

The Agda SDK:
a toolkit for writing Agda-powered GUI applications

Csimma Viktor

TDK-dolgozat

Lezárás dátuma: 2024. december 1.

The Agda SDK: a toolkit for writing Agda-powered GUI applications

For the 37th National Scientific Students' Circle Conference



Viktor Csimma

Computer Science MSc; first year

Supervised by associate professor
Ambrus Kaposi

Eötvös Loránd University

Faculty of Informatics

Department of Programming Languages and Compilers

Eötvös József College

Budapest, 2024

Abstract

Using `agda2hs` and some seemingly unorthodox techniques, writing Qt applications in C++ with Agda-based, provably correct backends is already possible. It is now time to give this power to the developer by providing a well-documented, general toolkit.

The package presented contains:

- platform-specific installation scripts that download all the necessary tools;
- a basic skeleton code with some important classes to get started;
- CMake scripts making compilation easy for everyone;
- a sample project demonstrating some capabilities and useful techniques;
- frontend classes providing access even to GUI developers unfamiliar with Agda;
- easy-to-understand documentation.

While writing the Agda code, I applied an implementation technique in which even complex Haskell concepts can be described in Agda (with the help of postulates). This way, the amount of foreign Haskell code needed can be greatly reduced.

With benchmarks, I prove that `agda2hs` is viable among other solutions, like the OCaml extraction feature of Coq and the default `MAlonzo` backend of Agda.

Among the tools included is an entirely new method of handling background Haskell calculations, via exportable future objects that are platform-independent and easy to use. This is probably the most advanced and user-friendly method of handling background computations in Haskell backends.

Hopefully, the project will help to show the so-far hidden potential of Agda as a production language, providing a new, powerful way to create provably correct backends.

Acknowledgements

I have written several times on how many people have helped me on my way, but I feel that by simply repeating it, it would lose its significance. I just would like to express to my supervisor, my teachers and the developers of the tools how much I owe them. Also, I wish to mention my family, as well as my friends at my second home. I love them and I am grateful they are part of my life.

*To my teachers, enablers, family, friends and the College –
with gratitude.*

Contents

1	Introduction	5
1.1	Agda and agda2hs	5
1.2	Coq	6
1.3	Comparison	6
1.4	Contributions and goals	7
2	Technologies and decisions	9
2.1	Futures	9
2.1.1	Motivation	9
2.1.2	Overview	10
2.2	Haskell dependencies	11
2.3	C++ class hierarchy	13
2.4	Installation	13
3	The SDK	14
3.1	Installation	14
3.2	CMake projects	14
3.3	Backend	14
3.4	Frontend	15
3.5	Testing	15
3.6	Documentation	16
3.7	Example projects	16
3.7.1	The skeleton project	16
3.7.2	EvenCounter	16
3.7.3	AcornCalc	16
4	Benchmarks	17
4.1	Hardware configuration	17
4.2	Test suite	17
4.3	Interpretation	18
4.4	Type-checking	18
4.5	Interactive evaluation	18
4.6	Compiling and running	19
4.6.1	Translation to the underlying language	20
4.6.2	Compilation to executables	21
4.6.3	Running the executables	22
4.7	Conclusion	22

1 Introduction

In the first section, I give an insight into existing code extraction tools and foreign interfaces for Agda and Coq. I also describe the motivation behind the project and its position among other solutions.

1.1 Agda and agda2hs

The original Agda project was started by Catarina Coquand in 1999[[agda1-paper](#), [agda-history](#)]; An interactive interface was soon presented for it, followed by an Emacs extension[[agda-emacs-paper](#)]. When Ulf Norell and Andreas Abel started in 2005 a complete rewrite of the language, called Agda 2¹, their goal was to create “a practical programming language based on dependent type theory”[[norell-thesis](#)].

In accordance with this goal, a compiler with the name of MAlonzo (also referred to simply as “the GHC backend”[[agda-docs-compilers](#)]) has been around since at least 2008[[malonzo-commit](#)]. This translates any legal Agda code to Haskell, with type coercions being used to simulate dependent types (since they are not available in Haskell). As this creates obfuscated, hard-to-read code, it is usually not enough for integrating Agda code into a Haskell environment:

“While many of the coercions inserted by MAlonzo are not necessary to make the code be accepted by GHC, they cannot be avoided in general because Agda supports full dependent types, while GHC (currently) does not. The coercions inserted by MAlonzo make it difficult to make the jump from having an Agda prototype of a Haskell program (or a component in a larger Haskell project) to having a production Haskell program.”[[agda2hs-paper](#)]

This was one of the reasons why agda2hs² was introduced in 2022[[agda2hs-paper](#)]. agda2hs is a code extraction tool, similar to those built into Coq[[coq-extraction](#)] and other theorem provers. While it only supports a subset of Agda where all dependently-typed portions have been marked for the compiler as irrelevant (these marks are also referred to as erasures[[agda-docs-run-time-irrelevance](#)]), it generates human-readable Haskell code that is almost as fast as if it had been written by hand, thereby enabling users to include verified Agda code in a Haskell project.

