An agda2hs-compatible representation of exact real arithmetic

For the Scientific Students' Circle Conference of December 2023
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 Collegium

Budapest, 2023

Abstract

Could an exact real arithmetic library be both computer-verified and efficient? This article describes **Acorn**, a high-performance in-development Agda implementation of exact real arithmetic described by Krebbers and Spitters in 2013[1].

Floating point numbers used by modern CPUs are only approximations of real numbers; therefore, we might lose precision while performing operations with them. In extreme cases, the whole computation might become useless. However, real numbers can be exactly represented on a computer using rational-valued functions. Since Russell O'Connor's publication of his monadic reals[9], even solutions verified by a computer can produce programs of a usable speed; however, they are still much less efficient than their non-verified counterparts.

Acorn is based on mathematical foundations and concepts of the CoRN analysis library[11] (which has been written in Coq). However, it takes a new approach to the topic and aims to popularise the field of verified exact-real arithmetic by *first* showing a usable implementation, with most proofs filled in only after this has been achieved. This means that most parts of the library are not yet verified; however, there already exists a framework to work in. The goal of full compatibility with the new *agda2hs* compiler[2] makes the code not only much more efficient but also easier to understand and use. The familiar, Haskell-style environment of Agda, combined with CoRN's type class-based algebraic operators, allows users of the library to get started quickly with writing useful software that is based on reliable calculations proven to be correct – this could popularise the concept among the wider programming community.

Tests show that Acorn can compete in efficiency with older solutions, while being verifiable in Agda. This way, there no longer need to be a verified and a separate efficient implementation: the latter can be generated automatically.

In the long run, even Bishop's constructive analysis[4] could be formalised in the system, enabling users to extract programs from every existence proof.

Section 1 provides a short historical overview of the evolution of computer-verified exact-real arithmetic and arguments for creating a new library based on previous solutions. After that, *Section 2* briefly summarises my previous work on the topic to further highlight the reasons why this project has been born. Finally, *Section 3* describes Acorn itself; with the most important design decisions, the basic structure of the library, its current state and the steps that are going to follow.

Acknowledgements

First of all, I would like to express my admiration for the work of *Errett Bishop*, *Luís Cruz-Filipe*, *Russell O'Connor*, *Robbert Krebbers* and *Bas Spitters*; all of whom I have cited in the article. It is incredible how much this field has evolved; even if largely unnoticed by the general public. Hopefully, my results will help to put it into the spotlight.

I am especially grateful to my supervisor *Ambrus Kaposi*, who not only has taught me Agda, but also gave me guidance and comments regarding the course of my research. I have spent my summer practice at his department, and his support for working on what I was excited about was crucial for me to succeed. Here, I would like to mention *András Kovács* for his valuable suggestions; he was the one who brought the agda2hs project to my attention. I should not forget the leaders of the *Informatics Workshop* of *Eötvös Collegium* either, namely *Tamás Kozsik* and *Levente Lócsi*, who have provided me advice, support and a great athmosphere to work in. (Actually, I met Ambrus here.)

The next person I would like to thank is *Jesper Cockx*, the maintainer of *agda2hs*. As I began to contribute to the project while working on the Bishop formalisation, he took the time to review my issues and pull requests, advised me on technical details and conventions and fixed some of the issues that hindered my work by himself. Without him, there would not be a compiler quick enough to base this library on it.

The reason why I could start working so quickly was a project formalising Bishop's constructive analysis in Agda. I thank *Zachary Murray* for allowing me to use the library he had written as a bachelor thesis at Dalhousie University. Had I started from scratch, I would probably still be thinking about how to calculate limits of Cauchy sequences.

Furthermore, I think this work could not have been done without my friends and acquintances at *Eötvös Collegium*; they were there when I needed help, motivated me to do better and often made a day happier with a little small talk or a simple smile. Honestly, despite not being well-known, I am convinced this is the best place for me to live and study at – and for many others.

Here, I express my special gratitude to *Petra Orendács*, who took upon herself the exhausting task of correcting my English. Without her proofreading, studying this article would probably be a much more unpleasant experience. [1]

Actually, I would also like to mention my students in the college and at the university, who have shown me that I am not alone with my enthusiasm about Agda. After holding tutorials for them, I always felt more motivated to keep on researching.

