Proving Inequalities over Reals with Computation in Isabelle/HOL

Johannes Hölzl *
Technische Universität München
hoelzl@in.tum.de

Abstract

When verifying numerical algorithms, it is often necessary to estimate inequalities over reals. Unfortunately, the large amount of numerical computations this requires significantly complicates manual deductive-style proofs.

We present an automatic proof method for inequalities over reals with bounded variables. The method translates inequalities into interval arithmetic calculations on floating-point numbers. The result is then evaluated using the code generator. The translation and evaluation is verified in Isabelle/HOL. Our main contributions are the boundary computations for square root, π, $\sin$, $\cos$, $\arctan$, $\exp$, and $\ln$ in Isabelle/HOL. The proof method is included in Isabelle2009.

Keywords inequalities over reals, interval arithmetic, decision procedure, Isabelle/HOL, computation

1. Introduction

When verifying a numerical algorithm or a computer system related to a physical domain, we need to prove many properties about real-valued formulas. An interesting class of such formulas consists of inequalities involving trigonometric functions and basic arithmetic operations with bounded variables.

As an example, [Daumas et al. 2009] cite the following formula that arose while verifying a flight control system:

$$\frac{3 \cdot \pi}{180} \leq \frac{g}{v} \cdot \tan\left(\frac{35 \cdot \pi}{180}\right) \leq \frac{3.1 \cdot \pi}{180}$$

$$v = 128.6\,\tfrac{m}{s} \quad \textit{velocity of the aircraft}$$

$$g = 9.81\,\tfrac{m}{s^2} \quad \textit{gravitational force}$$

They also mention that "a direct proof of this formula is about a page long and requires the proof of several trigonometric properties."

Real-valued inequalities also arise in the verification of numerical libraries. The algorithms provided by such libraries are often easy to verify, especially since theorem provers such as Isabelle/HOL provide libraries formalizing large parts of real analysis. However, numerical algorithms often rely on approximations of constants such as π and $\ln 2$ or on tables of precomputed numbers, which must be verified as well. The formalization of the CORDIC algorithm in [Harrison 1996] is as example of such a verification.

To verify real-valued inequalities, one option is to check them using a computer algebra system (CAS) such as Maple and Mathematica. These systems are designed to compute real functions and are very efficient at it.

However, it is not always possible to use CASs for program verification, because they are unsound and might define elementary functions slightly differently than the theorem prover used for the rest of the verification.[1] [Harrison and Théry 1993] give formulas for which CASs compute wrong results; for example, Maple evaluates the integral $\int_{-1}^{1} \sqrt{x^2}\,dx$ to 0 instead of 1, and Mathematica evaluates the undefined integral $\int_{-1}^{1} \frac{1}{\sqrt{x^2}}\,dx$ to 0.

Hence, we are interested in decision procedures that can be integrated in a theorem prover. Equalities involving only real numbers and basic arithmetic operations are easy to solve. Consider the equation

$$2 \cdot 3^5 - 27/3^2 = 483$$

To verify such an equation, we can use rewrite rules to replace the power operation, the division, and the subtraction, leaving only sums and products:

$$2 \cdot 3 \cdot 3 \cdot 3 \cdot 3 \cdot 3 \cdot 3 \cdot 3 = 483 \cdot 3 \cdot 3 + 27$$

Then we can apply rewrite rules implementing addition and multiplication. The rewrite rules required by this transformation are easy to establish. Assuming a binary representation for numerals, this technique is also fast enough for most purposes.

Unfortunately, this approach does not work when applied directly to transcendental functions, since they would need infinitely many basic arithmetic operations, and their result is generally not finitely representable. Instead, we approximate transcendental function using finite means. We approximate a real number by an interval between two floating-point numbers. This requires defining the approximation functions in Isabelle/HOL and proving their correctness. To prove an inequality, we approximate both sides and compare the resulting intervals. If the inequality holds for all numbers in the respective intervals, the theorem is proved.

The main contributions of the work presented in this paper [Hölzl 2009] are the following:

1. Formalizing upper-bound and lower-bound approximations of the most important elementary transcendental functions ($\sin$, $\cos$, $\arctan$, $\exp$, and $\ln$) and of square root in Isabelle/HOL.

* Supported by the DFG Graduiertenkolleg 1480 (PUMA).