I was interested whether agda2hs is fast enough to be viable among other solutions – particularly in comparison to the code extraction capabilities of Coq.

¹Now, this is simply referred to as “Agda”, having succeeded the previous implementation.

²The documentation[[agda2hs-docs](#)] consistently spells the name with lowercase letters. I follow this convention throughout the article.

1.2 Coq

Coq is an interactive theorem prover³ started as early as 1984 by Thierry Coquand and Gérard Huet[**coq-about**]. Coq is one of the most well-known systems of its kind, with some well-known applications such as a verified proof of the four-colour theorem[**gonthier**] and the CompCert C compiler – the latter even having industrial applications, with an extended version being sold as a commercial product.

The underlying language of Coq is OCaml (in contrast with Haskell being behind Agda); they also share a similar syntax based on ML (parallel to Agda’s resemblance of Haskell). OCaml is not as widespread as Haskell, but also has several applications even at large software companies, as it focuses on performance and usability[**pl-software-award**]. In contrast to Haskell, the evaluation strategy of OCaml is *strict* by default[**cornell-ocaml**].

Coq has a built-in tool for extracting definitions to lower-level functional languages; with the default being OCaml (Haskell and Scheme are also supported). This already does some optimisation in order to be compatible with the strict evaluation strategy of the language.[**coq-extraction**]

Recently, a foreign interface between Coq and C, called VeriFFI, has been announced; this has even been verified (with VST and Coq itself). VeriFFI is going to be announced in the Proceedings of the ACM on Programming Languages, in January 2025.[**veriffi**]

1.3 Comparison

“I always try to use the best tool for the job.

There are plenty of Coq proofs I wouldn’t want to write in Agda.

But there are quite a few Agda programs, that I wouldn’t like to write in Coq.”

– Wouter Swierstra[**agda-vs-coq**]

As we are going to see, Coq is superior compared to agda2hs in terms of translation and compilation time; usually even of the speed of the executable generated. Still, the differences are small enough to leave agda2hs as a viable solution. But why to use it if Coq is faster?

We can definitely say that Coq is more established when talking about practical programming tasks. Besides it being developed for decades more, the main reason has been summarised by Adam Chlipala as follows:

“None of the competition has well-developed systems for tactic-based theorem proving. Agda and Epigram are designed and marketed more as programming languages than proof assistants. Dependent types are great, because they often help you prove deep theorems without

³Technically, Coq is only the name of the software itself; the specification language is called Gallina[**coq-gallina**]. However, as the two are inseparable, the language is usually also referred to as Coq.

doing anything that feels like proving. Nonetheless, almost any interesting certified programming project will benefit from some activity that deserves to be called proving, and many interesting projects absolutely require semi-automated proving, to protect the sanity of the programmer.”[**chlipala**]

So, Coq is more suitable for writing complex proofs that are not connected to the structure of the program itself, as these have to be written in Agda by hand (without any semi-automation). However, even Chlipala notes that:

“Agda, Epigram, and similar tools have less implementation baggage associated with them, and so they tend to be the default first homes of innovations in practical type theory. Some significant kinds of dependently typed programs are much easier to write in Agda and Epigram than in Coq. The former tools may very well be superior choices for projects that do not involve any ‘proving’.”[**chlipala**]

Agda has, among others, a lighter syntax; designed for writing programs rather than proofs and making them more readable. This makes it suitable for a project in which someone wants to verify a small critical part of the program while still writing the rest of the code relatively quickly. Also, the unique features of the interactive Emacs extension makes it easier and more comfortable to work with, and integration with the rich Haskell ecosystem opens more possibilities for using it practically. Another use case is education, as the syntax of Haskell is usually more familiar to students, which might help them experiment with verified programming.

And if one has already chosen Agda, `agda2hs` clearly beats the default GHC backend (provided the code is compatible with it).

1.4 Contributions and goals

In previous papers, I already have shown the practical usability of `agda2hs` by building an exact-real arithmetic library[**acorn-article**] and then connecting it to a Qt-based C++ frontend.

