Real numbers and intervals in Haskell
The `ireal` package

Björn von Sydow
bjorn.von.sydow@gmail.com

December 9, 2014

Abstract

This paper describes the package `ireal`, which provides potentially arbitrary precision real and interval arithmetic. The package is based on ideas that go back to Boehm and Cartwright and which are also used in a well-known Haskell implementation of computable reals by David Lester. We believe that our package is significantly faster than other Haskell packages for exact real arithmetic. Two essential features which contribute to this are the following:

- Our package merges real numbers and intervals in one type `IReal`, which enables us to control the precision of subexpressions, an ability which is crucial for efficiency in some situations.
- A memoizing technique is used to speed up recomputations of the same number at different precisions. Unfortunately, this is implemented using impure parts of Haskell.

In spite of these improvements, our package is still far beyond the best imperative C/C++ packages. But in some cases the difference is less than an order of magnitude, so we believe that our package can be used not only as a calculator, but also for small-scale problems in mathematics and its applications. A selection of such examples are provided with the distribution.

This paper is mostly an introduction to the use of the package and not a detailed explanation of the implementation.

1 Introduction

How should real numbers be represented in computers? This old question is of immense practical importance for scientific computing, but also of great theoretical interest. For most purposes, the obvious answer is *as floating-point numbers*. These have evolved via numerous poorly defined, vendor-specific formats to the present, carefully specified IEEE standard. All large-scale numerical computing relies on the efficiency of floating-point computations, but also has to cope with the presence of round-off errors.

At the opposite end of the spectrum we find *the computable reals*, which allow for error-free computation with potentially arbitrary precision. The concept goes back to Turing, who established their basic properties, including the disturbing facts that equality of computable real numbers is undecidable and that the basic arithmetic operators are not computable in the standard decimal representation. The restriction to a subset of real numbers is natural, since the real numbers form an uncountable set, and thus one cannot represent them all finitely as terms in a programming language. Many different implementation techniques for computable real numbers have been proposed over the years. One important distinguishing feature between different approaches is whether the precision required in a computation is specified in advance or not. In one approach, a real number is an infinite object which can, indefinitely, emit more and more information on request, like a lazy infinite list (even though the standard decimal representation cannot be used). In another, the desired precision has to be given beforehand; in effect, a real number is a function which takes the desired precision as input and returns an approximation with this precision. For the lazy functional programmer, the first approach may seem more elegant, but, unfortunately, all experience indicates that it is vastly inferior from an efficiency point of view[1][1]. The burden of having to maintain all information needed in order to be able to continue to arbitrary precision is just too high, compared to the competition who knows the limits of what will be requested. But also the less inefficient versions are, on contemporary hardware, orders of magnitude slower than floating-point arithmetic.

An intermediate approach to high precision computation over the reals is to use a library for *multi-precision floating point numbers*. These are based on similar ideas as the hardware-supported IEEE floating point numbers, but extended in software to mantissas of arbitrary, but fixed, length. An example is the C library MPFR [9], which pro-

[1] The present author regrets that he did not trust this evidence, but spent too much time on fruitless attempts to make a competitive implementation based on continued fractions.

vides such numbers with supporting arithmetic and all the mathematical functions in C, including correct rounding in various rounding modes. This is a very efficient alternative, with the disadvantage that it provides correctly rounded results only for atomic operations; the result is correctly rounded if the arguments are exact. For compound expressions, where intermediate results are rounded, a manual error analysis must be performed.

So, the applications for implementations of computable reals are limited, but significant; they include small-scale but high-precision tasks in mathematics and applications, reference versions of numerical algorithms and other computations over the reals, and educational applications. However, in order to explore these possibilities beyond the most trivial situations, the inefficiencies of real number computation must be kept to a minimum. Unfortunately, the real number packages available on Hackage so far do not meet the requirements. The best candidate seems to be Lester's `CReal` module, which is part of the `numbers` package. This is, however, too slow for most applications beyond its intended use as a calculator, i.e. to compute a high-precision value of a moderately complicated real expression.

This paper describes a Haskell implementation of the computable reals, based on ideas very similar to Lester's module, and, by transitivity, on Boehm's and Cartwright's earlier work [1]. However, it adds substantial engineering improvements and a few important new ideas, which in some cases lead to orders of magnitude in speedup. We should note immediately that Lester deliberately avoided some simple and obvious improvements in order to facilitate another of his goals: to formally prove the implementation correct in PVS [5, 6]. Our package is still painfully slow in many situations, but it does extend the range of problems that can be treated.