PLMMS '09 August 21, 2009, Munich, Germany.

[1] For example, $x/0 = 0$ in Isabelle/HOL. Most CASs implement $x/0$ as undefined, which is not possible in Isabelle/HOL.

2. Implementing an evaluation function to approximate the result of a real arithmetic expression by an interval that contains the exact result.
3. Implementing a proof method for Isabelle/HOL that automatically verifies inequalities with bounded variables.

These contributions are included in Isabelle2009. The theory file `src/HOL/Decision_Procs/Approximation.thy` in the Isabelle source code archive implements the `approximation` method. The definitions, lemmas, and theorems presented in section 3 are directly generated from this sources. The proof for each lemma and theorem can also be found in this file.

2. Showcase Overview

In this section, we briefly review the implemented proof method by considering the simple formula $\arctan 1.5 < 1$ and following it step by step. Each line starting with a lowercase and boldface word is an Isabelle command. After each proof command we show the (numbered) subgoals to be proved.

It could seem that our proof method is not needed for this example, by simply using the identity $\arctan \frac{\pi}{2} = 1$ and applying the strict monotonicity of arc tangent. However, this would require proving the inequality $2 \cdot 1.5 < \pi$, which is not trivial to do by hand.

theorem *arctan-1-5-less-one*: *arctan 1.5 < 1*

1. arctan $\frac{15}{10} < 1$ (1.5 is internally represented as $\frac{15}{10}$)

Reify: The proof method relies on functions and datatypes defined in the Isabelle/HOL logic. The first step is to convert the formula into a value of an appropriate arithmetic expression datatype, a process called reification. The $interp_{ie}$ function evaluates the arithmetic expression, yielding an equivalent formula. The simplification rules *interpret-inequality-equations* define the semantics of our arithmetic.

apply (*reify interpret-inequality-equations*)

1. $interp_{ie}$
(*Less* (*Arctan* (*Mult* (*Num 15*) (*Inverse* (*Num 10*)))) (*Num 1*))
[]

Rewrite as approximation: Since $interp_{ie}$ maps arithmetic expressions directly to operations on real numbers, it is generally not executable. Therefore, the next step is to rewrite the goal so that it uses the approximation function $approx_{ie}$. This is achieved by applying the rule APPROX_INEQUALITY (*If bounded-by vs bs and* $approx_{ie}$ *prec eq bs then* $interp_{ie}$ *eq vs*.) on the goal.

APPROX_INEQUALITY is the central theorem in our work. It simply states the correctness of our approximation. By applying it to the reified formula, we obtain a call to $approx_{ie}$, which computes the formula using floating-point interval arithmetic. When the inequality holds for the bounds of the intervals, it also holds for the real numbers approximated by the intervals.

apply (*rule approx-inequality*[**where** *prec=10* **and** *bs=*[]])

1. bounded-by [] []
2. $approx_{ie}$ *10*
(*Less* (*Arctan* (*Mult* (*Num 15*) (*Inverse* (*Num 10*)))) (*Num 1*))
[]

Prove bounds: If the formula contains variables, we require each variable to belong to a fixed interval, which is then used as an approximation of the variable. In this example, no variables are used.

apply *simp*

1. $approx_{ie}$ *10*
(*Less* (*Arctan* (*Mult* (*Num 15*) (*Inverse* (*Num 10*)))) (*Num 1*))
[]

Evaluate: The $approx_{ie}$ function can be executed using the ML code generator of Isabelle/HOL. When the result of the computation is true, the inequality is proved. Otherwise, all we know is that it cannot be stated using the given precision parameter (which we passed when instantiating APPROX_INEQUALITY). In these cases, we can try again with a higher precision.

apply *eval*
done

Instead of writing the above proof steps, we can use the *approximation* proof method to perform the same steps automatically. Its parameter specifies the precision to use.

theorem *arctan 1.5 < 1* **by** (*approximation 10*)

3. Implementation

3.1 Reification

The first step of the proof method is to reify the inequality to prove, meaning that we convert it into a value of our arithmetic expression datatype.