Now, I am going to go a step further: not only an example is going to be presented, but an entire toolkit (called *the Agda SDK*); which makes such a process understandable and readable for others. The user-friendly documentation provided with the SDK greatly contributes to this.

During my work, I had to solve the problem of making background calculations easily interruptible; as in a GUI application, the user typically has to be provided a mechanism through which they can interrupt a process that seems to be too long. The result was a new

mechanism, called Future (after the similar C++ concept), that is going to be useful even on its own – so much that I am considering to make it available for any Haskell project, as it is not Agda-specific.

The Agda SDK, however small, can still be considered an SDK, as it is a collection of building tools and a small toolkit bundled into a minimal skeleton project. And as it has been said, “the simplicity of an SDK is just as valuable as the tools in the kit.” [**ibm-api-sdk**]

2 Technologies and decisions

Since my goal was to base the SDK on a repeatable methodology sufficient even for the needs of newcomers, I had to carefully consider what kind of elements to use for certain problems, as they had to be easy to understand, efficient and reliable. In this section, I am going to discuss some of the more important points.

2.1 Futures

This is probably the most important invention related to the SDK; so important that it might even be worth making available for any Haskell project, independently of the topic.

The concept of futures is named after the `std::future` class of the C++ standard library[`std-future`]. Similarly to that, a future represents a calculation which has not produced a result yet, enabling the holder to wait for it or detach it. However, these futures also provide for a mechanism for easily interrupting a lengthy calculation, which, along with their ease of use, makes them possibly the most advanced and convenient way for handling Haskell calculations from C++. This is especially true for those working on the frontend, who can launch, join or interrupt calculations without ever having learnt a functional language.

2.1.1 Motivation

The process through which I have come up with the concept is mostly described in the documentation[`sdk-docs-futures`]. There, I also discuss other concepts that have come across my mind, but I would like to summarise them here as well:

- Simple calls to Haskell functions exported to C (the original approach for which the FFI has been designed). These are easy to understand, especially for backend developers. However, these block the calling thread entirely, with no option for safe interruption whatsoever.
- Calls to Haskell functions wrapped into an `std::thread`. This makes it possible to run other actions while waiting for the result and even to run some actions (“triggers”) on completion. The main drawback of this approach is that it does not provide for interrupting the calculation, as such threads cannot be interrupted safely. (The problem cannot be circumvented with an `std::jthreads` either: the code inside has to manually check for a stop request[`std-jthread`], which it cannot do during a foreign call.)
- Utilising a so-called “watcher thread” on the Haskell side which, when woken up, interrupts the calculation thread with a `throwTo` call. A further question is how this watcher thread can be woken up from the frontend, for which I have had two suggestions:

- Using an OS semaphore (POSIX semaphores on Linux and Win32 events on Windows). This is very convenient if the program can only have one background calculation at the same time (as is the case with `AcornCalc`[[calc-article](#)]), as we can simply use a fixed semaphore name. If there are more than one, the developer has to think about how he could use different semaphore names for different calculations – in the worst case, they will have to pass a name every time, which makes this approach drastically more complex. (Much better is if there is a finite number of categories with at most one calculation each – then, each category can be given a single name.) Also, the code has to be written separately for each platform, and the implementation is error-prone (with open semaphores left behind in the system, the program might malfunction on the next run).

While semaphores had been useful for `AcornCalc`, I could not choose them as a general solution due to the aforementioned limitations.

- Using `PrimMVars` and the `hs_try_putmvar` function, as the documentation[[ghc-docs-ffi](#)] suggests. Since this is the “official” method, it would not be innovative to use it by itself; however, the `MVar` has to be somehow uniquely associated with the calculation. I solved this by including an interruption `MVar` and a calculation `MVar` in a single object, which can also be exported to C through a pointer. This way, any calculation can be handled on its own as a single object, with easy methods both for interruption and retrieving the result. Also, after replacing some Haskell concepts with postulates, the code can mostly be written in Agda as well.

The drawback is that due to the type variable the `MVar` contains, instantiations for different types are needed to retrieve data from the futures, which becomes especially tiresome when writing a C++ wrapper class. This means much boilerplate code for various C primitive types which largely do the same. (From these, only the ones for `int`, `void` and pointers have been written so far, but others can be written similarly.) Also, futures do not, by themselves, allow running triggers on completion; this again needs `std::thread` instances, and a correct implementation is not trivial.