And, of course, I cannot finish my acknowledgements without thanking my family for their persistent support; not only with healthy and delicious food (so that I do not live on cans all the time), but also with advice, patience and genuine love given both in good and bad times.

[1] I am going to buy her a box of chocolates soon. I promise.

Contents

1 Motivation

1.1 A brief historical overview

1.1.1 Bishop's constructive analysis

Constructive mathematics mainly differs from its classical counterpart by only accepting constructive existence proofs (hence the name). In other words, we can only prove an object's existence by actually constructing the object and proving that it fulfills the given requirements[3]. This meant a new way of approaching the familiar concepts of analysis. For example, if one proves the convergence of a series, they actually give a method (in fact, an *algorithm*) to calculate the limit, and then prove that it really is the limit. This means that one can (at least theoretically) compute the limit of any series which they prove to be convergent. The same goes for inverses, derivatives, and integrals.

However, the first usable constructive formalisation of analysis was published way after constructivism's emergence in the early 20th century. The American mathematician Errett Bishop published his book *Foundations of Constructive Analysis* in 1967, which was later expanded with the help of Douglas Bridges and republished in 1985[4]. Bishop proved many of the results of classical analysis in a constructive environment. However, there are some theorems that could not be proven constructively. In this case, there were often two constructive versions: one with the same hypotheses and a weaker result, and one with stronger hypotheses and the same result[5]. (An example for the former approach is Rolle's theorem: it does not say that we can found a point at which the derivative is 0; only that *for any* $\varepsilon > 0$ we can found a derivative less than or equal to ε.)

While building analysis until integration and important functions in Chapter 2, Bishop takes naturals, integers and rationals "for granted" – in fact, in the previous chapter he quotes Kronecker saying *"the positive integers were created by God"* and states *"[they] and their arithmetic are presupposed by the very nature of our intelligence"*. After this, he defines a *real number* as a sequence (x_n) of rational numbers for which it holds that

$$\forall m, n \in \mathbb{N}^+ : |x_m - x_n| \leq m^{-1} + n^{-1}.$$

Two real numbers ((x_n) and (y_n)) are *equal* if

$$\forall n \in \mathbb{N}^+ : |x_n - y_n| \leq 2n^{-1}.$$

Later, it is proven that the nth member of (x_n) is within the n^{-1} radius of the real number represented by the sequence. Thus, each member can be considered to be an approximation of a real, with the precision of n^{-1}.

This approach worked perfectly when formalising constructive analysis on paper. How-

ever, when implemented on a computer, this representation turns out to be of little use. Consider the definition of the sum of two reals x and y:

$$x + y := (x_{2n} + y_{2n})_{n=1}^{\infty}$$

We need to take the $2n$'th member of both sequences to have n^{-1} as the sum of their precisions. But there is a greater problem than that. Representing the rational numbers as the pair of the numerator and the denominator means that for each addition, a rational addition is needed. To add two rationals, we need (at least in the general case) to multiply the denominators with each other, then to multiply the numerators with the other number's denominator. On arbitrary length integers, multiplication can be a very time-consuming operation, and (as we will see later) this by itself renders computations based on the theory unusably slow, even in an efficient functional language like Haskell.

Of course, we need to point out that Bishop's work predated the widespread usage of computers – at the time, they were large, expensive, and slow.[5] Bishop himself notes in Chapter 1 that his main focus has been to create a comprehensive formalisation rather than to make it efficient:

> The transcendence of mathematics demands that it should not be confined to computations that I can perform, or you can perform, or 100 men working 100 years with 100 digital computers can perform. Any computation that can be performed by a finite intelligence [...] is permissible. This does not mean that no value is to be placed on the efficiency of a computation. [...] Mathematics should and must concern itself with efficiency, perhaps to the detriment of elegance, but these matters will come to the fore only when realism has begun to prevail. Until then our first concern will be to put as much mathematics as possible on a realistic basis without close attention to questions of efficiency.[4]

However, he appears to have thought of the new direction computers could give to constructive analysis. In Appendix B, he writes:

> It is clear that many of the results in this book could be programmed for a computer, by some such procedure as decribed above. In particular, it is likely that most of the results of Chaps. 2, 4, 5, 9, 10, and 11 could be presented as computer programs. [...] As written, this book is person-oriented rather than computer-oriented. It would be of great interest to have a computer-oriented version.[4]

This means the idea to make constructive proofs runnable on a computer is not new; it simply could not be realised in Bishop's time. Since then, however, both theory and hardware

have changed tremendously.

