- The implementations are written into full-foreign `.agda` files. These export the function `runInterruptibly :: IO a -> a -> IO a`.
  The first parameter is the calculation to be performed, the second one is the default value to be returned on interruption. Described with the terminology of POSIX, the function consists of the following steps:

  – Create an MVar of type `MVar (Maybe a)`.

[10] In Haskell, a *thunk* is an object in memory representing an unevaluated expression. Normally, its value is evaluated only when it is required for a further computation [42]. With the `force` function, one can control in which call (and on which thread) this evaluation happens.

  - Fork a new thread which is to perform the calculation and write the result of it into the MVar, wrapped in a `Just`.
  - Create the semaphore (in the calling thread).
  - Fork another thread (the *watcher thread*). This opens the semaphore just created and sleeps until it gets unlocked. After waking up, it checks whether there is a result; if not, it triggers a Haskell exception in the calculation thread and writes a `Nothing` into the MVar.
  - In the calling thread, install a handler for the SIGINT signal which unlocks the semaphore. This ensures that sending SIGINT (for example, by pressing Control-C) also has the effect of interrupting the calculation without exiting the program.
  - Wait until the MVar gets filled. This might mean two things:
    * The calculation has finished successfully (the result is a `Just`). The main thread then reinstalls the original handler for SIGINT, wakes the watcher thread up by unlocking the semaphore, and returns the result.
    * The calculation has been interrupted (the result is a `Nothing`). The thread also restores the SIGINT handler but then simply returns the default value provided in the parameter.
  - The semaphore is removed from the system by the watcher thread on its termination; since either way, it is the last thread to use it.

- The Win32 implementation is very similar; except that it uses events instead of named semaphores (and the corresponding functions), and the SIGINT handler is replaced by a handler for `CTRL_C_EVENT`. (The latter, however, is unfinished at the time of writing.)

The C function `acornInterruptEvaluation`, exported in the C header of the library, simply opens a handler to the semaphore/event, triggers it and closes it. The C preprocessor chooses the appropriate implementation, and since the type signatures are identical, the function can be used on the C++ side without having to differentiate between the platforms.

An example for using this mechanism for interruption is in the `HsCalcStateWrapper` class of the calculator. Here, when running an asynchronous reevaluation (in the method template `reevalCommandAsync`), the time-consuming part is placed onto an `std::thread` (which is a member of the class) which also executes a caller-provided function on the result string. The `interruptEvaluation` method stops the evaluation by calling `acornInterruptEvaluation` from the library, which then triggers the appropriate signal/event. Expecting that the thread terminates shortly after this, the method joins it (i.e. blocks until its termination). On destruction, if the thread is still running, it is stopped similarly.

For a visual illustration, see Figure 2.

4.5 Haskell instances

Most of the time, I used my self-defined type classes; but sometimes I needed to work together with the built-in class hierarchy of Haskell. For example, for using the usual output functions in the CLI, it is practical to define `Show` instances for many of Acorn's data types; while for compiling Krebbers' benchmark [39], the `Floating` class is needed.

Thankfully, many of the standard classes are available in the agda2hs standard library; the instances for them can be written in pure Agda. Once, I also expanded the library with the `Alternative` class (see PR #298 in the repository). However, this is not always that simple: `Floating`, for example, could be written in Agda but for effective proofs, this would already require exact-real arithmetic, which I did not want to hardwire into the library without greater consensus. In these cases, I simply wrote the instance in a FOREIGN pragma while importing the needed instances (both in Agda and the FOREIGN pragma so that agda2hs knew the dependencies).

A special example is NFData [30], needed for the full evaluation of thunks on the calculation thread (see section 4.4 above). In this case, I followed a similar full-foreign solution while also enabling the Haskell language extensions needed.

4.6 Configuring package management systems

One of my priorities was to ensure that developers wishing to use the library be able to do so, even if having no knowledge of Agda. To achieve this, I prepared Acorn as a package in two build systems, both of which also provide package management features: Cabal for Haskell developers and CMake for C/C++ developers.

4.6.1 Cabal

Cabal is the standard package system for Haskell; enabling users to install, create, build and distribute software. Cabal organises projects into packages; these can depend on other packages that the system automatically installs when needed. Packages contain, besides the code itself, a declarative configuration file (with extension `.cabal`) and (optionally) a `Setup.hs` file which describes how the package should be built [57].

As agda2hs compiles the code to Haskell, it is no surprise that I wanted to let others use it in Haskell projects. Cabal, however, only understands Haskell files and can only handle Haskell compilers (as this is what it was designed for). To tell Cabal to run agda2hs automatically, some special settings are needed in the configuration file (called `acorn.cabal`).

- All Agda source files have to be added to the global `extra-source-files` field. These then get copied to distribution packages [56] and can be accessed by Setup.hs.