2.1.2 Overview

On the Haskell side, a future (`Future a`) is an algebraic datatype containing two `MVars`: an `MVar ()` for interruption and an `MVar a` for the result. This can be exported to C by writing `StablePtrs` for the two `MVars` to a memory location (a pointer) provided by the caller. `runAsync` and `runAsyncC` create such futures, with the latter also writing it to the pointer.

On the C side, we need different `getSomethingFromFutureC` functions for all different C primitives (this is the source of the problems mentioned above). Interruption occurs by calling `hs_try_putmvar` on the first `StablePtr`, which also frees the `StablePtr`. (On interruption, an undefined value is written into the result `MVar`, using an `unsafeCoerce` – this had to be done since Haskell exceptions cannot be passed to C.) Otherwise, the `StablePtrs` have to be manually kept track of, as they are not freed automatically.

However, with an RAII-style C++ wrapper class (also called `Future`), handling these becomes much easier and safer. The constructor takes an `std::function<void(HsPtr)>` `[std-function]`, which should essentially be a Haskell callback starting the calculations and writing the `StablePtrs` to a given location. (These can most easily be constructed with lambdas, calling exported Haskell callbacks inside with the pointer and captured parameters.) From then on, its handling will be easy, with various flags and simple getter methods available, as well as an `interrupt()` call.

The destructor checks whether the `Future` has either been queried or interrupted, and if not, an error message is shown and `std::terminate` is called.

Running triggers, in a simple case, can be solved via the `TriggerFuture` class: this cannot be waited for, but instead executes triggers provided as `std::function` objects on completion. This actually waits for the underlying future and then executes triggers on an `std::thread`, while still providing the possibility of interruption.

An important problem to solve remains the chaining of futures in general: the existing object cannot be replaced with a new one in a trigger, because by deleting the future, the very function object being executed would be destroyed as well. In `EvenCounter`, I solved the problem by running normal futures in a for-loop on an `std::thread`; see `MainViewModel.cpp` at that project[[even-counter-repo](#)].

More details on futures are included in the documentation[[sdk-docs](#)].

2.2 Haskell dependencies

Many times during my work, I needed Haskell concepts that were unavailable in either `Agda` or the `agda2hs` library, usually because of their highly technical nature for which `Agda` has not been originally intended (e.g. `MVars` and `StablePtrs`). There are several ways to solve this situation:

- Writing all affected definitions in a foreign pragma (or even in native Haskell code). This is the most straightforward approach, and the one I had used in `Acorn`[[acorn-repo](#)]. It also has several drawbacks, though:
 - It is infective: once we write a definition this way, we have to write every other definition in every module depending on it as foreign pragmas.

- It makes testing harder: we only get to know whether our definitions work after running `agda2hs` and `GHC`.
- It is hard to explain and handle for newcomers, which makes it especially unfortunate for a general solution like an SDK.
- Defining the function as an Agda postulate and then including the real implementation in a foreign pragma. This approach is quick while still enabling normal work in an Agda editor (although only type soundness can be checked without running `agda2hs`). It is ideal for definitions that have no correspondents in Haskell builtin libraries and use too many Agda-foreign concepts, as expanding the `agda2hs` library would otherwise take a disproportionate amount of effort.
- If the definition has a counterpart in a builtin Haskell library, it can be added as a postulate to the `agda2hs` standard library (or even reimplemented there in Agda). This way, proofs can also be postulated or written, and we can safely depend on the behaviour of the Haskell counterpart. For convenient rewriting, only a list of module prefixes need to be added to the `agda2hs` code itself[[agda2hs-library-commit](#)]. As I have added a significant amount of code to the library that will be used by no one else, however, I have diverged significantly from the vanilla `agda2hs` distribution (which is the main reason why I use a custom branch).
- Alternatively, `agda2hs` has a feature called “rewrite rules”, through which one can provide Haskell concepts to be substituted for certain Agda definitions, via a YAML configuration file. As the documentation[[agda2hs-docs](#)] mentions, however, it is primarily meant for projects

“depending on a large library which is not `agda2hs`-compatible (e.g the standard library). In this case, you might not want to rewrite the entire library, but may still rely on it for proofs.”

For simply using Haskell definitions, it has been argued to be inefficient, as it provides no type-checking and needs duplicate information separate from the module itself[[agda2hs-compile-import](#)].

Since the start of my project, there has been a discussion on including a solution in `agda2hs` meant specifically for this purpose, with a concept called a “`COMPILE ... IMPORT`” pragma[[agda2hs-compile-import](#)]. This way, the user could add modules to their own projects starting with a `Haskell1.` prefix, and there add the definitions they want to use, either as postulates or as reimplementations. The compiler could then automatically check type soundness. It would be nice to take part in the implementation, as it could enable me to switch back to the vanilla, up-to-date version of `agda2hs`.