Example AN ARITHMETIC OF ADDITION*:*
— The *arith* datatype specifies the syntactic form of our expressions:
datatype *arith* = *Add arith arith* | *Num nat* | *Atom nat*

— *eval* is the interpretation function (i.e., the semantics of *arith*):
fun *eval* :: *arith* ⇒ *nat list* ⇒ *nat* **where**
eval (*Add a b*) *xs* = *eval a xs* + *eval b xs* |
eval (*Num x*) *xs* = *x* |
eval (*Atom n*) *xs* = *xs* ! *n*

Using this setup, we can now apply reification to a simple term:

$$3 + a \equiv eval\ (Add\ (Num\ 3)\ (Atom\ 0))\ [a]$$

Next, we would define a function in HOL that analyzes the supplied *arith* value and prove the correctness of the function.

The reification step relies on Chaieb's generic reification mechanism [Chaieb 2008]. In this framework, a HOL term of type τ is converted to a value of a datatype δ applied to an interpretation function $[\![\ \_\ ]\!]_{\_} :: \delta \Rightarrow \tau\ list \Rightarrow \tau$. To use the framework, we must provide rewrite equations for $[\![\ \_\ ]\!]_{\_}$ that specify how τ terms are mapped to δ values. As rewrite equations, we can normally use the definitional equations of $[\![\ \_\ ]\!]_{\_}$. For each type τ, we can specify as many types δ as we wish, each with its own semantic function $[\![\ \_\ ]\!]_{\_}$.

Reification then works as follows. For simplicity, we consider a single $[\![\ \_\ ]\!]_{\_}$ function for fixed τ and δ. First, we provide equations for $[\![\ \_\ ]\!]_{\_}$:

$$\begin{array}{lcl}
[\![C_1\ x_1\ \ldots\ x_m]\!]_{xs} & = & P_1\ [\![x_1]\!]_{xs}\ \ldots\ [\![x_m]\!]_{xs} \\
[\![C_2\ x_1\ \ldots\ x_m]\!]_{xs} & = & P_2\ [\![x_1]\!]_{xs}\ \ldots\ [\![x_m]\!]_{xs} \\
 & \vdots & \\
[\![C_n\ x_1\ \ldots\ x_m]\!]_{xs} & = & P_n\ [\![x_1]\!]_{xs}\ \ldots\ [\![x_m]\!]_{xs} \\
[\![Atom\ i]\!]_{xs} & = & xs_{[i]}
\end{array}$$

Here, the x_i's are variables, the C_j's are constructors of δ, and the P_k's are arbitrary terms. The equations specify how the constructors are interpreted. The left-hand sides are allowed to overlap. The generic reification algorithm tries to match a right-hand side P_k with the term to reify. When P_k matches, the corresponding constructor C_k is used to build the δ value. The previous step is then applied recursively to the arguments to P_k. If the term to

reify is a variable x, the last equation is used, and x is added to the variable list *xs*.

Reification was introduced by [Moore and Boyer 1981]. A more detailed overview of reification in Isabelle/HOL is provided by [Chaieb 2008].

3.2 Syntax and Semantics

Before we can use reification, we must define the datatype into which the *real* terms and inequalities are reified. We start with *real* terms:

Definition floatarith:

datatype *floatarith* = *Add floatarith floatarith*
| *Minus floatarith* | *Mult floatarith floatarith*
| *Inverse floatarith* | *Sin floatarith* | *Cos floatarith*
| *Arctan floatarith* | *Abs floatarith*
| *Max floatarith floatarith* | *Min floatarith floatarith* | *Pi*
| *Sqrt floatarith* | *Exp floatarith* | *Ln floatarith*
| *Power floatarith nat* | *Atom nat* | *Num float*

The *Atom* constructor represents a variable by index. The other constructors describe the arithmetic operations that we support in our interval arithmetic.

Next, we define $interp_{fa}$ *e xs* to evaluate the expression *e* with the list of variables *xs*. The *n*th element of the list *xs* corresponds to *Atom* $(n-1)$. This essentially establishes the mapping from *floatarith* constructors to the arithmetic operations.