In order to discuss our more significant improvements we will first review another idea: *interval arithmetic*. Computing with floating-point numbers inevitably involves numerical approximations, and careful selection of robust algorithms is needed to keep errors under control, provide realistic error estimates and thus give reliable results. A further step towards trustworthy scientific computing is taken in the field of validated numerics [12], where interval representations of real numbers are used, together with other techniques such as automatic differentiation, to provide approximate solutions, which are guaranteed to enclose the exact solution to a problem. Interval arithmetic combines smoothly with floating-point numbers, since the IEEE standard provides several rounding modes, so intermediate values can be rounded "outwards", thus yielding conservative approximations. Highly sophisticated libraries and toolboxes with interval algorithms for a wide range of problems are available [3, 10]. Using these, the resources needed to compute verified solutions to many large-scale numeric tasks are within an order of magnitude of those used by standard numerical methods. Methods from validated numerics and associated computations can also play prominent roles in mathematical proofs, as witnessed by [11].

In common interval arithmetic systems, numbers and intervals are distinct types. Numbers are typically of type `Double` and there is an abstract type `Interval`, represented by a pair of numbers. A number can be coerced to a degenerate, or *thin*, interval in order to do arithmetic, but conceptually, numbers and intervals are different things.

In the context of computable reals, however, the distinction between numbers and intervals becomes blurred. At any point in a computation, a number is only partially known; in effect, it is known as an interval approximation. Conversely, a sufficiently narrow interval is indistinguishable from a number, when computed to low precision. Using the second approach above, both a real number and a real interval can be represented as a function from the type of precisions to the type of rational intervals. What distinguishes a number from an interval is the property that the lengths of its approximating intervals tend to zero as precision increases beyond bounds, something which is not observable during computation. Thus, in our implementation we merge intervals and numbers into a common type `IReal`. This brings not only conceptual simplification, but also gives us the crucial ability to control the precision of intermediate results. As we will see, this ability will be essential in many situations.

This paper is intended as an introduction to the use of the package; it is not a detailed description of the implementation. However, in order to make best use of the package, some properties of the implementation should be understood; these properties are explained here.

2 Simple `ghci` interaction

In this section, we illustrate the simplest interactive usage of our package. Two somewhat more substantial examples are discussed in section 5; further application examples are supplied with the distribution and are briefly discussed in section 6.

The type `IReal` is an instance of all the standard numeric classes:

```
> exp (sin 1.6) + pi * sqrt 3 ? 30
8.158521101133937220554906871457
```

Here we used the `?` operator, which takes the number of decimals as second argument. We can request higher precision:

```
> exp (sin 1.6) + pi * sqrt 3 ? 500
8.15852110113393722055490687145668933596385406452 66
62124842948785898792353174827305898550780930921 3353
40502583621293709553827789657723487809926469809 8547
43483469082702578767470055421091854799508395477 2274
```

```
698366651070974841651537156007874775950204996964627
895931005821835142268064237677239672492891538676097
448072070415221517896256740470951264756385302167668
779056108765292318511746872053953038800519590979341
091494013057457755671364586688209441689724254241783
9614723465240073507715670115255386679823477
```

Simple expressions like this are computed instantaneously; also 5000 decimals takes only a small fraction of a second on a typical laptop. But 50000 decimals is problematic for our implementation; it is doable, but takes several seconds. It turns out that different algorithms are suitable in different ranges of precision and we have chosen the ones that behave well in the low range, up to at most a few thousand decimals, less for substantial computations. Indeed, our main interest is in solving problems with moderate precision in end results correctly, which may require high precision in intermediate results.

Note that even though `IReal` is an instance of `Show`, using `?` is the default way to print `IReal` values. See section 4 below for more discussion on this.

Interval values are built using the `+-` operator:

```
> exp (sin (1.6 +- 0.05)) + pi * sqrt (3 +- 0.02) ? 20
8.1[| 3300320390 .. 7778778516 |]
```

The result is displayed in non-standard, but hopefully easily understandable notation. The resulting interval is conservative: it is guaranteed to include all values of the expression $e^{\sin x} + \pi\sqrt{y}$ for $x \in [1.55, 1.65]$ and $y \in [2.98, 3.02]$. One can note that the maximal value is attained not in a corner of this rectangle of the xy plane, but in $(\pi/2, 3.02)$. We also see that the result does not display 20 decimals; interval upper and lower bounds display at most 10 differing decimal positions.

As always in interval arithmetic, the result may be too pessimistic, because of the well-known dependency problem:

```
> let x = 1 +- 2 in x * x ? 10
[| -3 .. 9 |]
```

The value -3 would result if the left occurrence of x had the value -1 and the right occurrence the value 3 (or conversely). Multiple occurrences of an interval-valued variable in an expression will always lead to such overestimation. The techniques developed in applications of interval analysis to mitigate this annoying problem apply also to our package.