- The Haskell modules to be generated from them must be added to the `autogen-modules` field of the library, as well as either to `exposed-modules` or `other-modules` [56].
- In a custom Setup.hs file, agda2hs has to be called before starting the build. To achieve this, I added a shell call (using the `callCommand` function) to the `preBuild` user hook [37].
- For the executable, a `Main.hs` file (which is *not* generated by agda2hs and therefore has to be added to version control) has to be created. This is because the `main-is` field (where the file belongs) does not yet support autogenerated modules [56]. The modules of the library need not be listed again; it is enough to add the library itself to `build-depends`.
- The same goes for the test suite, if any. Here, I named the file containing the main function `TestMain.hs`.

For now, **agda2hs has to be installed separately** (and this means my custom version described in 3.1), with the folder of the binary added to PATH. I plan to include the specific commit in `cabal.project` and call its methods from inside Haskell instead of through `callCommand` (which depends on the system shell); this way, there would be no need to install agda2hs on its own.

4.6.2 CMake

CMake is actually not a build system by itself, but a program that generates scripts for local build systems, enabling developers to write a single, cross-platform configuration file [34]. However, it is also a package management system, as it provides tools to import packages and create and install new ones [9]. This feature is more interesting now, as creating a CMake package is a nice way to enable integration with C and C++. But this is more challenging, as CMake has not been designed for use with Haskell, let alone Agda.

I have already mentioned in section 4.3.3 how I had managed to build a static library: I set "Agda" as the linker language and specified in variables what it means to build the project with Agda. These variables are:

- `CMAKE_Agda_CREATE_STATIC_LIBRARY` and
- `CMAKE_Agda_LINK_EXECUTABLE`.

Here, I called agda2hs (this is why it **needs to be installed separately** here, too) and then GHC with the necessary options, on a file called `All.agda` (or `All.hs` after translation). This file simply contains imports for all the files needed for the library, both in Agda and a foreign Haskell pragma. (When compiling the AcornShell executable, I passed `Main.hs` instead to GHC.)

I then installed the files needed for inclusion in other projects to the locations defined in GNUInstallDirs [13]. This means `/usr/local/...` on Unix [31] and `C:\Program Files (x86)\Acorn\...` or `C:\Program Files\Acorn\...` on Windows [10]. The files installed are:

- `AcornShell` – the executable.
- `libAcorn.a` – the static library.
- `src/Shell/Interaction.o` – on Windows, this object file somehow gets corrupted when added to `libAcorn.a`; therefore, it needs to be installed separately along with the library.
- `include/Acorn.h`, `include/TinyHsFFI.h` – the include headers.
- `AcornTargets.cmake`, `Config.cmake.in`, `AcornConfig.cmake` – this is to ensure Acorn can be imported by simply adding `find_package(Acorn 0.1 REQUIRED)` to the CMakeLists.txt of the client [11]. `Config.cmake.in` is written by hand and the other two are auto-generated. But for the latter, another command is needed at the end of the CMakeLists.txt file of Acorn, which looks like this [11]:

```
write_basic_package_version_file(
  ${CMAKE_CURRENT_BINARY_DIR}/AcornConfigVersion.cmake
  VERSION 0.1
  COMPATIBILITY SameMinorVersion
  )
```

Unfortunately, CMake cannot yet follow whether Agda files have been changed; so for a rebuild, one has to delete libAcorn.a and AcornShell and run `make all` again. After this, however, the generated Haskell and object files are rewritten only if needed, as agda2hs and GHC are able to track this.

In AcornCalc, apart from the above-mentioned `find_package(Acorn 0.1 REQUIRED)`, an extra step might be needed. On Windows, the separately bundled Interaction.o has to be linked like this:

```
if("Windows" STREQUAL ${CMAKE_SYSTEM_NAME})
  target_link_libraries(calc PRIVATE
      "${CMAKE_INSTALL_PREFIX}/../Acorn/lib/Interaction.o")
endif()
```

For the calculator, my initial strategy was to add a CMakeLists.txt to every subfolder and link the files there with `add_subdirectory` [12]. Later, as Qt Creator took care of the dependencies by itself (it added new files to `qt_add_executable` manually), I abandoned this approach.

On Ubuntu, I used Make as the backend for CMake; on Windows, I chose Ninja. For AcornCalc, I switched to Ninja on Ubuntu, too (as it was recommended for Qt).

4.7 Deployment

After the steps described in section 4.3.3, I packed the executable and the needed libraries (and on Linux, the shell script) into two zip files for the two platforms. Apart from decompressing, the calculator can be used without installation.

4.8 Tests

Due to Acorn's *"running-first approach"* [21], most of the code is not yet verified; this means that on the short run, tests might be useful for guaranteeing reliability until full proofs are written.