2.3 C++ class hierarchy

The `HsAppStateWrapper` class is a typical example of RAII in C++: it holds a `StablePtr` to an `AppState` object in the Haskell backend, creating it in the constructor and freeing it in the destructor. How this object itself is embedded in other classes, however, is not trivial. Containing a full `HsAppStateWrapper` in the view model itself makes sense, as it only contains a `StablePtr`, which is technically a `void*` (so it has the same size as a pointer). The view, however, contains a reference to the view model instance, as that consists of several non-trivial objects. I have considered to hide the app state wrapper behind a reference, too, for the sake of consistence; but I have dropped the idea due to the obvious inefficiency.

2.4 Installation

Here, as dependencies are platform-dependent, installation methods definitely have to be platform-dependent, too.

The two options were:

- installation scripts written in the native shell language which download and install dependencies in place;
- pre-built zip files.

I preferred installation scripts, as they are easier to be kept up-to-date (by simply changing the version numbers) and give the option to skip certain components in case the user has already installed them (this is especially important for Qt). However, this needs more extensive testing, has more dependencies (which might have to be installed separately) and may be sensitive to changes to the syntax or behaviour of the default shell. For Linux, scripts might also hamper distribution compatibility; for example, the script has only been tested on Ubuntu.

Another problem is that dynamically built Qt versions (which are the default) make it much harder to deploy programs, especially for computers not having Qt pre-installed. To avoid this problem, Qt can also be built statically. However, as the packages provided on Qt Company servers are all dynamically linked, this means Qt has to be compiled on sites, which is both storage- and time-consuming. This could be solved by a bundle containing a pre-built static Qt build, but for now, I stick with dynamic versions.

3 The SDK

Here, I only wish to give a quick overview of the structure and usage of the SDK, as the documentation[[sdk-docs](#)] already describes the details. Please check it out if you have any questions about the inner workings of the SDK (e.g. how project files work). Also, I recommend reading my previous article on AcornCalc[[calc-article](#)], a proof-of-concept project.

3.1 Installation

The Agda SDK is available as a clonable GitHub repository[[sdk-repo](#)], which contains:

- two installation scripts (one for Unix `sh` and one for PowerShell);
- a project skeleton, with pre-written project files, main source files and basic structure;
- several builtin tools (e.g. for asynchronous Haskell calculations or easier C exports);
- a documentation.

In essence, the custom `agda2hs` distribution used by the SDK, which has several extensions of its Haskell library, is also part of the SDK.

Installation of the necessary tools (that is: GHC, the above custom version of `agda2hs` and Qt) can be done simply by cloning the repository and running the OS-specific installation script. All components can be skipped in case they have already been installed manually.

3.2 CMake projects

The default structure of the project skeleton actually contains three fully functional CMake projects: one for the backend, one for the frontend and a root project embedding both. This enables the developer to compile only the backend (e.g. when they do not need a GUI frontend), only the frontend (for a globally installed backend library) or both together.

See the corresponding parts of the documentation for more details.

3.3 Backend

Compilation of the backend technically happens via:

- calling `agda2hs` on a file called `All.agda`, containing references to Agda and Haskell files to be included;
- then calling GHC on the previously generated `All.hs` file, or on `Main.hs` for a command-line binary (or on `TestMain.hs` for QuickCheck tests).

In the first step, the Agda typechecker built into `agda2hs` verifies Agda source code and compiles those definitions that are needed, along with foreign pragmas, into machine-generated (but human-readable) Haskell modules. GHC then compiles these into a static library or an executable.

For this to work, `All.agda` has to be continuously updated: files to be checked by the typechecker are imported on the Agda side, while those to be compiled by GHC on the Haskell side. See the documentation for details.

Communication with the frontend is done via the Haskell foreign function interface[[ghc-docs-ffi](#)]. Such can be included in any source file (the skeleton and example projects put them into `Interaction.agda`); the corresponding C type signatures should be copied into headers in the `include` folder. Again, see the corresponding part of the documentation.

Pre-written tools can be found under `src/Tool`. These include some data structures tailored for `agda2hs` usage, definitions for usage in proofs, postulates for foreign C types and the backend part of futures.

