Final Report for Deliverable Nr. 4.1

Improved Implementation of Real Numbers

Responsible: Bas Spitters, spitters@cs.ru.nl
Site: Radboud Universiteit Nijmegen, The Netherlands

Deliverable Date: February 2011

What follows below is a condensed version of the article 'Computer certified efficient exact reals in Coq', by Bas Spitters and Robbert Krebbers to appear in Proceedings of CICM11, LNAI. [1]

There is a big gap between numerical algorithms in research papers, which typically use concepts like Hilbert spaces and fixed point theorems from functional analysis, and their actual implementation which uses floating point numbers and matrix operations. This gap makes it difficult to trust the code, which is crucial when these algorithms are used in safety critical software (e.g. control of a power plant, medical software, airplane navigation systems,...). Similarly, it makes it difficult to trust the proof of theorems (e.g. Kepler conjecture, existence of chaos, existence of the Lorentz attractor) that rely on the execution of this code for numerical approximation. Finally, from a purely software engineering point of view, the situation is undesirable, because the gap between the (abstract mathematical) numerical algorithms and the (concrete floating point) implemented program makes the code difficult to maintain.

The challenge to close this gap has already been posed by Bishop in his fundamental work on constructive analysis [1]. Bishop proposed to use constructive analysis to bridge this gap. This gap can be narrowed by using

- exact real numbers or intervals instead of floating point numbers;
- functional programming instead of imperative programming;
- dependent type theory which allows us to compute with complete metric spaces;
- a proof assistant which allows us to verify the correctness proofs;
- constructive mathematics to tightly connect mathematics with computations and to avoid computationally impossible case distinctions.

Separately, all these tools have proved itself. By going to the limits of this proven technology we should be able to come within a small constant factor of floating point computations. In this way one would obtain a tool suitable for research and education in numerical analysis that allows one to compute abstractly at the level of functional analysis, e.g. to compute fixed

[1] http://www.cs.ru.nl/~spitters/calculemus.pdf

points of operators on Hilbert spaces. Like the development of FORTRAN and MatLab this will require a huge amount of work. In task 4.1 we focus on the performance of real number computation in the Coq proof assistant.

As said, floating point operations are fast, but require continuous effort on the part of the user in order to ensure that the results are correct. This burden can be shifted away from the user by providing a library of *exact* analysis in which the computer handles the error estimates. O'Connor implemented exact real numbers and trigonometric functions in the Coq proof assistant. His implementation has a reasonable performance. We dramatically improve the performance of his implementation in two ways: we use dyadic rationals built from the machine integers and we optimize computation of power series by using approximate division. Moreover, we use type classes for clean mathematical interfaces. This appears to be the first time that type classes are used in heavy computation. We obtain over a 100 times speed up of the basic operations and indications for improving the Coq system.

Introduction Real numbers, being infinite objects, cannot be represented exactly in a computer. Hence, in constructive analysis [1] one uses functions when fed a desired precision approximate a real numbers by a rational, or a dyadic number, to within that precision. The real numbers are the completion of the rationals. This completion construction can be organized in a monad, a familiar construct from functional programming. The completion monad provides an efficient combination of proving and computing [2]. In this way, O'Connor [3] implements exact real numbers and the transcendental functions on them in Coq.

A number of improvements in this implementation are suggested in [4]. First, we can use Coq's new machine integers. Second, we can use dyadic rationals (that are numbers of the shape $n * 2^e$ for $n, e \in \mathbb{Z}$, also known as infinitary floats). Third, the implementation of the trigonometric functions can be improved by using approximate division in the power series. Here we carry out all three optimizations. Unfortunately, changing O'Connor's implementation to use the new machine integers was far for from trivial, as he used a particular concrete representation of the rationals. To avoid this in the future, we provide an abstract specification of the dense set as *approximate rationals*. We use the theory of metric spaces and the algebraic hierarchy based on type classes [5] as described in task 4.2.

Virtual machine and machine integers. Coq includes a virtual machine, vm_compute, based on Ocaml's virtual machine [6] to allow efficient evaluation. Both the abstract machine and its compilation scheme have been proved correct, in Coq, with respect to the weak reduction semantics. However, we still need to extend our trusted core to a bigger kernel, as the *implementation* has not been formally verified. Machine integers were also added to the Coq system. The usual evaluation inside Coq (compute) uses a special inductive type for cyclic integers, but the virtual machine uses Ocaml's machine integers. This allows for a big speed-up, for which we pay by having to trust (the virtual machine and) that Ocamltreats these integers correctly. The time difference between computation with Coq's int and Ocaml's Big_int is about a factor of 20 on primality tests [7]. We find a similar ratio in our own experiments; see the benchmarks below.