I have tests in three layers of the code; most of them are incomplete, but they already serve as a demonstration on how such code can be tested.

- Native Agda "tests" are probably the most exciting, as they demonstrate how the type checker can be used for testing even before compilation begins. An example is provided in `src/Test/Parser.agda`.
  - Here, my first idea was to write definitions with the type signature of a general proof, but prove only the test cases wanted and use the `cheat` postulate[11] for all the other cases. By doing so, some directions are also provided on how to proceed with proving the correctness of the code. This pattern works when checking the parser directly (although strings have to be pattern-matched via their canonical list form, e.g. `'T' :: 'r' :: 'u' :: 'e' :: []`).

[11] The `cheat` postulate is defined in module `Tool.Cheat`. It can have an arbitrary type, so it can be used instead of a real proof wherever one wants to skip verifying a step. However, it is erased; meaning that it cannot substitute a proof where values contained in it are to be used run-time.

  - However, Agda does not support pattern matching on natural literals larger than 20[12], which is understandable for "real" proofs but uncomfortable for this use case. Therefore, one has to create definitions for larger test cases separately.
  - Finally, I have also included some concrete test cases for the full parser, providing what syntax tree should be built when running the parser on a given string.

- On the Haskell side, I used the *QuickCheck* [6] automatic testing library. It has proved to be really useful; as I have found several bugs in the parser while writing these tests (even ones related to edge cases) and also rethought the types of literals in syntax trees to better represent those given by the user. An example is `src/Test/Haskell/Parser.agda`.
  - I first provided default generators for some types using the `Arbitrary` class [43]. The most important of them is `Exp`, the type of syntax trees. Random syntax trees will be frequently needed here; I had to ensure, however, that these remain of a manageable size. I solved this by using the `frequency` function [6], giving a constant (but large) chance for leaf expressions and a specifiable chance for inner nodes. The latter can be customised using the `maxSize` option of `Args` [7].
  - I then implemented a function called `expToString`, which converts syntax trees back to strings with as few brackets as possible. This enables me to write a general test case: if I converted an arbitrary tree to a string and then parse it again, I should get back the original tree.
  - The propositions to be used can actually be written in Agda. For now, I have written one for a unary operator, one for a binary operator and the above-mentioned general test.
  - I have collected the tests into an `IO Bool` function which runs all of them (for the ones generating the tree, it gives a `maxSize` of 10 instead of the default 100 [7]). It returns whether all the tests were successful.
  - Finally, in the handwritten `TestMain.hs` file's `main`, I ran this function (and, in the future, a similar function from all test modules). I then called `exitSuccess` on success and `exitFailure` otherwise; without this, it would always return with `PASS` regardless of the result (and only the logs would tell whether the tests have really passed).
  - I have added a `test-suite` to `acorn.cabal` with `TestMain.hs` in the `main-is` field. Therefore, the tests can be run simply with `cabal test`.

[12] The error message is: *"Matching on natural number literals is done by expanding the literal to the corresponding constructor pattern, so you probably don't want to do it this way."*

- I also have a test module on the C++ side.
  - For this, I used the well-known Catch2 [35] test library, creating several tests for the `HsCalcStateWrapper` and `MainViewModel` C++ classes. These are usually related to issuing commands with given errors, given types of result or a given precision. As these are "ordinary" Catch tests, they are easy to write even by those not familiar with functional programming whatsoever.
  - The test has its own `main.cpp` file and is added to the CMakeLists.txt of AcornCalc as another executable.

5 Future plans and directions

Although the project already demonstrates what Agda is capable of using the right tools and architecture, there are still many things to do. Even besides that, there are almost endless possibilities and directions to make use of computer-verified programming in real-life software.

5.1 Short-term plans

A rather straightforward option to extend this research is to create a basic CI/CD solution for Acorn. Since I already have test cases and frameworks, this would not be hard to do, and would demonstrate how problems could be found during the type checking phase already.

A more complicated task is to eradicate the "import hell" present in several source files: since agda2hs writes Haskell imports based on the definitions used and not on the Agda imports themselves, it cannot detect dependencies of foreign pragmas. They have to be imported in a foreign pragma itself; but then they must also be imported on the Agda side because otherwise, the type checker might not recognise that the module depends on another module and might omit it from type checking, which is especially problematic if that module is dysfunctional or has been updated. A simple solution could be to copy all Agda imports directly into the Haskell code; that way, these dependencies only have to be included once. This option could also be hidden behind a flag so as not to break existing behaviour.

5.2 Long-term possibilities

Here, the first one is also straightforward: to finish the proofs for Acorn. This would provide a living, practical demonstration for a computer-verified library ready for real-life usage; however, this seems to require much effort and endurance. Fortunately, I have already got assistance for this work.