Definition $interp_{fa}$ :: *floatarith* ⇒ *real list* ⇒ *real*:

$interp_{fa}$ (*Add a b*) *vs* = $interp_{fa}$ *a vs* + $interp_{fa}$ *b vs*
$interp_{fa}$ (*Minus a*) *vs* = − $interp_{fa}$ *a vs*
$interp_{fa}$ (*Mult a b*) *vs* = $interp_{fa}$ *a vs* · $interp_{fa}$ *b vs*
$interp_{fa}$ (*Inverse a*) *vs* = *inverse* ($interp_{fa}$ *a vs*)
$interp_{fa}$ (*Sin a*) *vs* = *sin* ($interp_{fa}$ *a vs*)
$interp_{fa}$ (*Cos a*) *vs* = *cos* ($interp_{fa}$ *a vs*)
$interp_{fa}$ (*Arctan a*) *vs* = *arctan* ($interp_{fa}$ *a vs*)
$interp_{fa}$ (*Min a b*) *vs* = *min* ($interp_{fa}$ *a vs*) ($interp_{fa}$ *b vs*)
$interp_{fa}$ (*Max a b*) *vs* = *max* ($interp_{fa}$ *a vs*) ($interp_{fa}$ *b vs*)
$interp_{fa}$ (*Abs a*) *vs* = |$interp_{fa}$ *a vs*|
$interp_{fa}$ *Pi vs* = *pi*
$interp_{fa}$ (*Sqrt a*) *vs* = *sqrt* ($interp_{fa}$ *a vs*)
$interp_{fa}$ (*Exp a*) *vs* = *exp* ($interp_{fa}$ *a vs*)
$interp_{fa}$ (*Ln a*) *vs* = *ln* ($interp_{fa}$ *a vs*)
$interp_{fa}$ (*Power a n*) *vs* = $(interp_{fa}\ a\ vs)^n$
$interp_{fa}$ (*Num f*) *vs* = *f*
$interp_{fa}$ (*Atom n*) *vs* = $vs_{[n]}$

We can support additional operations using appropriate lemmas, assuming they can be expressed in terms of *floatarith*:

Lemma FLOATARITH-ADDITIONAL-ARITHMETIC**:**

$interp_{fa}$ (*Mult a* (*Inverse b*)) *vs* = $\frac{interp_{fa}\ a\ vs}{interp_{fa}\ b\ vs}$
$interp_{fa}$ (*Add a* (*Minus b*)) *vs* = $interp_{fa}$ *a vs* − $interp_{fa}$ *b vs*
$interp_{fa}$ (*Mult* (*Sin a*) (*Inverse* (*Cos a*))) *vs* = *tan* ($interp_{fa}$ *a vs*)
$interp_{fa}$ (*Exp* (*Mult b* (*Ln a*))) *vs* = $interp_{fa}$ *a vs powr* $interp_{fa}$ *b vs*
$interp_{fa}$ (*Mult* (*Ln x*) (*Inverse* (*Ln b*))) *vs* =
log ($interp_{fa}$ *b vs*) ($interp_{fa}$ *x vs*)

Here is an example of how a *real* term maps to its *floatarith* representation:

Example FLOATARITH-EXAMPLES**:**

$\frac{1}{sin\ x} - cos\ y =$
$interp_{fa}$
(*Add* (*Mult* (*Num 1*) (*Inverse* (*Sin* (*Atom 1*))))
(*Minus* (*Cos* (*Atom 2*))))
[*x*, *y*]

Next, we introduce a datatype *inequality* that represents the syntax of inequalities and a function $interp_{ie}$ that defines their semantics. We distinguish between $<$ and $\leq$.

Definition inequality:

datatype *inequality* = *Less floatarith floatarith*
| *LessEqual floatarith floatarith*

Definition $interp_{ie}$ :: *inequality* ⇒ *real list* ⇒ *bool*:

$interp_{ie}$ (*Less a b*) *vs* = ($interp_{fa}$ *a vs* < $interp_{fa}$ *b vs*)
$interp_{ie}$ (*LessEqual a b*) *vs* = ($interp_{fa}$ *a vs* ≤ $interp_{fa}$ *b vs*)

With the definitions and lemmas presented in this section, we can call the *reify* proof method to rewrite an HOL inequality into an *inequality* value. Thanks to Chaieb's generic reification framework, everything is done inside the logic without needing any custom ML code specific to the syntax.

3.3 Interval Arithmetic

Once the formula has been reified into a *inequality* value, we must compute both sides. Since the result of transcendental functions like arc tangent is generally not finitely representable, we approximate them using floating-point interval arithmetic. At the approximation level, a real value x is represented by a pair of floating-point numbers $(\underline{x}, \overline{x})$, which denotes the interval[2] $\{\underline{x} \mathinner{..} \overline{x}\}$.