To illustrate the discussion at the end of the Introduction, let us compute a very narrow interval:

```
> exp (sin (0.5 +- 1e-40)) + pi * sqrt (3 +- 1e-50) ? 30
7.056544389144737295099235698513
```

The result is certainly an interval, but at the requested precision of 30 decimals, it looks like a number. By requesting more decimals, the true nature of the result becomes apparent:

```
> exp (sin (0.5 +- 1e-40)) + pi * sqrt (3 +- 1e-50) ? 40
7.0565443891447372950992356985127579794 70[| 1 .. 5 |]
```

Finally, we note that we can compute values far beyond the reach of double precision arithmetic:

```
> exp 10000 ?? 10
0.8806818226e4343
```

The `??` operator demands output in scientific notation, with the indicated number of significant digits.

3 Representing real numbers

As mentioned above, this paper does not focus on the internals of the package. However, in order to make efficient use of it, some aspects of the implementation must be understood.

We begin with a discussion of the representation of real numbers. We assume that the set of real numbers, the arithmetic operations and the elementary functions are already well understood, and our problem is to represent real numbers and their arithmetic using the data types and operations available to us in Haskell. Following the ideas from [1] and [6], we represent the number x by a function $\hat{x} : \mathbb{N} \to \mathbb{Z}$, with the requirement that for all $p \in \mathbb{N}$

$$|x \cdot 2^p - \hat{x}(p)| < 1. \tag{1}$$

This requirement may be more intuitive if we divide by 2^p to get

$$|x - \frac{\hat{x}(p)}{2^p}| < \frac{1}{2^p}$$

So, for a given precision p, we have to find a rational number with denominator 2^p, which approximates x to within 2^{-p}. The advantage in restricting the denominator to this special form is that the function values need indicate only the integer numerator, rather than the approximating rational number, and we can do most of the required computations in (unbounded) integer arithmetic.

Note that for a given real number x, (1) does not uniquely specify $\hat{x}$. For example, if $x = \sqrt{2}$, then $\hat{x}(1)$ can be 2 or 3, $\hat{x}(2)$ either 5 or 6 and $\hat{x}(3)$ either 11 or 12.

It follows easily from (1) that for all $p, r \in \mathbb{N}$

$$|\hat{x}(p+r) - \hat{x}(p) \cdot 2^r| \leq 2^r. \tag{2}$$

Conversely, for any function $\hat{x}$ satisfying (2), the sequence $\{\frac{\hat{x}(p)}{2^p}\}_{p=0}^{\infty}$ of rational numbers is a Cauchy sequence, and thus represents a real number.

In order to do arithmetic, assume that we know representations $\hat{x}$ and $\hat{y}$ of real numbers x and y. Then

$$\widehat{x+y}(p) = \textbf{scale}\ (\hat{x}(p+2) + \hat{y}(p+2))(-2),$$

where **scale** $m\,p = m \cdot 2^p$, correctly rounded. Thus, to approximate $x+y$ to precision p we have to approximate the operands to slightly higher precision and then scale down the sum. The proof of this is straightforward:

$$
\begin{aligned}
&|(\hat{x}(p+2)+\hat{y}(p+2))-(x+y)\cdot 2^{p+2}| \\
\leq\ &|\hat{x}(p+2)-x\cdot 2^{p+2}|+|\hat{y}(p+2)-y\cdot 2^{p+2}| \\
<\ &2.
\end{aligned}
$$

Division by 4 gives $|(\hat{x}(p+2)+\hat{y}(p+2))/4-(x+y)\cdot 2^p| < 1/2$. Rounding to **scale** $(\hat{x}(p+2)+\hat{y}(p+2))(-2)$ adds another $1/2$ to the error estimate.

This reasoning suggests the following code fragment (which will be slightly modified later):

```
data IReal = IR (Int -> Integer)

instance Num IReal where
  IR x + IR y = IR (\p -> scale (x(p+2) + y(p+2)) (-2))
  IR x * IR y = IR f
    where sx = lg2 (abs (x 0)) + 3
          sy = lg2 (abs (y 0)) + 3
          f p = scale (x (p+sy) * y (p+sx)) (-(p+sx+sy))

  fromInteger n = IR (shift n)
  ...
```

We will not explain the code for multiplication in detail, but two things should be noted:

- Again we approximate x and y to higher precision, perform the operation and then scale down the result.
- Here, the required extra precision depends on the size of *the other* factor, so we first make a crude approximation of the operands (to precision zero), in order to estimate the precision required in the main step.