Afterwards, I can shift my focus on searching for the areas where such provably correct but quickly written libraries are needed the most. Coq serves the most safety-critical applications, but I believe there might be cases where a reasonable balance is needed between trustworthiness and ease of coding (and therefore production costs).

Finally, perhaps the most exciting perspective I can see is to create a proof automation tool based on language models or other ML-related technology that outperforms current solutions and is capable of writing proofs that would take considerable time for humans. This might be the one feature that would really start Agda on its way to industrial usage.

6 Summary

Despite the incompleteness of the implementation and the unsolved problems, I think I have achieved my goal: to demonstrate that Agda is already capable of being used in real-life programs. Everything needed is now there:

- a fast compiler with enough features to handle a particularly complex library;
- a layered architecture pattern enabling Agda code to interact with Haskell and even C-style imperative languages,
- techniques to hide the details of the backend and let even those unfamiliar with functional programming write an efficient frontend;
- a way to integrate Agda libraries under a Qt GUI;
- a method for easy distribution;
- ability to write platform-specific code pieces into the library;
- integration with widely used package management systems which enables programmers unfamiliar with Agda to utilise the library;
- effective testing methods.

Of course, there is much to be improved, and writing proofs in Agda is still a difficult task. But with the advancement of automated provers, we might soon see the language get more used and loved – or even use Agda-based software ourselves.

References

[1] Francesco Ariis et al. *When did it become so hard to install Haskell on Windows?* 2020. URL: https://mail.haskell.org/pipermail/haskell-cafe/2020-May/132264.html (visited on 05/01/2024).

[2] Errett Bishop. *Foundations of Constructive Analysis*. McGraw Hill, 1967. ISBN: 978-4871877145.

[3] Edwin Brady. *Idris: A Language for Type-Driven Development*. URL: https://www.idris-lang.org/ (visited on 05/02/2024).

[4] Douglas Bridges, Erik Palmgren, and Hajime Ishihara. *Constructive Mathematics*. URL: https://plato.stanford.edu/entries/mathematics-constructive (visited on 05/10/2024).

[5] *Caesar interaktív kiszolgálás*. Hungarian. IT Directorate, Eötvös Loránd University. URL: https://iig.elte.hu/content/caesar-interaktiv-kiszolgalas.t.20547 (visited on 05/10/2024).

[6] Koen Claessen, Björn Bringert, and Nick Smallbone. *QuickCheck. Automatic testing of Haskell programs*. Hackage. URL: https://hackage.haskell.org/package/QuickCheck (visited on 04/30/2024).

[7] Koen Claessen, Björn Bringert, and Nick Smallbone. *QuickCheck. Automatic testing of Haskell programs*. Test.QuickCheck.Test. Hackage. URL: https://hackage.haskell.org/package/QuickCheck-2.15/docs/src/Test.QuickCheck.Test.html (visited on 04/30/2024).

[8] *CMake 3.29.2. Documentation. cmake-properties(7) → LINKER_LANGUAGE*. Kitware, Inc. and Contributors. URL: https://cmake.org/cmake/help/latest/prop_tgt/LINKER_LANGUAGE.html (visited on 04/28/2024).

[9] *CMake 3.29.2. Documentation. cmake-packages(7)*. Kitware, Inc. and Contributors. URL: https://cmake.org/cmake/help/latest/manual/cmake-packages.7.html (visited on 05/02/2024).

[10] *CMake 3.29.2. Documentation. cmake-variables(7) → CMAKE_INSTALL_PREFIX*. Kitware, Inc. and Contributors. URL: https://cmake.org/cmake/help/v3.17/variable/CMAKE_INSTALL_PREFIX.html (visited on 04/29/2024).

[11] *CMake 3.29.2. Documentation. CMake Tutorial → Step 11: Adding Export Configuration*. Kitware, Inc. and Contributors. URL: https://cmake.org/cmake/help/latest/guide/tutorial/Adding%5C%20Export%5C%20Configuration.html (visited on 04/30/2024).

[12] *CMake 3.29.2. Documentation. cmake-commands(7) → add_subdirectory*. Kitware, Inc. and Contributors. URL: https://cmake.org/cmake/help/latest/command/add_subdirectory.html (visited on 05/01/2024).

[13] *CMake 3.29.3, Documentation. cmake-modules(7) → GNUInstallDirs*. Kitware, Inc. and Contributors. URL: https://cmake.org/cmake/help/latest/command/add_subdirectory.html (visited on 05/10/2024).

[14] Jesper Cockx et al. *agda2hs. Compiling Agda code to readable Haskell*. GitHub. URL: https://github.com/agda/agda2hs (visited on 05/10/2024).

[15] Jesper Cockx et al. *agda2hs. Pull requests*. GitHub. URL: https://github.com/agda/agda2hs/pulls (visited on 05/10/2024).