Instead of floating-point numbers, we could have used rational numbers, but floating-point numbers are easier to implement efficiently. Another alternative would have been to use exact arithmetic [Lester and Gowland 2003, Harrison 1996]. In this setting, each real value x is represented by a function $f_x(p)$ that returns an integer approximating x at precision p: $|f_x(p) - x \cdot 2^p| < 1$. However, this approach requires us to prove the above inequality for each operation, something we avoid by using interval arithmetic.

Comparing intervals works as follows: $\{\underline{x} \mathinner{..} \overline{x}\}$ is less than $\{\underline{y} \mathinner{..} \overline{y}\}$ if $\overline{x} < \underline{y}$. Ideally, we would like the intervals to be as small as possible. For example, knowing that $\sin(x)$ and $\cos(x)$ belong to the interval $\{-1 \mathinner{..} 1\}$ would not even suffice to show the trivial fact $\sin(0) < \cos(0)$.

The computations are parameterized by a used-specified *precision* p that controls the bit width of the floating-point numbers. For terms involving a single elementary operation, a floating-point value $x = m \cdot 2^e$ is approximated by bounds respecting the constraints

$$\overline{x} - x \quad \leq \quad 2^{e-p}$$

$$x - \underline{x} \quad \leq \quad 2^{e-p}$$

This guarantee also holds for cos and sin applied to values belonging to $\{-\pi \mathinner{..} \pi\}$ and $\{-\pi/2 \mathinner{..} \pi/2\}$, respectively.

3.4 Floating-Point Arithmetic

The floating-point numbers are formalized in the `ComputeFloat` theory developed for the Flyspeck II project [Obua 2008]. It represents floating-point numbers as a mantissa and an exponent, both of which must be integers:

datatype *float* = *Float int int*

Definition *mantissa* :: *float* ⇒ *int*: *mantissa* (*Float m e*) = *m*

Definition *exponent* :: *float* ⇒ *int*: *exponent* (*Float m e*) = *e*

Isabelle's integers are mapped to arbitrary precision integers in ML when compiled by the code generator.

We overload the polymorphic constant *real* to interpret *float* values as *real* values:

Definition *pow2* :: *int* ⇒ *real*:
pow2 e = (*if* 0 ≤ *e then* 2^e *else inverse* 2^{-e})

[2] We use the Isabelle-specific notation with braces rather than the mathematical notation $[\underline{x}, \overline{x}]$.

Definition *real* :: *float* ⇒ *real*:
real (*Float a b*) = *real a* · *pow2 b*

For the operations on floating-point numbers, we omit the implementation of the operations itself. They obey the following obvious equations:

Lemma FLOAT_ADD**:** *real* $(a+b)$ = *real a* + *real b*
Lemma FLOAT_MINUS**:** *real* $(-a)$ = $-$ *real a*

Lemma FLOAT_SUB**:** *real* $(a-b)$ = *real a* − *real b*

Lemma FLOAT_MULT**:** *real* $(a \cdot b)$ = *real a* · *real b*

Lemma FLOAT_POWER**:** *real* $x^n = (real\ x)^n$

Such an equation is not possible for division, which is implemented by lower-bound and upper-bound functions, parameterized with the precision:

Lemma FLOAT_DIVL**:** *real* (*float-divl prec x y*) $\leq \frac{real\ x}{real\ y}$
Lemma FLOAT_DIVR**:** $\frac{real\ x}{real\ y} \leq$ *real* (*float-divr prec x y*)

These operations are used in the next section to implement the elementary operations. As a result, the ML code generated from the Isabelle theories uses our arbitrary-precision *float* type, rather than ML's machine floating-point numbers.

For clarity, the *real* constant is implicit in the rest of this paper.