The proof that the multiplication algorithm is correct is omitted; see [1, 5].

The final code line above shows the `fromInteger` function, which coerces an integer into a real number. To represent integer n at precision p, we simply need to multiply n by 2^p, or, equivalently, shift n to the left p steps.

In the above, we implicitly made coercions as needed between integers, rational and real numbers, as is usually done in mathematics. Another approach could be to start out from integers and rational numbers only and *define* a real number to be a function $\hat{x} : \mathbb{N} \to \mathbb{Z}$ satisfying (2). In this paper we will, however, not develop this point of view further.

4 Limitations

Our implementation has a number of unavoidable limitations, having to do with fundamental properties of the computable reals.

Display format. `IReal` values shown as output are *not* correctly rounded. The error can be up to one unit in the least significant digit, not 0.5 as correct rounding would demand. This is for computability reasons: the decimal notation with correct rounding does not have enough redundancy.

As a simple illustration of the problems with correct rounding, consider computing the value of an expression which has value exactly 0.25, and displaying the result with one decimal. The computation proceeds by computing increasingly narrow interval approximations which all include the exact value 0.25, but none of these allows to decide if the single correctly rounded decimal should be 2 or 3, leading to non-termination. Thus, more redundancy in output is necessary. Our chosen policy allows that, as soon as we know that the value is greater than 0.2 and less than 0.3, we can output either of these.

This redundancy is also the reason for not recommending to use the `Show` instance; we cannot guarantee to provide equal string representations for equal values. This is less of a problem for `?`, which is a command, not an expression:

```
(?) :: IReal -> Int -> IO ()
```

Our reason for providing the instance at all is that some QuickCheck combinators require it to be present.

Partial functions. Functions, which have a subset of the reals as their natural domain, will often lead to non-termination when applied to an argument at the border of their domain. An example is $1/\sin\pi$, where the denominator will be approximated to ever higher precisions, in a fruitless attempt to bound it away from zero. Similar remarks apply to e.g. $\sqrt{0}$ and $\log 0$.

Comparison operators. Equality on the computable reals is undecidable; `IReal` is an instance of `Eq`, but the Boolean expression e_1 `==` e_2 will lead to non-termination if the values of the two expressions are actually equal as real numbers. On the other hand, non-equal numbers will be correctly identified as such. Similar remarks hold for other comparisons.

As a weak substitute we provide total, but approximate tests in the form of the three operators

```
(=?=), (<!), (>!) :: IReal -> IReal -> Int -> Bool
```

The types may be surprising, but the intuition here is that e_1 `=?=` e_2 is a function which, when applied to a precision q, returns a Boolean value, with the semantics that if this value is `True`, then $|e_1 - e_2| < 2^{-q}$; if it is `False`, they are definitely unequal. Similarly, if $(e_1$`<!`$e_2)\,q$ is `True`, then x is definitely less than y; if it is `False`, then $e_1 > e_2 - 2^{-q}$. To facilitate the common practice of working with decimal precisions, we provide a simple conversion function

```
atDecimals :: (Int -> a) -> Int -> a,
```

with the property that if e_1 `=?=` e_2 `` `atDecimals` `` d is `True`, then $|e_1 - e_2| < 10^{-d}$.

5 Two examples

In this section we consider two a bit more demanding computations, which also illustrate that computing in type `IReal` requires knowledge of some of its idiosyncrasies. In particular, evaluating a compound expression to a given precision will typically mean that the subexpressions will have to be evaluated to higher precisions. In some situations, for heavily nested expressions, this may lead to excessive precision requirements and very slow computations.

5.1 Approximating $\zeta(5)$

This example shows that very unbalanced expression trees can be harmful when computing with `IReal`s.

We consider the sum

$$\zeta(5) = \sum_{n=1}^{\infty} \frac{1}{n^5},$$

i.e. the value of the Riemann zeta function at 5. Let us approximate it by replacing ∞ by 10^k for suitable k:

```
> sum [recip (fromInteger (n^5)) | n <- [1..10^k]] ? 30
```

The following table shows the result for small k, and also the computation time:

k	Value	Time (s)
2	1.036927752692953288826366860916	0.01
3	1.036927755143120425914699111457	0.02
4	1.036927755143369901336365069790	2.97
5	?????	> 100

Not only is the time exponential in k, as should be expected, but it grows so quickly that it is not feasible to compute the sum of more than a few tens of thousands of terms.