3.4 Frontend

The frontend is, by default, a Qt-based C++ GUI application (but technically, any C++ or C code can be attached to the backend). It can be compiled via CMake, called either on the entire project or just the frontend (in the latter case, it will look for a globally installed backend). It is advisable to compile from Qt Creator, as it finds the appropriate Qt installation and makes the process more transparent. Files can be registered to CMake via local `CMakeLists.txt` files placed in subfolders.

The tools provided in the skeleton project include wrappers for backend concepts (including futures), as well as a default structure for a view model and a view.

3.5 Testing

Before beginning with verifying a program, it is often useful to already have some tests at hand. The Agda SDK provides three mechanisms through which one can write test cases quickly:

- Agda tests, evaluated by the typechecker before compilation, can actually be placed anywhere in the backend, but the recommended location is under the `Test` folder. An example is defining a simple equality between two concrete values (e.g. a function call and its result) with `refl`.
- Haskell QuickCheck tests, run by issuing `cabal test`, can be written under `Test/Haskell` in the backend. These are based on Boolean predicates that are then tested against a huge amount of randomly generated inputs.

- Ordinary Catch2-based C++ test cases can be added under `src/Test` in the frontend.

3.6 Documentation

The documentation is written in Sphinx and hosted at *Read the Docs*[[sdk-docs](#)]. It uses informal English and explains the concepts and workings of the SDK in an easy-to-understand way, meant for helping readers get familiar with the toolkit as quickly as possible.

3.7 Example projects

3.7.1 The skeleton project

This is basically the SDK itself, with the installation scripts and the pre-written skeleton of a project. Wrapper classes and tools like futures are also included.

3.7.2 EvenCounter

An example project, containing an integer counter that is proven in Agda to contain only even numbers. Concrete values can be added from the GUI, and another button increases the counter with 2 every second, for five seconds, in an interruptible way. Used for demonstrating capabilities of the SDK and futures.

It can be found on the `even-counter` branch[[even-counter-repo](#)] of the skeleton repository, as there are many cases where changes in the project are cherry-picked to the skeleton project (or vice versa).

A binary for [Windows](#) is available under `csimmaviktor.web.elte.hu`.

3.7.3 AcornCalc

A previous project which utilises many of the concepts of the Agda SDK, and which actually serves as the basis of the code. It is an example for a practical, large-scale use case: it embeds exact-real arithmetic into a GUI calculator application. For details, see the repositories for the backend[[acorn-repo](#)] and the frontend[[calc-repo](#)], as well as my previous articles[[acorn-article](#), [calc-article](#)].

For this, there is both a [Linux](#) and a [Windows](#) executable; these were linked using a static version of Qt (and hence much smaller and easier to handle).

4 Benchmarks

I have for long owed a detailed comparison of the performance of various code extraction solutions, showing the viability of `agda2hs` among them. Now, I pay off the debt.

4.1 Hardware configuration

For development and testing, I used 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 24.04 and Windows 10 22H2 installed. Benchmarks have been done on Ubuntu.

4.2 Test suite

The exact files I used can be found at the GitHub repository *viktorcsimma/benchmarks*[\[benchmarks-repo\]](#).

The source files themselves are based on code which András Kovács has previously used for similar benchmarks`[smalltt]`; more precisely, `conv_eval.agda` and `conv_eval.v`. There, he defines a lambda-based representation of natural numbers and binary trees, then forces the typechecker to decide on the equality of certain complex expressions, or to evaluate them. This makes it possible to measure the efficiency of various typecheckers.

I rewrote the definitions with Haskell-style type classes; thereby circumventing the need for `--type-in-type` and enabling translation to Haskell (but worsening code readability). For compilation tests, I added code which evaluates expressions on an extracted function; I then compiled and ran that function to measure the time needed for translation, compilation and running.

Coq has a builtin tool (the `--time` flag of `coqtop`) that outputs the time spent on individual commands; I used it for the appropriate benchmarks on type-checking. For Agda, I simply used the `time` command of Bash.

Before each measurement, I commented out every unnecessary definition and checked or compiled only the one needed – I had noticed that it often makes elaboration several seconds faster if previous, less complex objects have been checked before.

4.3 Interpretation

All the measurements in the tables are in seconds; smaller values are better.

Where three values appear with slashes, the measurements are for the `-00`, `-01` and `-02` flags of GHC, respectively.

At runtime, *ME* means “*memory exhausted*” (the process was killed by the OS due to it consuming almost all of the system memory).

4.4 Type-checking

For comparing type-checking performances, I made both Agda and Coq check equalities of complex natural numbers and trees, as András had done before.