[16] Jesper Cockx et al. *agda2hs Documentation*. URL: https://agda.github.io/agda2hs/ (visited on 05/10/2024).

[17] Jesper Cockx, Viktor Csimma, et al. *agda2hs. Forked by viktorcsimma.* Branch have-it-both-ways. GitHub. URL: https://github.com/viktorcsimma/agda2hs/tree/have-it-both-ways (visited on 05/11/2024).

[18] Jesper Cockx et al. "Reasonable Agda Is Correct Haskell: Writing Verified Haskell using agda2hs". In: *Proceedings of the 15th ACM SIG-PLAN International Haskell Symposium*. Ljubljana, 2022. URL: https://jesper.sikanda.be/files/reasonable-agda-is-correct-haskell.pdf.

[19] Catarina Coquand and Thierry Coquand. "Structured Type Theory". In: *Proceedings of the Workshop on Logical Frameworks and Meta-Languages*. 1999. URL: https://www.eecs.uottawa.ca/~afelty/LFM99/CoquandCoquand.pdf (visited on 05/12/2024).

[20] Catarina Coquand, Dan Synek, and Makoto Takeyama. "An Emacs-Interface for Type-Directed Support for Constructing Proofs and Programs". In: *Proceedings of the European Joint Conferences on Theory and Practice of Software*. 2006. URL: http://www.cse.chalmers.se/~coquand/emacs.pdf (visited on 05/12/2024).

[21] Viktor Csimma. "Acorn – an agda2hs-compatible representation of exact real arithmetic". In: *Proceedings of the Scientific Students' Circle Conference at the Faculty of Informatics of Eötvös Loránd University*. Budapest, Nov. 2023. URL: https://csimmaviktor.web.elte.hu/acorn.pdf (visited on 04/30/2024).

[22] Viktor Csimma. *AcornCalc. Binary distribution for Windows*. URL: https://csimmaviktor.web.elte.hu/calc_windows.zip (visited on 05/11/2024).

[23] Viktor Csimma. *AcornCalc. Binary distribution for Linux*. URL: https://csimmaviktor.web.elte.hu/calc_linux.zip (visited on 05/11/2024).

[24] Viktor Csimma. *AcornCalc. A calculator based on the Acorn library, with GUI written in Qt*. GitHub. URL: https://github.com/viktorcsimma/acorn-calc (visited on 05/10/2024).

[25] Viktor Csimma and Zsófia Fekete. *Acorn. An agda2hs-compatible implementation of Krebbers–Spitters reals, focusing on usability*. GitHub. URL: https://github.com/viktorcsimma/acorn (visited on 05/10/2024).

[26] Nils Anders Danielsson et al. *The Agda standard library*. GitHub. URL: https://github.com/agda/agda-stdlib (visited on 05/10/2024).

[27] Martin Davis. "The Early History of Automated Deduction". In: *Handbook of Automated Reasoning* (2001). URL: https://web.archive.org/web/20120728092819/http://www.cs.nyu.edu/cs/faculty/davism/early.ps.

[28] Herman Geuvers. "Proof assistants: History, ideas and future". In: *Sādhanā* (2009). URL: https://www.ias.ac.in/article/fulltext/sadh/034/01/0003-0025 (visited on 05/10/2024).

[29] The University of Glasgow. *base-4.19.1.0. Basic libraries*. Hackage. URL: https://hackage.haskell.org/package/base-4.19.1.0/docs/ (visited on 04/23/2024).

[30] The University of Glasgow. *deepseq-1.5.0.0. Deep evaluation of data structures*. Hackage. URL: https://hackage.haskell.org/package/deepseq-1.5.0.0/docs/ (visited on 04/23/2024).

[31] *GNU Coding Standards. Variables for Installation Directories*. Free Software Foundation, Inc. Chap. 7.2.5. URL: https://www.gnu.org/prep/standards/html_node/Directory-Variables.html (visited on 04/29/2024).

[32] Georges Gonthier. "Formal Proof — The Four-Color Theorem". In: *Notices of the American Mathematical Society* (2008). URL: https://www.ams.org/notices/200811/tx081101382p.pdf (visited on 05/10/2024).

[33] *History*. The Agda Wiki. URL: https://wiki.portal.chalmers.se/agda/Main/History (visited on 05/12/2024).

[34] Bill Hoffman and Kenneth Martin. *The Architecture of Open Source Applications. CMake*. URL: https://aosabook.org/en/v1/cmake.html (visited on 05/02/2024).

[35] Martin Hořeňovský et al. *llvm-mingw: An LLVM/Clang/LLD based mingw-w64 toolchain. Release 20220323 with LLVM 14.0.0*. GitHub. URL: https://github.com/mstorsjo/llvm-mingw/releases/tag/20220323 (visited on 05/01/2024).