1.1.2 Later developments

Here, I would like to give a short summary of the most important developments that led to the theory formalised in this project. Much of the work has been done at the University of Nijmegen in the Netherlands.

- The FTA project, whose original aim was to formalise the Fundamental Theorem of Algebra (FTA) in Coq, built an algebraic hierarchy on which almost all later work is based. It introduced the type `CReals`, defined by axioms and then implemented as a set of Cauchy sequences of rational numbers. The publication on the real number implementation was published in 2000.[7][8]

- The original version of the C-CoRN project, as described by Luís Cruz-Filipe in 2004, was the first one to push for a computable and computer-proven formalisation of constructive analysis. This uses the Bishop-style definition instead of simply relying on Cauchy sequences in general. Cruz-Filipe conducted experiments on computing approximations of $\sqrt{2}$ and found that the 12th approximation (with a precision of $\pm 1/12$) does get computed, although after more than 30 hours. He expressed hope that *"in a not too distant future it will be possible to run the extracted program in reasonable time, even if never at a speed that can rival computer algebra systems"*.[8]

- A breakthrough came with Russell O'Connor's monadic reals in 2005 – essentially, he created the first model runnable and usable in practice[9]. He introduced several new concepts:

  - The usage of a more general concept of *metric* and *prelength* spaces instead of the rationals as the function range. For the definition of metric spaces, he introduces the equivalence relation $B : \mathbb{Q}^+ \to X \to X \to *$, which corresponds to two elements of the space being within a distance of a given rational to each other. (The concrete implementation still uses rationals given by pairs of arbitrary-length integers.)
  - Regular *functions* instead of Bishop's regular *sequences*. A function $x : \mathbb{Q}^+ \to \mathbb{Q}$ is regular if
    $$\forall \varepsilon_1 \varepsilon_2 : B_{\varepsilon_1+\varepsilon_2}(x(\varepsilon_1), x(\varepsilon_2)).$$
    This extends regular sequences by allowing not only approximations with the precision of rationals of the form $1/n$, but of any positive rational.
  - The $\mathfrak{C}$ monad, representing the completion of a metric space. By defining it as a monad, it allows us to lift *uniformly continuous* functions and operators

on the rationals to the reals. This makes our work much simpler and gives us new opportunities (ordering relations are decidable here, while not on the reals). Perhaps the most representative concept of the completion monad is the *map* function with the type signature $(X \to Y) \Rightarrow (\mathfrak{C}(X) \to \mathfrak{C}(Y))$:

$$map(f) := \lambda x.f \circ x \circ \mu_f.$$

Here, μ_f is the modulus of continuity; this ensures that the return values for two rationals within an ε ball of each other will also be no further than ε. This is also why this method *only* works on uniformly continuous functions.

- The *compress* function that "makes the rational outputs of regular functions simple while keeping the real number it represents the same". This greatly increases the efficiency of the computation. Russell defines it as:

$$compress(x) := \lambda\varepsilon.approx(x(\frac{\varepsilon}{2}), \frac{\varepsilon}{2}),$$

where $approx(q, \varepsilon)$ is the simplest rational number in $[q - \varepsilon, q + \varepsilon]$. (The latter is built into Haskell's `Data.Ratio` module as `approxRational`.) Simply put, it searches for approximations of the values "good enough" to keep the function regular, but more easily representable.

A Haskell implementation of the library competed in the *"Many Digits" Friendly Competition* in October 2005 and could compute "thousands of decimal digits within minutes or seconds for many of the problems", while competing with much more efficient, but not computer-verified solutions. However, O'Connor noted that the "Era" exact real library used dyadic rationals in its representation, which made it more suitable for difficult problems.[9]

- Robbert Krebbers and Bas Spitters submitted an enhanced implementation of O'Connor's reals in 2011. This "speed[ed] up the basic operations of O'Connor's implementation by a 100 times"[1].
  - They extensively used type classes (which were back then relatively new in Coq). This makes working with common operators much simpler, as there is no need to use a separate name for them for different number types.
  - Using type classes, they implemented a detailed algebraic structure similar to that of FTA.
  - They defined the type class of *approximate rationals*, on which only an approximate division operator is expected. This enables the use of *dyadics* as the underlying representation, which tremendously improves performance.