Order theory To abstract from $\mathbb{N}$, $\mathbb{Z}$, $\mathbb{Q}$ and $\mathbb{R}$ and their various implementations, we provide a basic library for order theory and its interaction with algebraic structures based on type classes [5]. For example,

```
Class SemiRingOrder `{Equiv A} `{RingPlus A} `{RingMult A} `{RingZero A} (o : Order A) := {
  srorder_partialorder :> PartialOrder (≤) ;
  srorder_plus : `(x ≤ y ↔ ∃ z, 0 ≤ z ∧ y = x + z) ;
  srorder_mult : `(0 ≤ x → ∀ y, 0 ≤ y → 0 ≤ x ∗ y) }.
```

This allows us to refer by canonical names to lemmas as those shown below for $\mathbb{N}$, $\mathbb{Z}$, $\mathbb{Q}$ and the dyadics.

```
Lemma plus_compat x1 y1 x2 y2 : x1 ≤ y1 → x2 ≤ y2 → x1 + x2 ≤ y1 + y2 .
Lemma sprecedes_1_2 : 1 < 2.
```

For instances of $\mathbb{N}$, $\mathbb{Z}$, $\mathbb{Q}$ it is easy to define an order satisfying these interfaces:

```
Instance nat_precedes `{Naturals N} : Order N | 10 := λ x y, ∃ z, y = x + z.
```

However, often we encounter an a priori different order on a structure, most likely an order defined in Coq's standard library (like Nle on N). Therefore we prove that an arbitrary order satisfying these interfaces while also being a TotalOrder uniquely specifies the order on $\mathbb{N}$, $\mathbb{Z}$ and $\mathbb{Q}$. For example:

```
Context `{Naturals N} `{Naturals N2} {f : N → N2} `{!SemiRing_Morphism f}
  {o1 : Order N} `{!SemiRingOrder o1} `{!TotalOrder o1}
  {o2 : Order N2} `{!SemiRingOrder o2} `{!TotalOrder o2}.
Global Instance: OrderEmbedding f.
```

Basic operations The operation nat_pow is most commonly defined as repeated multiplication and the operation shiftl is defined as repeated multiplication by 2. However, if we implement these operations that way, they become too slow for the purpose of our work: efficient computation with real numbers. Instead we specify the desired behavior of these operations. This approach allows for different implementations for different number representations and avoids definitions and proofs becoming implementation dependent.

We introduce interfaces that specify the behavior of the operations abs, shiftl, shiftr, nat_pow, int_pow and Euclidean division. Again there are various ways of specifying these interfaces: with Σ-types, bundled or unbundled[2]. In general, Σ-types are convenient for functions whose specification is easy, for example:

```
Class Abs A `{Equiv A} `{Order A} `{RingZero A} `{GroupInv A}
   := abs_sig: ∀ (x : A), { y : A | (0 ≤ x → y = x) ∧ (x ≤ 0 → y = −x)}.
Definition abs `{Abs A} := λ x : A, ` (abs_sig x).
```

However, for more complex operations such an interface is different from the usual mathematical specification because we cannot quantify over all possible input values. Now there are two alternative ways: a bundled or an unbundled interface. Since these interfaces are not used for hierarchies the disadvantages of unbundled interface do not apply. Let us first describe the bundled approach. The function shiftl n multiplies a number by 2^n:

```
Class ShiftL A B `{Equiv A} `{Equiv B} `{RingOne A}
   `{RingPlus A} `{RingMult A} `{RingZero B} `{RingOne B} `{RingPlus B} := {
```

[2] These notions are explained in task 4.2

```
  shiftl : A → B → A ;
  shiftl_proper : Proper ((=) ⟹ (=) ⟹ (=)) shiftl ;
  shiftl_0 :> RightIdentity shiftl 0 ;
  shiftl_S : ∀ x n, shiftl x (1 + n) = 2 ∗ shiftl x n }.
Infix "≪" := shiftl (at level 33, left associativity).
```

Although this interface seems reasonable, it does not work well in Coq. The simpl tactic which is used to simplify a goal will unfold occurrences of shiftl to their underlying definition (for example in case of BigN, the expression x ≪ n becomes BigN.shiftl x n). This is rather inconvenient because Coq will then be unable to use lemmas using ≪ for rewriting. This problem is caused because shiftl is a projection of a record, which is in fact a δ-redex that will be unfolded by simpl. Currently there seems to be no way to adjust the behavior of simpl to remove this inconvenience. A similar problem was already observed in Ssreflect [8].