The reason for this disappointing performance is in the definition of addition and its interaction with the prelude function `sum`, which leftfolds the summation, thus computing $\sum_{i=1}^{n} x_i$ as

$$(\ldots((x_1 + x_2) + x_3) + \ldots) + x_n.$$

Thus, when we request the value of this sum at precision p we need to compute x_n at precision $p+2$, x_{n-1} at precision $p+4$, $\ldots$, x_1 at precision $p+2n$. So, for $n = 10^5$, we need to add (many) integers with up to 200 000 binary digits.

To fix the problem, we simply change to a balanced sum operator:

```
bsum []  = 0
bsum [x] = x
bsum xs  = bsum (f xs)
  where f (x:y:xs) = x+y : f xs
        f xs = xs
```

The maximal precision necessary for the operands now grows logarithmically, rather than linearly, with the number of terms, and the timing behaviour is as should be expected:

k	Value	Time (s)
3	1.036927755143120425914699111457	0.01
4	1.036927755143369901336365069791	0.11
5	1.036927755143369926328865536457	1.06
6	1.036927755143369926331365236458	11.2

The function `bsum` is provided in the package, and should be preferred in most situations where sums of many terms are computed.

Finally, we note that the method used here is extremely naive as a means of computing $\zeta(5)$.The last approximation above, using a million terms, has only 24 correct decimals. This can be improved a bit by estimating the remainder, but much more sophisticated methods exist. Among the examples provided with the distribution is a file demonstrating how to compute $\zeta(5)$ to a thousand decimals in little time.

5.2 The logistic map

Sometimes the general estimates of how much extra precision is needed in subexpressions are far too pessimistic. In such cases one may use the the predefined function

```
prec :: Int -> IReal -> IReal
```

to bound the precision and improve efficiency. The intuition here is that `prec` $q\,x$ is "x evaluated at decimal precision q", i.e. a value that differs from x by at most 10^{-q} – in fact, `prec` $q\,x$ is a *real interval* of width 10^{-q} containing x. Using `prec` implies a trade-off between precision and speed: `prec` $q\,x$ immediately computes the approximation at precision q and all subsequent approximations, also to higher precision, are then just scaling (since they will not increase the actual precision) and thus very fast. The benefits of this become apparent in situations where x will be computed many times, at various (high) precisions, but the actual precision needed can be bounded to q.

The logistic map provides a simple, well-known example of chaotic behaviour in non-linear dynamic systems. The map is defined by

$$f_c(x) = c \cdot x \cdot (1 - x),$$

where $0 < c \leq 4$ is a parameter. For the indicated parameter values, f_c maps the unit interval $0 \leq x \leq 1$ into itself, and we can iterate the map starting from some initial value x_0. To compute the resulting orbit, we define

```
orbit c x0 = iterate f x0
   where f x = c * x * (1 - x)
```

This is not the place to tell the fascinating story of how orbits depend on c; we have to be content with noting that for most c above a threshold value $c_0 \approx 3.5699$, the orbit becomes chaotic, wandering aimlessly along its attractor. What is interesting for us is that in this range of parameter values, orbits cannot be computed in type `Double`, as shown by this `ghci` interaction:

```
> orbit 3.75 0.3 !! 100
0.2700006882287392
> orbit 3.75 0.3 !! 100 ? 30
0.779073576335491987762386034597
```

First the value of the expression is computed using the default type `Double`, and then in type `IReal`. As we know, the latter result has an error less than 10^{-30}. Thus, we see that the computed `Double` value is completely useless. This is typical for chaotic behaviour: extremely small initial disturbances (such as rounding errors) grow exponentially and come to dominate the computed value after a few dozen steps. In type `IReal`, on the other hand, we can in principle trace the orbit arbitrarily long, but in practice we face problems with exponentially growing computation times, so we can only compute a few hundred steps of the orbit. The main reason is the fact, noted above, that in a multiplication both operands are computed twice: first the operands are evaluated to low precision (with an error of 1) in order to determine which precision is needed in the main approximation. This causes exponential behaviour in deeply nested expressions as here.

The remedy is to note that we can actually bound the precision required in intermediate results more exactly than the worst-case estimates inherent in the multiplication algorithm.

So, we consider the following question: if x_0 is disturbed to x_0' with $|x_0' - x_0| < 10^{-q}$, how is x_n affected? We have that $x_n = f_c^{(n)}(x_0)$, where the (n) superscript denotes function application iterated n times. Hence, using the mean value theorem, we get

$$|x_n' - x_n| \leq \sup_{x \in [x_0', x_0]} \left| \frac{d}{dx} f_c^{(n)}(x) \right| |x_0' - x_0| < c^n 10^{-q}.$$

From this we can deduce that to ensure $|x_n' - x_n| < 10^{-d}$, it is enough that $q \geq n \log_{10} c + d$. In fact, this estimate holds at every iteration, so that in order to compute x_n with an error of at most 10^{-d}, we can replace f_c by $\texttt{prec}\ q \circ f_c$ in the definition of the orbit.