– The *approx* function now only needs to produce an arbitrary `AppRationals` element between given bounds; the result may vary with different implementations. On the dyadics, the function shifts the mantissa to the right, losing a given amount of bits from precision. This simplifies the process.

Like O'Connor, they have created both a Coq and a Haskell[10] implementation; the latter has often provided me inspiration for solving performance problems.

The C-CoRN project today relies on the work of Krebbers and Spitters[11], as does the library to be presented in this article.

1.2 Reasons to create a new library

C-CoRN has become an extensive library, reaching as far as the fundamental theorem of analysis in formalisation and relying on fast, computable real numbers. Therefore, one may well ask why it is needed to reimplement a largely simplified version of it in Agda.

- Nine years after the Dutch authors published their paper, work on a new compiler called *agda2hs*[2] began. agda2hs takes a pre-formatted Agda code subject to some rules and then turns selected runnable parts of it into human-readable Haskell. Through this, not only the generated code will be simpler and more efficient, but the programmer also knows what exactly is going to happen in the background. As far as I know, the project has not yet been used in creating efficient exact real representations.

- Agda is designed for programs rather than proofs, as is my initiative. As I am going to explain in section 3, my priority is to make the library available to others; most proofs are going to be added later. Agda supports this approach better then Coq does; providing a clear, Haskell-like syntax that makes it more convenient to write useful software. Of course, adding the missing proofs is going to be a much harder work than it would be in Coq; but by then, the project can gain a momentum by being able to demonstrate its viability, and an open-source collaboration might begin.

- Krebbers and Spitters wrote both a Haskell and a verified Coq implementation. Computing circa 3000 decimals of $\sqrt{2}$ took 0.2s in Haskell (without compiler optimisations) and 7.4s in Coq. Laurent Théry has suggested that the reason for this might be the inefficient representation of fast integers in Coq[1]. Using the arbitrary precision integers of Agda (and Haskell) might help.

Note that although I admire Bishop's work and am excited about constructive mathematics, this paper is not "a piece of constructivist propaganda", as he described his book. My goal is to provide a verifiable and still fast implementation of real numbers (and in the long

run, the foundations of analysis) where every existence proof gives an algorithm terminating in reasonable time. In this process, I view constructive mathematics mainly as a powerful tool useful for the computation of results in practice. Reigniting discussion about the foundations of logic is desirable, but only secondary to the original goal.

I would like to emphasise this because Bishop felt most mathematicians did not take his work seriously[5], and this might have been because it seemed to be no more than an intellectual exercise being of no practical use. All the work of the above-mentioned people, and hopefully mine too, provides evidence that this is not the case. I believe that by first demonstrating the usefulness of the theory before working on the proofs themselves, there will be a better chance that it awakens the interest of the mathematical and programming community, and work on a verified, fast computable analysis system will begin.

2 My previous work

2.1 Extending Zachary Murray's Bishop formalisation

I started to work on the topic in June 2022. My first thought was to formalise Bishop's analysis in CCTT, a new cubical-compatible language written by András Kovács and others[12]. Then, I found Zachary Murray's implementation in Agda, written in the spring of 2022[13]. He kindly allowed me to use his work; so my first project was the continuation of that library.

Murray successfully formalised many of the fundamental concepts of analysis; from the real numbers and their properties through the uncountability of the reals until sequence limits and even some results on supremums (which aren't as straightforward as in classical analysis). During my work on extending the library, I:

- Proved the alternating series test (and gave a definition of π using it).
- Switched from using $Fin\ n \to \mathbb{R}$ for finite sequences to sigma indices ($\Sigma\mathbb{N}(\lambda k \to k \leq n)$); as they are much more convenient to work with in proofs (and the indices can easily be converted to natural numbers).
- Written methods generalising certain theorems on $\mathbb{N} \to \mathbb{R}$ to finite sequences.
- Formalised Bishop's definition of continuity (which is, actually, uniform continuity)[4] and proved the constructive version of the *extreme value theorem* (called *Weierstrass theorem* in the library).