Instead we use an unbundled interface, which has a lot in common with our interfaces for algebraic structures. Now shiftl no longer contains a δ-redex.

```
Class ShiftL A B := shiftl: A → B → A.
Infix "≪" := shiftl (at level 33, left associativity).
Class ShiftLSpec A B (sl : ShiftL A B) `{Equiv A} `{Equiv B} `{RingOne A}
   `{RingPlus A} `{RingMult A} `{RingZero B} `{RingOne B} `{RingPlus B} := {
  shiftl_proper : Proper ((=) ⟹ (=) ⟹ (=)) (≪) ;
  shiftl_0 :> RightIdentity (≪) 0 ;
  shiftl_S : ∀ x n, x ≪ (1 + n) = 2 ∗ x ≪ n }.
```

We do not specify shiftl as shiftl x n = x ∗ 2 ^ n since on the dyadics we cannot take a negative power while we can shift by a negative integer.

Decision procedures The Decision type class collects types with a decidable equality [5]. We tweaked the implementation of this type class in order to let it be evaluated lazily.

Approximate rationals In order to make our implementation of the reals independent of the underlying dense set, we provide an abstract specification of such a set. The interface of this specification is called *approximate rationals* and is inspired by the notion of *approximate fields* which is used in the Haskell implementation of the exact reals by Bauer and Kavler [9]. We provide an implementation of this interface by dyadics based on Coq's machine integers.

Our interface describes an ordered ring containing Z that is dense in Q. Here Z are the binary integers from Coq's standard library, and Q are the rationals based on these binary integers. We do not parametrize by arbitrary integer and rational implementations because they are hardly used for computation.

Also, for efficient computation, this interface contains the operations: approximate division, normalization, an embedding of Z, absolute value, power by N, shift by Z, and decision procedures for both equality and order.

```
Class AppDiv AQ := app_div : AQ → AQ → Z → AQ.
Class AppApprox AQ := app_approx : AQ → Z → AQ.
Class AppRationals AQ {e plus mult zero one inv} `{!Order AQ}
   {AQtoQ : Coerce AQ Q_as_MetricSpace} `{!AppInverse AQtoQ}
   {ZtoAQ : Coerce Z AQ} `{!AppDiv AQ} `{!AppApprox AQ}
   `{!Abs AQ} `{!Pow AQ N} `{!ShiftL AQ Z}
   `{∀ x y : AQ, Decision (x = y)} `{∀ x y : AQ, Decision (x ≤ y)} : Prop := {
```

```
aq_ring :> @Ring AQ e plus mult zero one inv ;
aq_order_embed :> OrderEmbedding AQtoQ ;
aq_ring_morphism :> SemiRing_Morphism AQtoQ ;
aq_dense_embedding :> DenseEmbedding AQtoQ ;
aq_div : ∀ x y k, B_{2^k} ('app_div x y k) ('x / 'y) ;
aq_approx : ∀ x k, B_{2^k} ('app_approx x k) ('x) ;
aq_shift :> ShiftLSpec AQ Z (≪) ;
aq_nat_pow :> NatPowSpec AQ N (^) ;
aq_ints_mor :> SemiRing_Morphism ZtoAQ }.
```

O'Connor [2] keeps the size of the rational numbers small to avoid efficiency problems. He introduced a function approx x ϵ that yields the 'simplest' rational number between x $- \varepsilon$ and x $+ \varepsilon$. In our interface we modify the approx function slightly: app_approx x k yields an arbitrary element between x $- 2^k$ and x $+2^k$. Using this function we define the compress operation on the real numbers: compress := bind (λ ϵ . app_approx x (Qdlog2ϵ)) such that compress x = x.

In the remainder of this section we briefly describe our implementation by the dyadics.