[36] *HsFFI.h: A mapping for Haskell types to C types, including the corresponding bounds*. Bundled with GHC 9.2.8. The GHC Team.

[37] Isaac Jones. *Cabal-3.6.2.0. A framework for packaging Haskell software*. Distribution.Simple. Hackage. URL: https://hackage.haskell.org/package/Cabal-3.6.2.0/docs/Distribution-Simple.html (visited on 04/29/2024).

[38] Andre Knispel. *Agda vs. Coq vs. Idris*. 2020. URL: https://whatisrt.github.io/dependent-types/2020/02/18/agda-vs-coq-vs-idris.html (visited on 05/02/2024).

[39] Robbert Krebbers and Jelle Herold. *fewdigits. forked from r6.ca/FewDigits/*. GitHub. URL: https://github.com/robbertkrebbers/fewdigits/ (visited on 05/10/2024).

[40] Robbert Krebbers and Bas Spitters. "Type classes for efficient exact real arithmetic in Coq". In: *Logical Methods in Computer Science* (Vol. 9(1:01) 2013). URL: https://arxiv.org/pdf/1106.3448 (visited on 05/10/2024).

[41] *Linux Programmer's Manual. sem_overview – overview of POSIX semaphores*. Distributed with Ubuntu 20.04.

[42] Simon Marlow. *Parallel and Concurrent Programming in Haskell*. O'Reilly, 2013, pp. 9–10. ISBN: 978-1-449-33594-6.

[43] Joe Nelson. *The Design and Use of QuickCheck*. Jan. 14, 2017. URL: https://begriffs.com/posts/2017-01-14-design-use-quickcheck.html (visited on 05/10/2024).

[44] Ulf Norell. "Towards a practical programming language based on dependent type theory". PhD thesis. Chalmers University of Technology, 2007. URL: http://www.cse.chalmers.se/~ulfn/papers/thesis.html (visited on 05/12/2024).

[45] Ulf Norell et al. *The Agda User Manual. What is Agda?* URL: https://agda.readthedocs.io/en/v2.6.4.3/getting-started/what-is-agda.html (visited on 05/02/2024).

[46] Ulf Norell et al. *The Agda User Manual. Run-time Irrelevance*. URL: https://agda.readthedocs.io/en/v2.6.2.2/language/runtime-irrelevance.html (visited on 05/10/2024).

[47] *Qt 6.7. Qt for Linux/X11. Building from Source*. The Qt Company Ltd. URL: https://doc.qt.io/qt-6/linux-building.html (visited on 05/11/2024).

[48] *Qt 6.7. Qt for Windows. Building from Source*. The Qt Company Ltd. URL: https://doc.qt.io/qt-6/windows-building.html (visited on 05/11/2024).

[49] *Qt 6.7. Qt for Linux/X11. Deployment*. The Qt Company Ltd. URL: https://doc.qt.io/qt-6/linux-deployment.html (visited on 04/28/2024).

[50] *Qt 6.7. Supported Platforms*. The Qt Company Ltd. URL: https://doc.qt.io/qt-6/supported-platforms.html (visited on 05/10/2024).

[51] *Qt 6.7. XCB-XInput*. The Qt Company Ltd. URL: https://doc.qt.io/qt-6/qtgui-attribution-xcb-xinput.html (visited on 05/13/2024).

[52] Bas Spitters et al. *CoRN – Coq Repository at Nijmegen*. GitHub. URL: https://github.com/coq-community/corn/tree/master (visited on 05/02/2024).

[53] Martin Storsjö et al. *Catch2. A modern, C++-native, test framework for unit-tests, TDD and BDD*. GitHub. URL: https://github.com/catchorg/Catch2 (visited on 04/30/2024).

[54] Bjarne Stroustrup. *Bjarne Stroustrup's C++ Style and Technique FAQ. Why doesn't C++ provide a "finally" construct?* URL: https://www.stroustrup.com/bs_faq2.html#finally (visited on 05/10/2024).

[55] Wouter Swierstra. *AgdaVsCoq*. The Agda Wiki. URL: https://wiki.portal.chalmers.se/agda/Main/AgdaVsCoq (visited on 05/02/2024).

[56] *The Cabal User Guide, 3.10. Cabal Reference*. Package Description. The Cabal Team. Chap. 6. URL: https://cabal.readthedocs.io/en/3.10/cabal-package.html (visited on 04/28/2024).

[57] *The Cabal User Guide, stable version. Cabal Explanation*. What Cabal does. The Cabal Team. Chap. 1. URL: https://cabal.readthedocs.io/en/stable/cabal-context.html (visited on 04/28/2024).

[58] *The CompCert verified compiler. Commented Coq development*. The CompCert Project. URL: https://compcert.org/doc/index.html (visited on 05/02/2024).