My results can be found on the main branch of the `bishop` repo[14]. I would not like to write much about this branch of the library; as its main focus was to continue the mathematical formalisation, not to create a runnable code. However, a later branch is interesting for us; this is what the next subsection is going to be about.

2.2 An agda2hs-compatible rewrite

It would have been desirable to create a version of the library where the numbers we work with can actually be calculated. A concrete goal was to be able to approximate the irrational numbers e and π; as they were the only irrational numbers at hand. However, all computations with them got stuck; both when running in an interpreter and when compiled with *MAlonzo* (the default compiler which turns arbitrary Agda code into Haskell). I thought this might be because of the highly complicated code MAlonzo generates; so I started working on an agda2hs-compatible version of the library. This can be found on the agda2hs branch of the repo[14].

While doing so, I have made some valuable contributions to the agda2hs project (those interested can check them out at the repository of the compiler[2], among the pull requests). Still, the main goal could not be reached: any computations with e stalled.

On the other hand, having generated a human-readable Haskell code, I was able to analyse (with the help of the profiling options of GHC[17]) what consumes the most time during the attempt to approximate e. The culprit was the addition operation, defined in a Haskell-style pseudolanguage as[4]:

```
(+) :: Real -> Real -> Real
x + y = \ n -> x (2 * n) + y (2 * n)
```

This seems to be innocent, but involves the addition of two rationals, which are represented by the ratio of two arbitrary-length integers. These need to be multiplied by each other to get a common denominator; not only does this produce huge numbers, but also takes too much time. Seeing that the representation itself was the problem, I moved forward reading about more up-to-date solutions (as Ambrus had advised me).

That was when I came across the work of Russell O'Connor, Robbert Krebbers and Bas Spitters, and decided to create an agda2hs-compatible representation of it, being mathematically proven and efficient at the same time.

3 The Acorn library

The current code of Acorn is available on GitHub[15].

3.1 Design decisions

3.1.1 Implementation of the reals

This follows the concept already presented by Krebbers and Spitters[1].

- First, a type class called `AppRationals` is defined. This differs from the usual notion of rational numbers in that we only expect an approximate division operator (with a precision we can choose), and an operator approximating a number with a given precision (that we choose, too).
- Afterwards, we implement the completion monad described by O'Connor[9], and define the real numbers as the completion of any type in the `AppRationals` type class.
- In `RealTheory.Reals`, we define the operators on the reals (usually using *bind* and similar functions).
- The exponential and trigonometric functions are then defined.
  - We do this with the help of appropriate power series, which are proven to be alternating series on a small rational interval. They produce valid regular functions there; furthermore, the theorem mentioned as Theorem 33 by O'Connor[9] states that
  $$\sum_{i=0}^{\inf} a_i = x \text{ where } x(\varepsilon) := \sum_{\varepsilon < a_i} a_i.$$
  In other words, to compute the ε-precision approximation of the limit, we can simply sum the elements until we found the first one that is smaller than or equal to ε.
  - We extend them to all rationals by shifting the parameters to the given intervals using specific rewrite rules. For example, for the exponential function, we use:
  $$e^x = \frac{1}{e^{-x}}$$
  $$e^x = e^{\frac{x}{2}} \cdot e^{\frac{x}{2}}$$
  And for sine:
  $$sin\, x = 3 \cdot sin\frac{x}{3} - 4 \cdot (sin\frac{x}{3})^3$$

We apply these rules until we shrink the parameter under a given bound in absolute value, which is usually much smaller than what would be needed in theory. This way, the alternating series will get faster under ε, resulting in a massive speed improvement. K&S recommend 2^{-50} in general and 2^{-75} for higher-precision calculations[1].

  - Finally, we prove them to be uniformly continuous on an appropriate interval (often dependent on the parameter itself!) and lift them to the reals using `bindC`.

- The concrete representation of `AppRationals` we are going to use is the completion of dyadic rationals, defined in `Implementations.Dyadic`.