The dyadic rationals are numbers of the shape $n*2^e$ for $n, e \in \mathbb{Z}$. In order to remain independent of an integers implementation, we abstract over it. For our eventual implementation of the approximate rationals we use Coq's machine integers, bigZ. Now given an arbitrary integer implementation Int it is straightforward to define the dyadics. Here we will just show the ring operations.

```
Record Dyadic := dyadic { mant : Int ; expo : Int }.
Infix "$" := dyadic (at level 80).
Global Instance dy_inject: Coerce Int Dyadic := λ x, x $ 0.
Global Instance dy_opp: GroupInv Dyadic := λ x, −mant x $ expo x.
Global Instance dy_mult: RingMult Dyadic := λ x y, mant x ∗ mant y $ expo x + expo y.
Global Instance dy_0: RingZero Dyadic := ('0:Dyadic).
Global Instance dy_1: RingOne Dyadic := ('1:Dyadic).
Global Program Instance dy_plus: RingPlus Dyadic := λ x y,
  if decide_rel (≤) (expo x) (expo y)
  then mant x + mant y ≪ (expo y − expo x) ↾ _ $ min (expo x) (expo y)
  else mant x ≪ (expo x − expo y) ↾ _ + mant y $ min (expo x) (expo y).
```

Power series Elementary transcendental functions as exp, sin, ln and arctan can be defined by their power series. A power series is particularly suited for computation if its coefficients are alternating, decreasing and have limit 0. For $-1 \leq x \leq 0$,

$$\exp\ x = \sum_{i=0}^{\infty} \frac{x^i}{i!}$$

is of this form. To approximate exp x with error ε we take the partial sum until $\frac{x^i}{i!} \leq \varepsilon$.

In order to implement this efficiently we use a stream representing the series and define a function that sums the required number of elements. For example, the series 1, a, a^2, a^3, ... is defined by the following stream.

```
CoFixpoint powers_help (c : A) : Stream A := Cons c (powers_help (c ∗ a)).
Definition powers : Stream A := powers_help 1.
```

Streams in Coq, like lists in Haskell, are lazy. So, in the example the multiplications are accumulated.

Since Coq only allows structural recursion (and guarded co-recursion) it requires some work to convince Coq that our algorithm terminates. Intuitively one would describe the limit as an upperbound of the required number of elements using the Exists predicate.

```
Inductive Exists A (P : Stream A → Prop) (x : Stream) : Prop :=
  | Here : P x → Exists P x
  | Further : Exists P (tl x) → Exists P x.
```

This approach leads to performance problems. The upperbound, encoded in unary format, may become very large while generally only a few terms are necessary. Due to vm_compute's eager evaluation scheme, this unary number will be computed before summing the series. Instead we use LazyExists [10].

```
Inductive LazyExists A (P : Stream A → Prop) (x : Stream A) : Prop :=
  | LazyHere : P x → LazyExists P x
  | LazyFurther : (unit → LazyExists P (tl x)) → LazyExists P x.
```

An alternative solution would have been to Barras' trick to compute with well-founded recursions by lazily adding 2^n Acc_intro constructors in front of a well_foundedness proof, so that the actual proof is never reached in practice:

```
Fixpoint guard A (R: A -> A -> Prop) n (wfR: well_founded R): well_founded R :=
  match n with
    | 0 ⇒ wfR
    | S n ⇒ fun x ⇒ Acc_intro x (fun y _ ⇒ guard n (guard n wfR) y)
  end.
```

O'Connor's InfiniteAlternatingSum s returns the real number represented by the infinite alternating sum over s, where the stream s is decreasing, non-negative and has limit 0. We have extended this in two ways. First, by generalizing some of the work to abstract structures. Second, as we do not have exact division on approximate rationals, we extended his algorithm to work with approximate division. The latter required changing InfiniteAlternatingSum s to InfiniteAlternatingSum n d which computes the infinite alternating sum of the stream $\lambda i.\frac{n_i}{d_i}$. This change allows us to postpone divisions. We have to determine both the length of the partial sum and the required precision of the divisions. We obtain these values by finding a k such that:

$$\mathbf{B}_{\frac{\varepsilon}{2}}\ (\mathsf{app_div}\ n_k\ d_k\ (\mathsf{log}\frac{\varepsilon}{2k}) + \frac{\varepsilon}{2k})\ 0. \tag{1}$$

Here n_i and d_i denote the i-th element of the series n and d respectively. Now k is the length of the partial sum, and $\frac{\varepsilon}{2k}$ is the required precision of division. Using O'Connor results it is not too hard to verify that these values are correct and such a k indeed exists for a decreasing, non-negative stream with limit 0.