[59] *The Coq Proof Assistant*. The Coq Team. URL: https://coq.inria.fr/ (visited on 05/12/2024).

[60] *The Glasgow Haskell Compiler, 9.2.8. Foreign Function Interface (FFI)*. The GHC Team. Chap. 6.17. URL: https://downloads.haskell.org/~ghc/9.2.8/docs/html/users_guide/exts/ffi.html (visited on 04/23/2024).

[61] *The Glasgow Haskell Compiler, 9.2.8. Running GHC on Win32 systems*. The GHC Team. Chap. 13. URL: https://downloads.haskell.org/~ghc/9.2.8/docs/html/users_guide/win32-dlls.html (visited on 04/28/2024).

[62] *The Glasgow Haskell Compiler, 9.2.8. Flag reference*. The GHC Team. Chap. 5.6. URL: https://downloads.haskell.org/~ghc/9.2.8/docs/html/users_guide/flags.html (visited on 04/28/2024).

[63] *The X New Developer's Guide. Modern Extensions to X*. Xinput 2. 2013. URL: https://csimmaviktor.web.elte.hu/calc_linux.zip (visited on 05/13/2024).

[64] Xi Wang et al. “Jitk: A Trustworthy In-Kernel Interpreter Infrastructure”. In: *USENIX Symposium*. 2014. URL: https://people.csail.mit.edu/nickolai/papers/wang-jitk.pdf (visited on 05/01/2024).

[65] Eric W. Weisstein. *Bolzano's Theorem*. URL: https://mathworld.wolfram.com/BolzanosTheorem.html (visited on 05/10/2024).

[66] *Windows App Development. Using Event Objects (Synchronization)*. Microsoft. URL: https://learn.microsoft.com/en-us/windows/win32/sync/using-event-objects (visited on 05/10/2024).

[67] Limin Xiang. “A formal proof of the four color theorem”. In: *arXiv* (2009). URL: https://arxiv.org/pdf/0905.3713 (visited on 05/10/2024).