More details can be found both in the cited article[1] and in the code. In this paper, I am focusing on the code itself and technical issues, as the mathematical foundations have already been laid by previous articles.

3.1.2 Type classes

In general, the library is being built using Krebbers' and Spitters' type class-based approach; resulting in a very convenient usage of common operators for any type we work with. Here are some points where I have diverged from the structure of the Coq library[11].

- I have left out most of the algebraic structures with only one operator (semigroups, groups etc.). The reason was that it would have been technically difficult to apply them to the addition and multiplication operators of semirings. Instead, all rules associated with these structures are defined separately for `_+_` and `_*_` in the `SemiRing` class.

- We introduce most of the operators among with their properties. To avoid clutter, we generally avoid classes which only bring in an operator without properties (like `Plus`, `Mult` or `Zero` in CoRN). There are some exceptions to this where we need the operators to have different properties in different classes; the most notable examples being `Le` and `Lt` in `Algebra.Order`.

- My focus was getting real numbers work quickly *(see later)*, not to create a comprehensive algebraic hierarchy. Therefore, I usually formalised only those concepts that I needed to build the reals.

3.1.3 Dependencies

Based on my earlier experience, I definitely wanted to avoid using the Agda standard library, as its aim is proof simplicity rather than performance[16] and it is very difficult (though not impossible) to write agda2hs-compatible libraries with. Instead, I use the agda2hs standard

library, with concepts that get directly compiled to their Haskell equivalents. That, however, means that much of the theorems proved for naturals and integers need to be reimplemented. I have often used agda-stdlib for inspiration.

On the Haskell side, I have also added built-in alternatives of self-defined functions for the sake of efficiency; the most prominent ones being `Data.Bits.shift` and `integerLog2#`. Since we redefine operators like `(+)` and `(*)`, we can usually only use Prelude with a qualified import to avoid conflicts.

3.1.4 A "running-first" approach

This is a very important concept; on which I have already written earlier. Although the ultimate goal is a library which is both efficient and proved to be correct, my emphasis is on the former. That explains my extensive use of the `cheat` postulate and `FOREIGN` pragmas. The project can gain a greater momentum if we already have a working implementation where only the proofs need to be filled; that could then be done in collaboration with others. This approach definitely has some drawbacks: there is a great chance that there are some errors in the code, and if something turns out to be mathematically incorrect, that can result in a significant portion of the library having to be rethinked and restructured. Still, even if only some of the results are supported by Agda proofs, that already means an advantage compared to mathematical libraries found in most programming languages, where correctness is barely ever proved.

3.1.5 Generating Haskell code with agda2hs

Maybe the most distinctive feature of Acorn is compatibility with agda2hs[2]. When working with the compiler, we always need to consider whether the code we would like to compile makes sense in Haskell. This includes respecting naming conventions, avoiding the use of dependent types (unless the dependent part is erased), and other technical limitations.

That certainly brings some compromises with itself; one of the greatest of them is that we have to use an erased `_<_` proposition (unlike in the Bishop repository). This is a problem because `_<_` is defined using an existence quantifier, the content of which (called the *witness*) is needed later, for example when calculating the inverse of a real number. To left it unerased, we would need to provide a type synonym in the `Lt` type class for the witness, which Haskell does not support. Therefore, we esentially follow Krebbers' and Spitters' approach by defining the proposition as erased and calculating the witness only when necessary. For this, however, I had to bypass termination checking with the `cheat` postulate. (See `RealTheory.Reals` for details.)

There are also instances where the conversion to Haskell would work in theory, but compilation still fails (e.g. because of pattern matching on the `suc` constructor, which is not

supported by agda2hs yet). In other cases, the code compiles but would be ineffective (such as in the case of the power operation, where I use fast exponentiation instead of the incremental one which is easier to write proofs about). Consistently with the *"running-first approach"* explained above, I usually bypassed the compiler in these situations by writing the Haskell code by myself and placing it into a `FOREIGN` pragma. Of course, these "holes" will eventually need to be filled in.

Even if the compilation can be run successfully, special attention is needed to resolve clashes with Haskell's Prelude library, as we often use the same operators as those imported from there. A `FOREIGN` pragma containing `import qualified Prelude` (and usually other imports which do not get into the code, but would be needed) helps here.