Using this idea we can follow the orbit to thousands of elements in a fraction of a second. Let us compute x_{1000} with 100 decimals:

```
> let f c x = c * x * (1-x)
> let q = ceiling (1000 * logBase 10 3.75 + 100)
  in iterate (prec q . f 3.75) 0.3 !! 1000 ? 100
0.6735323088616032058281479405090590454180011125314
385751711164765300056206996985595402995256137440410
```

The methodology here, to limit the precision of intermediate results, might seem dangerous; we made a manual and error-prone analysis to determine the required q. However, this is not as harmful as it seems – if we erroneously provide a too small first argument to `prec`, this will not make the computed result invalid, but the result will be an interval which simply is too wide for the requested precision:

```
> let q = ceiling (1000 * logBase 10 3.75 + 100) - 2
  in iterate (prec q . f 3.75) 0.3 !! 1000 ? 100
0.6735323088616032058281479405090590454180011125314
3857517111647653000562069969855954029952561374404
[| 06 .. 13 |]
```

Thus, the computation itself provides a conservative error estimate; we could also have determined a suitable q by trial and error and still got a rigorous result.

The moral here is more general than this example: when using the prelude function `iterate` to compute a list of `IReal` values, it is often useful (or essential) to combine this with bounding the precision using `prec`.

6 Further applications

In directory `applications` of the distribution we give a number of small example applications to indicate how the package can be used. These are only briefly discussed here. For each file we mention below a typical problem that can be solved in less than a second on a 2.5 GHz MacBook Pro. These examples are more or less arbitrarily chosen, just to give an impression of what is possible.

- Automatic differentiation (`FAD.hs`). This is not an application of real arithmetic, but several of the other applications make use of automatic differentiation. Unlike the case of real arithmetic, several useful packages implementing this technique are available on Hackage. However, they all seem to define the derivative of a product $f \cdot g$ by directly using the elementary fact

  $$\frac{d}{dx}(f \cdot g) = \frac{df}{dx} \cdot g + f \cdot \frac{dg}{dx}.$$

  This is elegant, but becomes exponential in the order of differentiation for higher derivatives, and hence unsuitable for much of validated numerics, which often uses moderately high order derivatives. We therefore provide a module `FAD` that offers simple forward automatic differentiation with a more involved but more efficient definition of differentiation of products.

- Newton's method for solving equations (`Newton.hs`). Here we implement the interval Newton's method, which in many respects has much nicer behaviour than the classical version (see [12]). As an example, we consider the equation $3\cos(30x^2) = x^3$, and find the 19 roots in $[-1, 1]$, all to 1000 decimals, in a second.

- Fast Fourier Transform (`FFT.hs`). This modules offers a simple-minded implementation of the FFT for lists of complex numbers. Based on this, we also provide cosine and sine transforms. These functions illustrate the technique to give an extra precision parameter in order to improve efficiency. The FFT of a 512-element list can be computed to 100 decimals in less than a second.

- Clenshaw-Curtis quadrature (`ClenshawCurtis.hs`). We use Waldvogel's elegant method to compute weights [13], based on the discrete cosine transform, and show as an example how to quickly compute

$$\mathrm{erf}(1) = \frac{1}{\sqrt{\pi}} \int_{-1}^{1} \mathrm{e}^{-x^2} dx$$

  to 100 decimals, using 129 points. The argument that 129 points are enough can be based either on well-known strict error estimates for Clenshaw-Curtis quadrature with analytic integrands, or, less rigorously, on computation of a sequence of approximations with more and more points.

- A much better way to implement the error function is given in `Erf.hs`, using a MacLaurin expansion and some of the internal auxiliary functions of the library. This exemplifies how the library can be extended to special functions. Now erf(1) can be computed to 20000 decimals in a second.

- Linear algebra (`LinAlg.hs`). Common linear algebra algorithms, typically based on mutating vectors and matrices, are difficult to express in functional languages and we do not address that problem. Here we offer a simple solver for systems of linear equations, based on LQ-factorization, where the lower triangular L is represented as a list of lists and the orthogonal Q as a list of normal vectors of Householder reflectors. Only small systems can be handled, but on the other hand extremely ill-conditioned problems pose no problems. A 25×25 system with the notoriously ill-conditioned Hilbert matrix as coefficient matrix can be solved to 100 decimals in a second.