	Agda	Coq
NatConv1M	ca. 0	3.430
NatConv5M	19.220	16.807
NatConv10M	39.294	41.828
TreeConv15	0.098	0.023
TreeConv18	0.061	0.039
TreeConv19	0.051	0.039
TreeConv20	14.108	1.471
TreeConv21	26.216	2.792
TreeConv22	53.148	5.515
TreeConv23	102.571	10.830

For natural numbers, the result is inconclusive; for trees, however, Coq is about ten times faster.

4.5 Interactive evaluation

This part is about evaluating expressions while in the interactive mode (so by the proof assistants themselves). I also used the code of András here, which evaluates trees via a function called `forceTree`: this puts Boolean true values in the leaves and ANDs them together, thereby forcing the evaluation of the entire tree. As noted at the original benchmark repository, there is no way in Agda “to only force the normal forms, without doing printing or conversion checking”[`smalltt`]; therefore, I have left those Coq tests out.

Here, the four contestants are:

- the interactive mode of Agda (the command-line version, for the sake of measuring time more easily);
- three different evaluation tactics of Coq:
 - `lazy`,

- `compute`
- `vm_compute`.

For Agda, I used the terminal command with the `--interactive` option (see `agda.interactive.sh` in the repository[**benchmarks-repo**]); for Coq, I again relied on the builtin tool.

	Agda	Coq (lazy)	Coq (compute)	Coq (vm_compute)
ForceTree15	0.471	0.135	0.148	0.005
ForceTree18	1.204	1.005	0.687	0.050
ForceTree19	2.110	1.796	1.167	0.098
ForceTree20	3.732	3.503	2.129	0.194
ForceTree21	7.177	6.940	3.974	0.581
ForceTree22	14.037	13.374	7.905	1.217
ForceTree23	27.573	26.482	14.762	2.194

Agda is on par with Coq’s lazy tactic, which works similarly. However, the other tactics of Coq (especially `vm_compute`, which works by compiling to the bytecode of an abstract machine) are faster with magnitudes. The flexibility of Coq triumphs over Agda in this use case.

4.6 Compiling and running

This is the most interesting part from our point of view. The question is whether `agda2hs` really is competitive compared to existing solutions: the default GHC backend of Agda (MAlonzo) and the OCaml extraction tool of Coq.

Here, I evaluated either a natural number or a tree (with `forceTree`) on the main function that was compiled to an executable. For OCaml, a main function had to be added by hand in order to get the source code compiled; see `prepare_ocaml.sh` in the repository[**benchmarks-repo**].

4.6.1 Translation to the underlying language

	Agda MAlonzo	agda2hs	Coq OCaml extraction
n2	1.162	0.610	0.162
n100	1.172	0.590	0.156
n10k	1.173	0.600	0.140
n100k	1.195	0.559	0.152
n1M	1.163	0.582	0.151
n5M	1.133	0.581	0.146
n10M	1.193	0.560	0.153
n100M	1.162	0.572	0.144
n500M	1.807	0.955	0.153
n1G	1.837	0.976	0.149
t15	1.101	0.621	0.164
t18	1.090	0.613	0.148
t19	1.142	0.615	0.149
t20	1.081	0.592	0.143
t21	1.143	0.603	0.145
t22	1.091	0.602	0.139
t23	1.131	0.602	0.145

As expected, translation times seem to be dependent on code structure, rather than the concrete values used in the examples. `agda2hs` is about twice as fast as `MAlonzo` is; although the reason for this might be that `MAlonzo` also needs to compile the files of the Haskell library to its own Haskell format. `Coq OCaml extraction` is by far the fastest at this phase, with translation times about four-six times lower than those of `agda2hs`.

4.6.2 Compilation to executables

	Agda MAlonzo	agda2hs	Coq OCaml extraction
n2	1.303/1.819/1.823	0.915/0.934/0.950	0.017
n100	1.306/1.791/1.900	0.850/0.901/0.929	0.021
n10k	1.351/1.743/1.847	0.885/0.929/0.910	0.016
n100k	1.334/1.787/1.818	0.868/0.914/0.924	0.022
n1M	1.291/1.823/1.853	0.889/0.906/0.889	0.019
n5M	1.339/1.755/1.829	0.871/0.911/0.902	0.024
n10M	1.279/1.767/1.866	0.848/0.931/0.887	0.023
n100M	1.334/1.805/1.869	0.878/0.886/0.925	0.018
n500M	2.002/2.775/2.824	1.335/1.488/1.501	0.029
n1G	1.985/2.752/2.886	1.393/1.434/1.449	0.017
t15	1.294/1.820/1.848	0.881/0.971/0.941	0.022
t18	1.308/1.807/1.826	0.890/0.925/0.948	0.021
t19	1.270/1.799/1.857	0.912/0.923/0.925	0.021
t20	1.268/1.806/1.826	0.872/0.913/0.917	0.022
t21	1.326/1.755/1.865	0.882/0.952/0.941	0.020
t22	1.312/1.796/1.856	0.866/0.922/0.927	0.017
t23	1.352/1.737/1.870	0.888/0.939/0.922	0.018

Running GHC on agda2hs-generated code is again significantly faster than on the type coercion-based format written by MAlonzo (although again, the reasons might also include the larger number of files).