3.1.6 Tests

Due to the *"running-first approach"*, 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. Tests are not included in the library yet; however, I already have some ideas on the topic.

The Agda side is relatively easy to write tests for: the builtin equality type gives the opportunity to use the typechecker itself to validate them. These might occasionally be more general and cover a possibly infinite number of test cases (e.g. by parametrising one or more aspects). This is not enough, though: the Haskell side needs testing as well. Since agda2hs is still an experimental project, there are rare cases in which the code gets compiled to legal Haskell code that, however, does not mean the same. An example is the precedence order of semigroup operators, which first did not get compiled into Haskell by itself, causing erroneous calculations. (This could be temporarily fixed by a `FOREIGN` pragma.)

For Haskell tests, I will probably use *QuickCheck*, which generates a large number of random test cases to validate given Boolean predicates about the output of a code[18]. This covers much more cases than what a human could write, and follows a logic similar to that of Agda by focusing on predicates rather than comparing outputs blindly.

3.1.7 Long-term goals

Since I have focused on writing programs rather than proofs, there are many uses of the `cheat` postulate in the code which need to be replaced with true correctness proofs. This is going to be a complicated task; maybe so complicated that I cannot do it myself. (I have already thought about exploring the capabilities of neural networks in proof automation. However, it would be a much more difficult project.)

As I have mentioned in the abstract, another direction is to formalise Bishop's constructive analysis, presenting existence proofs as programs. This way, on the long run, we could even be able to calculate derivatives and integrals.

3.2 Structure

The following table summarises the current structure of the Acorn library; largely in the order of dependencies (although these are sometimes circular).

Folder	Description
Tools	Some important tools frequently used later in the library that do not fit into the other categories.
Algebra	Algebraic structures and their operators, like `SemiRing` with `_+_` and `_*_` or `Field` with `recip`. Metric and prelength spaces' type classes can also be found here.
Operations	Type classes for operators not defined in algebraic structures and their properties; bundled into type classes.
Implementations	Concrete implementations of different number types (often imported from `Agda.Builtin` in the beginning) and instances for them.
RealTheory	Tools directly needed to implement the real numbers; such as the `AppRationals` class and the completion monad. The `Reals` module containing instances for the abstract reals can also be found here.
Function	Certain important irrational-valued functions. At the time being, this means the exponential function, sine and the square root function. `Trigonometric.agda` also contains the definition of π.
HaskellInstances	Instances of some type classes of Haskell's Prelude related to fractional numbers, using the agda2hs library. They are mainly needed for compatibility with Krebbers' test suite[10] and not often used in the library itself.

The purpose of an individual file is usually written on the first few lines of the file, as a comment.

3.3 Current state

The version of Acorn described in this article is commit `c06c7de`. Commit `57811ed` of agda2hs run on Ubuntu 20.04 (and based on Agda 2.6.4) successfully typechecks and compiles this version. The generated Haskell code is accepted by GHC 9.2.8 without modifications.

At this time, all the field operations work, and there are already some irrational-valued functions (that means *exp*, *sin*, *sqrt* and a limited-domain version of *arctg*). π is also defined.

3.4 Speed tests

Since executables are actually compiled from the Haskell code generated by agda2hs, I compared the library to the Haskell implementation written by Krebbers (which has been forked from O'Connor's original)[10]. The test program used was also the one made by him, modified to cooperate with Acorn's reals.

The tests Krebbers and Spitters have conducted were more than twelve years old; therefore, it was important to reproduce them on a modern computer.

The specifications of the computer were:

- Type: Asus UM431-DA
- CPU: AMD Ryzen 7 3700U
- System memory: 16 GiB
- GPU: Radeon RX Vega 10
- Storage: SK Hynix 256 GB NVMe-M.2 SSD

For the test, I set the bound under which the absolute values of the parameters had to be pushed to 2^{-75}, the level recommended by K&S for higher-precision calculations. From their original test set in `test.hs`[10], I only ran those in which all used functions were supported by Acorn.