As noted above, we have specified the precision of division in powers of 2 instead of using a rational value. This is allows us to replace (1) with

$$\mathbf{B}_{\frac{\varepsilon}{2}}\ (\mathsf{app_div}\ n_k\ d_k\ (\mathsf{log}\ \varepsilon - (k+1)) + 1 \ll (\mathsf{log}\ \varepsilon - (k+1)))\ 0.$$

Here k is the length of the partial sum, and 2^l, where $l = \mathsf{log}\ \varepsilon - (k+1)$, is the required precision of division. This variant can be implemented without any arithmetic on the rationals and is thus much more efficient.

This method gives us a fast way to compute the infinite alternating sum, in practice, only few extra terms have to be computed and due to the approximate division the auxiliary results are kept as small as possible.

Using this method to compute infinite alternating sums we have so far implemented exp and arctan. Furthermore, we extend the exponential to its complete domain by repeatedly applying the following formula.

$$\mathsf{exp}\ x = (\mathsf{exp}(x \ll 1))^2 \tag{2}$$

Our tests have shown that reducing the input to a value between $-2^k \leq x \leq 0$ for $50 \leq k$ yields major performance improvements as the series will converge much faster. For higher precisions setting it to $75 \leq k$ gives even better results.

By defining arctan on $[0, 1)$, we can define

$$\pi := 176 * \mathsf{arctan}\frac{1}{57} + 28 * \mathsf{arctan}\frac{1}{239} - 48 * \mathsf{arctan}\frac{1}{682} + 96 * \mathsf{arctan}\frac{1}{12943}.$$

Since we do not have exact devision on the approximate rationals, we here see the purpose of parameterizing infinite sums by two streams.

Square root We use Wolfram's algorithm [11, p.913] for computing the square root. Its complexity is linear, in fact it provides a new binary digit in each step. We also have a Haskellprototype of Newton iteration which performance very well. We aim to verify it in Coq, perhaps building on the work by Julien and Pasca [12]. Our correctness proof should be substantially easier than theirs, as our exact reals automatically take care of the rounding. Moreover, our Haskell prototype indicates that the performance of our data structure is much better than their real numbers as streams of 'bits'.

```
Context `(Pa : 1 ≤ a ≤ 4).
Fixpoint AQroot_loop (n : nat) : AQ * AQ :=
  match n with
  | O ⇒ (a, 0)
  | S n ⇒
      let (r, s) := AQroot_loop n in
      if decide_rel (≤) (s + 1) r
      then ((r − (s + 1)) ≪ (2:Z), (s + 2) ≪ (1:Z))
      else (r ≪ (2:Z), s ≪ (1:Z))
  end.
```

Benchmarks The first step in this research was to create a Haskellprototype based on O'Connor's implementation of the real numbers in Haskell [2]. The second step was to implement this prototype in Coq. Currently, our Coq development contains the field operations, computation of power series, exp, arctan, π and the square root. The correctness of these operations has been verified in the Coq system. However, we have verified Wolfram's algorithm in Coq, whereas we use the much faster Newton algorithm in Haskell.

Expression	Decimals	O'Connor	Krebbers/Spitters
$\sin\ (\sin\ (\sin\ 1))$	10000	71s	5s
$\cos\ (10^{50})$	10000	2.7s	0.6s
$\tan\ (\sqrt{2}) + \operatorname{arctanh}\ (\sin\ 1)$	500	133s	2.2s

Table 1: HASKELL, compiled with `ghc` version 6.12.1, using `-O2`.

Expression	Decimals	O'Connor	Krebbers/Spitters
π	300	56s	1.1s
$\exp\ (\exp\ (\exp\ (\frac{1}{2})))$	25	123s	0.23s
$\exp\ \pi - \pi$	25	52s	0.1s
$\arctan\ \pi$	25	134s	1.4s

Table 2: COQ trunk revision 13841.

We present some benchmarks[3] comparing the old and the new implementation, both in HASKELLand COQ. Table 1 shows some benchmarks in HASKELLwith compiler optimizations enabled (`-O2`) and Table 2 compares our COQ implementation with O'Connor's. More extensive benchmarking shows that our HASKELLimplementation generally benefits from a 15 times speed while the speed up in COQ is usually more than a 100 times. This difference is explained by the fact that O'Connor's HASKELLimplementation already used fast integers, while his COQ implementation did not. In the same times as shown in Table 2 for the old implementation, the new implementation is able to compute the first 1750 decimals of π, 450 decimals of $\exp\ (\exp\ (\exp\ (\frac{1}{2})))$, 425 decimals of $\exp\ \pi - \pi$ and 85 decimals of $\arctan\ \pi$. This is an improvement of up to 18 times of the number of decimals.