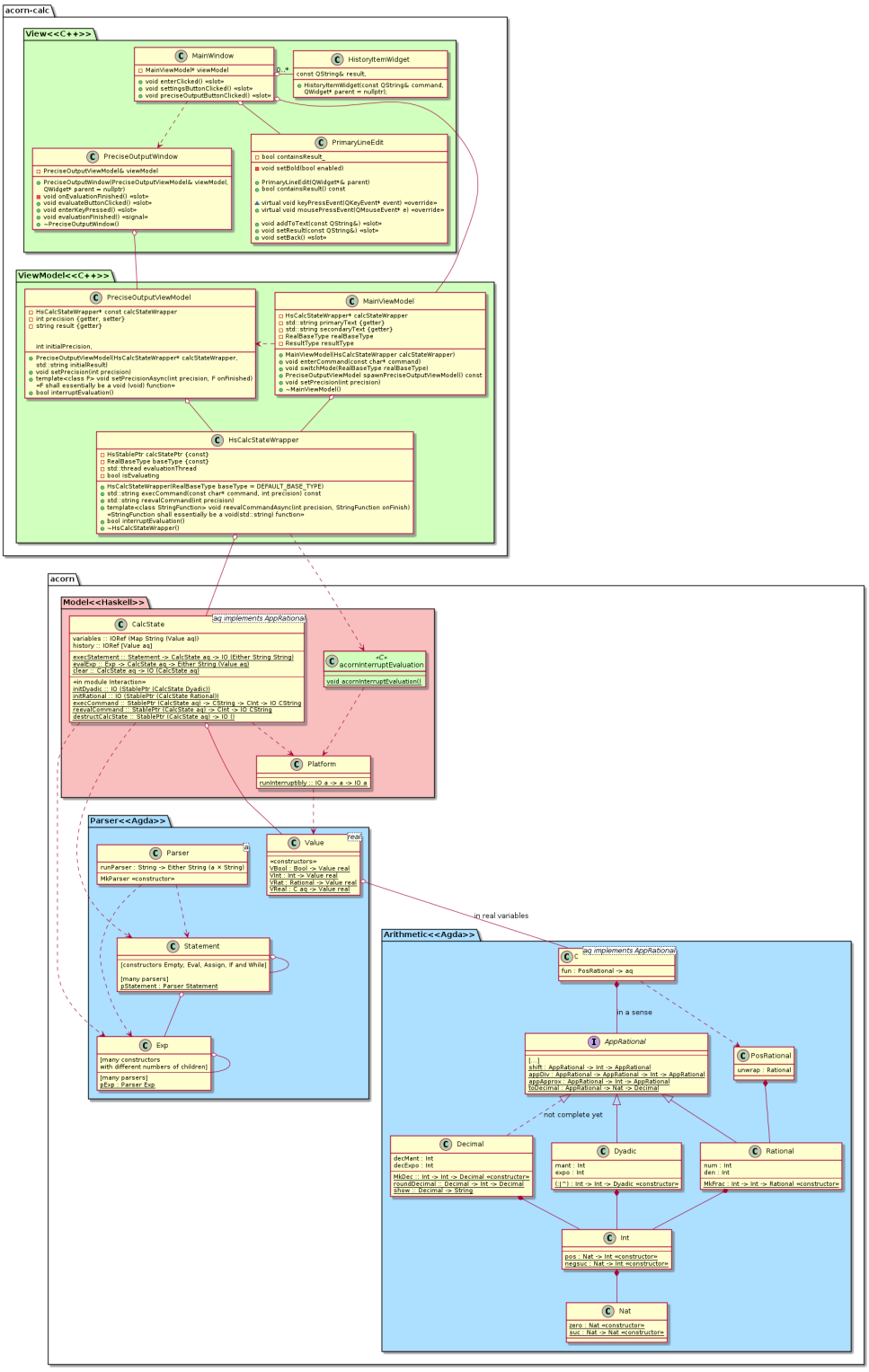

Figure 1: A basic diagram of how layers are built upon each other, with the most important classes and modules shown.

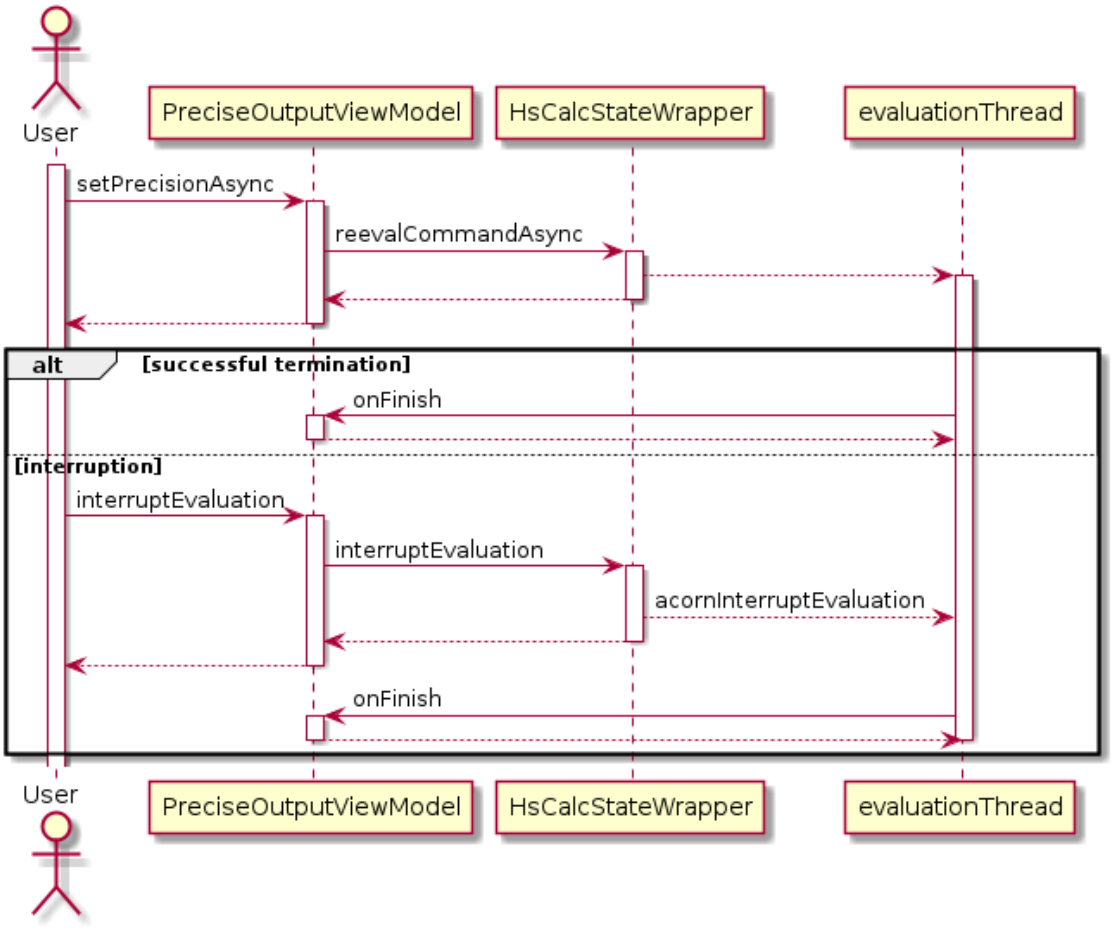

Figure 2: Asynchronous evaluation and its interruption.