- Solving initial value problems for ordinary differential equations (`ODE.hs` and `Taylor.hs`). The solver uses a variable order Taylor method, combined with Picard-Lindelöf's method, to guarantee a valid enclosure, following the ideas in [12].

- Computing $\zeta(5)$ efficiently (`Zeta.hs`). In this file we use Koecher's formula to compute $\zeta(5)$ to much higher precision than in section 5; a second is enough for 1000 decimals.

- An experiment in parallel execution (`Para.hs`). A summation is split into several subsums, which are sparked individually. This has only been tested on a two-core machine where speedup is almost perfect.

7 Representing intervals

We now turn to representing intervals. The obvious intuition is that a real interval is a pair of real numbers, the lower and upper bound. To merge numbers and intervals into one type, we define a real interval I to be a function $f_I : \mathbb{N} \to \mathbb{Z} \times \mathbb{Z}$, with the property that if $(l,u) = f_I(p)$, then

$$l < x \cdot 2^p < u.$$

for all $x \in I$. This requirement is not enough, however; it does not ensure that ever higher precisions converge to an interval. We must also require that the end points are real numbers, i.e. that $\lambda p.\mathtt{fst}(f_I(p))$ and $\lambda p.\mathtt{snd}(f_I(p))$ satisfy (2).

Also numbers are represented by functions of this type, with the further property that the function values (l,u) are always thin intervals, i.e. fulfil $u = l+2$. Thus we are led to modify the code fragment from section 3:

```
data IntegerInterval = I (Integer, Integer)

midI (I (l,u)) = shift (l+u) (-1)
radI (I (l,u)) = shift (u-l+1) (-1)
lowerI (I (l,_)) = l
upperI (I (_,u)) = u

isThin x = radI x == 1

instance Num IntegerInterval where
   I (l1,u1) + I (l2,u2) = I (l1+l2, u1+u2)
   fromInteger n = I (n-1,n+1)
   ...

data IReal = IR (Int -> IntegerInterval)

instance Num IReal where
   IR x + IR y = IR (\p -> scale (x(p+2) + y(p+2)) (-2))
   ...
```

We omit the straightforward definition of scaling for integer intervals; given that, the definition of addition is as before. Multiplication becomes quite a bit more complicated, since we need to keep track of signs in order to ensure that the two endpoints are ordered.

8 Elementary functions

Following well-established ideas for high-precision computation with real numbers, most of the functions in `Floating` are implemented using truncated MacLaurin expansions, combined with extensive range reductions, so that the expansions are only used for "small" arguments. Let us consider the exponential function as an example. For any x and n we have

$$e^x = e^{x - n\log 2} \cdot 2^n$$

Thus, to compute e^x, we first choose integer n such that $x - n\log 2$ is small, by computing $x/\log 2$ to precision 0. Then we use the MacLaurin expansion to compute $e^{x-n\log 2}$ and scale the result n steps.

9 Correctness

We have proved informally the correctness of the arithmetic operations and the general MacLaurin expansion function used for the elementary functions. We do not give the details here; the proofs are similar to those in Lester and Boehm, with some extra details to handle intervals. In particular the latter are a bit messy, with accompanying risk of mistakes in the proofs.

Instead, we will discuss the property-based testing we have done of the implementation, using QuickCheck. The real numbers and the elementary functions form a rich theory, with many properties that can be tested. A few examples are

$$\begin{aligned} \exp(\log x) = x, &\text{ for all } x > 0. \\ \sin^2 x + \cos^2 x = 1, &\text{ for all } x, \\ \log|x| + \log|y| = \log|xy|, &\text{ for all non-zero } x \text{ and } y. \end{aligned}$$

In order to test properties like these, several steps are necessary:

- As discussed above, equality between computable reals is not decidable, so we cannot test the above properties. We have to be content with testing that the two sides differ at most by a very small amount ε; in our test suite we have chosen $\varepsilon = 10^{-200}$.

- We have to define *generators* of real numbers, i.e. functions that generate pseudorandom numbers, uniformly distributed over a given range.

- We have to express the tests using the QuickCheck combinators.

We have implemented these steps and also added a number of other tests; it is easy to come up with a large number of tests, such as the following:

- Not only exp and log are inverses of each other, but also sin and asin, sinh and asinh, sqrt and (ˆ2), and others, leading to tests similar to the first example above.

- Many trigonometric and other identities, well-known from calculus, are tested.

- We test that `IReal` and `Double` versions of functions give consistent results, e.g. differ in at most the two least significant bits of the `Double` mantissa.

- We also test the tools themselves, such as that both generated numbers and randomly generated expressions with real number variables are proper real numbers (i.e. satisfy the Cauchy criterion), that functions applied to interval arguments give proper intervals as results (i.e. both endpoints are proper real numbers), etc.