We conclude this section with a comparison between the performance of Wolfram's algorithm in COQ and HASKELL. The HASKELLprototype (without compiler optimizations) is quite fast, computing 10,000 iterations (giving 3,010 decimals) of $\sqrt{2}$ takes 0.2s. In COQ it takes 7.5s using type classes and 7.2s without type classes. Here we exclude the time spend on type class resolution. Thus type classes cause only a 3% performance penalty on computations. The experimental native_compute performs evaluation by compilation to native OCAMLcode. This approach uses the OCAMLcompiler available and is interesting for heavy compilation. Our first experiments indicate a 10 times speed up with Wolfram iteration. Unfortunately, native_compute does not work with COQ trunk yet, so we were unable to test it with our implementation of the reals.

Conclusions We have greatly improved the performance of real number computation in COQ by using COQ's new machine integers. We produced highly structured and abstract code using type classes with no apparent performance penalty. Moreover, COQ's notation mechanism combined with unicode characters gives nicely readable statements and proofs. Type classes were a great help in our work. However, their current implementation in COQ is still experimental and at times too slow. Canonical structures provide an alternative, and

[3] All benchmarks have been carried out on an Intel Core Quad 2.4 GHz with 8GB of memory running DEBIAN GNU/LINUXwith kernel 2.6.32. The sources of our developments can be found at http://robbertkrebbers.nl/research/reals.

partially complementary, implementation of type classes [13]. By choice, structures restrict to deterministic proof search, this makes them more efficient, but also somewhat more intricate to use. The use of canonical structures by the SSREFLECTteam [14] makes it plausible that with some effort we could have used canonical structures for our work instead. However, the SSREFLECT-library is currently not suited for setoids which are crucial to us. The integration of unification hints [15] into COQ may allow a tighter integration of type classes and canonical structures. We needed to adapt our correctness proofs to prevent the virtual machine from eagerly evaluating them. Lazy evaluation for Prop would have allowed us to use the original proofs.

References

[1] Bishop, E.A.: Foundations of constructive analysis. McGraw-Hill (1967)

[2] O'Connor, R.: A Monadic, Functional Implementation of Real Numbers. MSCS **17**(1) (2007) 129–159

[3] O'Connor, R.: Certified Exact Transcendental Real Number Computation in Coq. In: TPHOLs 2008. Volume 5170 of LNCS. (2008) 246–261

[4] O'Connor, R., Spitters, B.: A computer verified, monadic, functional implementation of the integral. TCS **411**(37) (2010) 3386–3402

[5] Spitters, B., van der Weegen, E.: Type classes for mathematics in type theory. MSCS, special issue on "Interactive theorem proving and the formalization of mathematics" (2011)

[6] Armand, M., Grégoire, B., Spiwack, A., Théry, L.: Extending Coq with imperative features and its application to SAT verification. In: ITP 2010. Volume 6172 of LNCS. (2010) 83–98

[7] Spiwack, A.: Verified Computing in Homological Algebra, A Journey Exploring the Power and Limits of Dependent Type Theory. PhD thesis, INRIA (2011)

[8] Gonthier, G., Mahboubi, A., Tassi, E.: A Small Scale Reflection Extension for the Coq system. Technical Report RR-6455, INRIA (2008)

[9] Bauer, A., Kavkler, I.: A constructive theory of continuous domains suitable for implementation. Annals of Pure and Applied Logic **159**(3) (2009) 251–267

[10] O'Connor, R.: Incompleteness and Completeness: Formalizing Logic and Analysis in Type Theory. PhD thesis, Radboud University Nijmegen (2009)

[11] Wolfram, S.: A new kind of science. Wolfram Media (2002)

[12] Julien, N., Pasca, I.: Formal Verification of Exact Computations Using Newton's Method. In: TPHOLs 2009. Volume 5674 of LNCS. (2009) 408–423

[13] Gonthier, G., Ziliani, B., Nanevski, A., Dreyer, D.: Making ad hoc proof automation less ad hoc. (2011)

[14] Garillot, F., Gonthier, G., Mahboubi, A., Rideau, L.: Packaging mathematical structures. In: TPHOLs 2009. Volume 5674 of LNCS. (2009) 327–342

[15] Asperti, A., Ricciotti, W., Coen, C., Tassi, E.: Hints in Unification. In: TPHOLs 2009. Volume 5674 of LNCS. (2009) 84–98