However, the OCaml compiler runs almost instantly on the source code extracted from the Coq module. Note that OCaml extraction only outputs the definitions that are really going to be used during computation.

For MAlonzo, GHC optimisation flags add an overhead of around 30–40%; in the case of agda2hs, this is much less significant.

4.6.3 Running the executables

	Agda MAlonzo	agda2hs	Coq OCaml extraction
n2	0.014/0.014/0.014	0.014/0.014/0.013	0.003
n100	0.013/0.013/0.014	0.013/0.013/0.012	0.006
n10k	0.012/0.014/0.012	0.014/0.012/0.014	0.003
n100k	0.023/0.024/0.013	0.034/0.013/0.014	0.004
n1M	0.232/0.160/0.185	0.224/0.141/0.150	0.029
n5M	0.951/0.832/0.813	0.766/0.191/0.181	0.073
n10M	2.039/1.656/1.670	1.560/1.410/1.388	0.156
n100M	21.008/18.686/18.721	15.242/2.290/2.291	1.502
n500M	ME	ME	7.786
n1G	ME	ME	15.224
t15	0.013/0.024/0.024	0.013/0.013/0.024	0.009
t18	0.054/0.033/0.044	0.034/0.033/0.035	0.049
t19	0.095/0.063/0.075	0.044/0.043/0.054	0.108
t20	0.184/0.124/0.113	0.063/0.074/0.083	0.188
t21	0.314/0.224/0.244	0.134/0.124/0.145	0.344
t22	0.604/0.423/0.404	0.255/0.244/0.234	0.709
t23	1.153/0.794/0.793	0.444/0.434/0.423	1.373

Here, ME means the program exhausted the system memory and was killed by the OS.

As it can be seen, there is again a huge difference between MAlonzo and agda2hs; especially with optimisation, as it has a greater effect on agda2hs-generated code than that of MAlonzo. Tree evaluation, however, is a point where agda2hs-compiled executables even beat those extracted from Coq.

4.7 Conclusion

The type-checking performance of Agda and Coq has already been compared, and as András has concluded, Coq is definitely superior to Agda in this field. However, I was more interested in how the various backends would perform.

The OCaml extraction feature of Coq is many times faster than agda2hs in both translation and compilation. At run-time, results were not so conclusive, with multipliers being much smaller. For tree evaluation, Agda even beat Coq with both backends. It can also be seen that the `-O1` flag of GHC often helped accelerate executables significantly.

agda2hs outperformed the builtin GHC backend of Agda at practically every field; although I have to note that they serve a different purpose: MAlonzo is for compiling arbitrary Agda code to Haskell, while agda2hs only supports a Haskell-compatible subset of Agda. Having said this, most practical programs do not need dependent types, and with a bit of caution, most definitions can be kept agda2hs-supported.

It might seem like if the superiority of Coq at almost every field made Agda obsolete. Please note, however, that Agda has other advantages compared to Coq; namely, a cleaner syntax designed more for writing programs rather than proofs and a unique, helpful text editor mode[**agda-vs-coq**]. For those wishing to enjoy these features, agda2hs might be a good choice.

5 Summary

I have described a software toolkit for writing Agda-based, reasonably fast backends and connecting them to C++-based frontends. Through this, I could provide a repeatable, stable methodology for writing such applications, which can be mastered by anyone who is familiar with Agda and C++. For frontend development, contributions are possible even for those having no experience with functional languages whatsoever. Through benchmarks, I could establish that the underlying compiler is viable compared to other solutions, especially when taking the unique advantages of Agda to account.

In addition, I have provided a description of an innovative way of managing Haskell calculations in C++, via objects called futures. This has a value that is independent from the SDK and Agda itself.

Hopefully, the project will lead to a greater interest in the language, and encourage those already working in it to explore its so-far hidden potential. Even if only one developer finds the SDK and creates something with it to their own entertainment, I can say the effort put into creating it has not been futile.