The fact that our package passes the testsuite strongly increases our confidence in the correctness of the implementation. The methodology of combining informal proofs (which prove correctness but may contain bugs) and extensive machine-tested properties (which cover only some aspects of correctness but are less susceptible to mistakes) is quite powerful; it certainly falls short of completely formal, machine-checked proofs, but it also involves far less effort.

Our current testsuite is a bit ad hoc; it would be an interesting problem to design a systematic suite for real arithmetic, including the elementary functions.

10 Memoization

We have to discuss one more internal detail of the implementation; it is a dark corner, which makes use of unsafe, impure parts of Haskell.

A real number is a function, and in an involved computation we may compute approximations of it many times, to different precisions. But the function values for different precision arguments are not independent. In fact, if $\hat{x}(p) = n$ and $q < p$, then **scale** $n\,(q-p)$ is a valid result for $\hat{x}(q)$. Thus, if we cache the result of the highest precision approximation computed so far, approximations to lower precision can be computed very quickly. It turns out that doing so is crucial for efficiency and in our implementation a number is such a memoized function. But our memoization mechanism is based on storing the highest approximation in an `MVar`, using techniques borrowed from Augustsson's `uglymemo`, culminating in the spectacularly ugly definition

```
unsafeMemo f = unsafePerformIO .
                 unsafePerformIO (memoIO f)
```

We have spent considerable time searching for a pure, elegant and efficient solution to this memoization problem, without success.

11 Other implementations

We have already mentioned the work of Boehm and Cartwright and that of Lester. A Java version of the former is available as a calculator (see [2]). Lester's Haskell module is available on Hackage as part of Augustsson's `numbers` package.

Several packages for multi-precision real arithmetic, based on GNU MPFR are also available on Hackage, including `hmpfr`, `HERA` and the various `AERN` packages. Unfortunately, these packages require a specially compiled `ghc` to avoid conflicts caused by the fact that both `ghc` and MPFR rely on the GNU library GMP for bignum arithmetic.

Other Haskell implementations of computable reals, not available on Hackage, are Plume's Calculator for Exact Real Computations and Potts' and Edalat's IC-reals, based on continued fractions. Our package seems easier to install and significantly less inefficient than both of these.

Compared to the best C/C++ packages, such as Müller's iRRAM [4], our package is much slower. We have only made comparisons by trying the problems from competitions such as [7] and [8], where our package for many problems are only a few times slower, while there are other problems that we just can't handle.

References

[1] Hans Boehm and Robert Cartwright. Exact real arithmetic: formulating real numbers as functions, in Research topics in functional programming, Addison-Wesley 1990, pp. 43–64.

[2] Hans-J. Boehm. The constructive reals as a Java Library, J. Log. Algebr. Program (64) 2004, pp. 3–11.

The calculator is available at
`http://www.hboehm.info/crcalc/CRCalc.html`.

[3] boost Interval Analysis Library.
URL: `http://www.boost.org`.

[4] iRRAM - Exact Arithmetic in C++.
URL: `http://irram.uni-trier.de`.

[5] David Lester and Paul Gowland. Using PVS to validate the algorithms of an exact arithmetic, Theoretical Computer Science (291)(2) 2003, pp. 203–218.

[6] David Lester. The world's shortest correct exact real arithmetic program?, Information and Computation (216) 2012, pp. 39-46.

[7] Milad Niqui and Freek Wiedijk. 'Many digits' Friendly Competition. Tech report, Radbout University Nijmegen, 2005.
URL: `http://www.cs.ru.nl/ freek/pubs/comp_rep.pdf`.

[8] More Digits Friendly Competition, at RNC 7, LORIA, Nancy, France, July 2006.
URL: `http://rnc7.loria.fr/competition.html`

[9] The GNU MPFR Library.
URL: `http://www.mpfr.org`.

[10] S.M. Rump. INTLAB - INTerval LABoratory. In Tibor Csendes, editor, Developments in Reliable Computing, pp. 77–104. Kluwer Academic Publishers, Dordrecht, 1999.
URL: `http://www.ti3.tuhh.de/rump/intlab`.

[11] Warwick Tucker. A Rigorous ODE Solver and Smale's 14th Problem, Foundations of Computational Mathematics (2) (1) 2002, pp. 53–117.

[12] Warwick Tucker. Validated Numerics: A Short Introduction to Rigorous Computations. Princeton University Press 2011.

[13] Jörg Waldvogel. Fast Construction of the Fejér and Clenshaw-Curtis Quadrature Rules. BIT (43)(1), 2003, pp. 1-18.