The results:

	Expression	**Decimals**	**Acorn**	**K&S**
P01	$sin\,(sin\,(sin\,1))$	5.000	$0.70s$	$0.71s$
P02	$\sqrt{\pi}$	5.000	$0.52s$	$0.61s$
P03	$sin\,e$	5.000	$0.27s$	$0.45s$
P04	$exp\,(\pi \cdot \sqrt{163})$	5.000	$1.02s$	$0.53s$
P05	$exp\,(exp\,e)$	5.000	$0.86s$	$0.77s$
P07	$exp\,1000$	20.000	$0.22s$	$0.27s$
C02	$\sqrt{\frac{exp\,1}{\pi}}$	10.000	$2.77s$	$3.11s$
C03	$sin\,((exp\,1+1)^3)$	5.000	$0.25s$	$0.43s$
C04	$exp\,(\pi \cdot \sqrt{2011})$	5.000	$1.16s$	$0.58s$
C05	$exp\,(exp\,(\sqrt{exp\,1}))$	5.000	$1.42s$	$0.85s$
C07	π^{1000}	20.000	$2.11s$	$3.74s$

The numbers do not decide between the two solutions: Acorn is sometimes faster and sometimes slower. I have not discovered a pattern explaining in which situations Acorn performs better; however, it can clearly compete with an implementation written entirely in

Haskell. Considering that this fast code can be generated from Agda without any human interaction, I think I could demonstrate that agda2hs might bring verifiable arithmetic (and Agda in general) closer to real-life usage.

References

[1] Robbert Krebbers, Bas Spitters. *Type classes for efficient exact real arithmetic in Coq.* In: Logical Methods in Computer Science, Vol. 9(1:01), 2013.

[2] Jesper Cockx et al. *agda2hs – Compiling Agda code to readable Haskell.* On GitHub.

[3] *Constructive Mathematics.* In: Stanford Encyclopedia of Philosophy. Retrieved on 17th September 2023.

[4] Errett Bishop. *Foundations of Constructive Analysis.* McGraw-Hill, 1967.

[5] Michael Beeson. *Foundations of Constructive Analysis – a New Foreword.* Ishi Press International, New York, 2012.

[6] Mark Bickford. *Formalizing Constructive Analysis in Nuprl.* Cornell University, 2016. Retrieved on 17th September 2023.

[7] Herman Geuvers and Milad Niqui. *Constructive Reals in Coq: Axioms and Categoricity.* TYPES 2000, Durham, UK. Retrieved from ResearchGate on 30th September 2023.

[8] Luís Cruz-Filipe. *Constructive Real Analysis: a Type-Theoretical Formalization and Applications.* 2004. On Radboud Repository. Retrieved on 30th September 2023.

[9] Russell O'Connor. *A monadic, functional implementation of real numbers.* In: Mathematical Structurres in Computer Science, 2007. Retrieved on 30th September 2023.

[10] Robbert Krebbers, Jelle Herold. *fewdigits – forked from r6.ca/FewDigits/.* Available on GitHub.

[11] Bas Spitters et al. *CoRN – Coq Repository at Nijmegen.* On GitHub.

[12] András Kovács. *Efficient Evaluation for Cubical Type Theories.* 2nd Conference on Homotopy Type Therory, 2023. Retrieved on 18th September 2023.

[13] Zachary Murray. *Constructive Analysis in the Agda Proof Assistant.* Dalhousie University, 2022. (arXiv:2205.08354) Retrieved on 18th September 2023.

[14] Zachary Murray, Viktor Csimma. *bishop – A constructive analysis library written in Agda.* On GitHub.

[15] Viktor Csimma. *Acorn – An agda2hs-compatible implementation of Krebbers–Spitters reals, focusing on usability.* On GitHub.

[16] Nils Anders Danielsson et al. *The Agda standard library.* On GitHub.

[17] The GHC Team. *The Glasgow Haskell Compiler, 9.6.3. GHC User's Guide, Chapter 8. Profiling.* On haskell.org. Retrieved on 30th September 2023.

[18] Koen Claessen, John Hughes. *QuickCheck – Automatic Specification-Based Testing.* On cse.chalmers.se. Retrieved on 12th November 2023.

[19] Peter Alfeld. *Understanding Mathematics. Euler's number to 10,000 digits.* On www.math.utah.edu. The University of Utah. Retrieved on 31th October 2023.