3.5 The Approximation Function

The core of our decision procedure is the $approx_{ie}$ function, which returns *True* if the given *inequality* holds for the specified variable bounds and precision; otherwise, it returns *False*:

Definition
$approx_{ie}$ :: *nat* ⇒ *inequality* ⇒ (*float* × *float*) *list* ⇒ *bool*:
$approx_{ie}$ *prec* (*Less a b*) *bs* =
  (*case* ($approx_{fa}$ *prec a bs*, $approx_{fa}$ *prec b bs*) *of*
   (*None*, *b*) ⇒ *False* | (⌊(*l*, *u*)⌋, *None*) ⇒ *False*
   | (⌊(*l*, *u*)⌋, ⌊(*l′*, *u′*)⌋) ⇒ *u* < *l′*)
$approx_{ie}$ *prec* (*LessEqual a b*) *bs* =
  (*case* ($approx_{fa}$ *prec a bs*, $approx_{fa}$ *prec b bs*) *of*
   (*None*, *b*) ⇒ *False* | (⌊(*l*, *u*)⌋, *None*) ⇒ *False*
   | (⌊(*l*, *u*)⌋, ⌊(*l′*, *u′*)⌋) ⇒ *u* ≤ *l′*)

The $approx_{ie}$ function relies on

Definition
$approx_{fa}$ :: *nat* ⇒ *floatarith* ⇒ (*float* × *float*) *list* ⇀ *float* × *float*:
which approximately evaluates an arithmetic expression. It computes an appropriate interval based on the specified precision if the expression is defined for the given variable bounds; otherwise, it returns *None*. The function is defined by cases following the structure of the *floatarith* datatype.

$approx_{fa}$ *prec* (*Num f*) *bs* = ⌊(*f*, *f*)⌋
$approx_{fa}$ *prec* (*Atom i*) *bs* = (*if i* < |*bs*| *then* ⌊$bs_{[i]}$⌋ *else None*)

The equation for *Atom* looks up the bounds of the variable in the list *bs*.

For the remaining equations, we need the following auxiliary functions:

approx′ calls $approx_{fa}$ with an expression and rounds the resulting interval's bounds to the supplied precision.

lift-un, *lift-un′*, and *lift-bin′* lift the boundary functions to a function of type (*float* × *float*) *option*. If the result of the previous operation was *None*, they return *None*. *lift-bin′* and *lift-un′* also pass the result to a ⌊_⌋ constructor to get a (*float* × *float*) *option*.

The implementation of $approx_{fa}$ for addition and unary minus on intervals is based on the equations

$$\begin{aligned}(\underline{a},\overline{a}) + (\underline{b},\overline{b}) &= (\underline{a}+\underline{a}, \overline{a}+\overline{b})\\ -(\underline{a},\overline{a}) &= (-\overline{a}, -\underline{a})\end{aligned}$$

Equipped with our auxiliary functions, they are straightforward to implement:

$approx_{fa}$ *prec* (*Add a b*) *bs* =
  lift-bin′ (*approx′ prec a bs*) (*approx′ prec b bs*)
   (λ*l1 u1 l2 u2*. (*l1* + *l2*, *u1* + *u2*))
$approx_{fa}$ *prec* (*Minus a*) *bs* =
  lift-un′ (*approx′ prec a bs*) (λ*l u*. (− *u*, − *l*))

The *min*, *max*, and |_| operators are implemented similarly:

$approx_{fa}$ *prec* (*Min a b*) *bs* =
  lift-bin′ (*approx′ prec a bs*) (*approx′ prec b bs*)
   (λ*l1 u1 l2 u2*. (*min l1 l2*, *min u1 u2*))
$approx_{fa}$ *prec* (*Max a b*) *bs* =
  lift-bin′ (*approx′ prec a bs*) (*approx′ prec b bs*)
   (λ*l1 u1 l2 u2*. (*max l1 l2*, *max u1 u2*))
$approx_{fa}$ *prec* (*Abs a*) *bs* =
  lift-un′ (*approx′ prec a bs*)
   (λ*l u*. (*if l* < 0 ∧ 0 < *u then 0 else min* |*l*| |*u*|, *max* |*l*| |*u*|))

For multiplication, we must perform a case distinction on the signs of the lower and upper bounds of the operands' intervals. To ease this, we introduce two auxiliary functions:

$$pprt\ x = \begin{cases} x & \text{when } x \geq 0 \\ 0 & \text{otherwise} \end{cases}$$

$$nprt\ x = \begin{cases} x & \text{when } x \leq 0 \\ 0 & \text{otherwise} \end{cases}$$

They conveniently allow us to write the lower and upper bounds of the product as a sum that considers all four cases:

$approx_{fa}$ *prec* (*Mult a b*) *bs* =
  lift-bin′ (*approx′ prec a bs*) (*approx′ prec b bs*)
   (λ*a1 a2 b1 b2*.
     (*nprt a1* · *pprt b2* + *nprt a2* · *nprt b2* +
      pprt a1 · *pprt b1* + *pprt a2* · *nprt b1*,
      pprt a2 · *pprt b2* + *pprt a1* · *nprt b2* +
      nprt a2 · *pprt b1* + *nprt a1* · *nprt b1*))

Our syntax explicitly supports x^k where k is a natural number. This adds no expressive power since we could rewrite x^k into $x \cdot \cdots \cdot x$ (k times), but we can compute x^k both more efficiently and in some cases with a smaller bound interval.

$approx_{fa}$ *prec* (*Power a n*) *bs* =
  lift-un′ (*approx′ prec a bs*)
   (λ*l u*. *if odd n* ∨ 0 < *l then* (l^n, u^n)
        else if u < 0 *then* (u^n, l^n) *else* (0, $(max\ (-l)\ u)^n$))

For $1/x$, we first checks if 0 is in x's interval. If it is, we return *None*; otherwise, we compute the lower and upper bounds of $1/x$ using *float-divl* and *float-divr*, which divide at the specified precision.

$approx_{fa}$ *prec* (*Inverse a*) *bs* =
  lift-un (*approx′ prec a bs*)
   (λ*l u*. *if* 0 < *l* ∨ *u* < 0
        then (⌊*float-divl prec 1 u*⌋, ⌊*float-divr prec 1 l*⌋)
        else (*None*, *None*))

The remaining constructors of *floatarith* correspond to sin, cos, π, arctan, $\sqrt{\ }$, exp, and ln. For those that are monotonic, we have separate functions for lower and upper bounds. We will see how to define them in section 3.7.

approx$_{fa}$ *prec* (*Sin a*) *bs* =
  lift-un′ (*approx*′ *prec a bs*) (*bnds-sin prec*)
approx$_{fa}$ *prec* (*Cos a*) *bs* =
  lift-un′ (*approx*′ *prec a bs*) (*bnds-cos prec*)
approx$_{fa}$ *prec Pi bs* = ⌊(*lb-pi prec*, *ub-pi prec*)⌋
approx$_{fa}$ *prec* (*Arctan a*) *bs* =
  lift-un′ (*approx*′ *prec a bs*)
   (λ*l u*. (*lb-arctan prec l*, *ub-arctan prec u*))
approx$_{fa}$ *prec* (*Sqrt a*) *bs* =
  lift-un (*approx*′ *prec a bs*)
   (λ*l u*. (*lb-sqrt prec l*, *ub-sqrt prec u*))
approx$_{fa}$ *prec* (*Exp a*) *bs* =
  lift-un′ (*approx*′ *prec a bs*)
   (λ*l u*. (*lb-exp prec l*, *ub-exp prec u*))
approx$_{fa}$ *prec* (*Ln a*) *bs* =
  lift-un (*approx*′ *prec a bs*) (λ*l u*. (*lb-ln prec l*, *ub-ln prec u*))

As we have seen in section 3.2, subtraction, division, $\tan$, x^y, and $\log_x$ are implemented as rewrite rules in terms of the other operations. For these, no equations are needed for *approx*$_{fa}$.

To use *approx*$_{fa}$ in a theorem prover, we must show its correctness, which means the following: If the result of *approx*$_{fa}$ lies in the interval represented by the pair (l, u), the result of *interp*$_{fa}$ is between l and u.

Naturally, this holds only if the input intervals bound the exact values. This is expressed by the *bounded-by vs bs* predicate, where *vs* are the variables and *bs* are the corresponding bounds. The proof method uses the simplifier to prove *bounded-by vs bs* from the list of assumptions provided by the user. The predicate *bounded-by* is implemented recursively:

Definition *bounded-by* :: *real list* ⇒ (*float* × *float*) *list* ⇒ *bool*:

bounded-by (*v* # *vs*) ((*l*, *u*) # *bs*) =
((*l* ≤ *v* ∧ *v* ≤ *u*) ∧ *bounded-by vs bs*)
bounded-by [] [] = *True*
bounded-by [] (*v* # *va*) = *False*
bounded-by (*v* # *va*) [] = *False*

The correctness theorem for *approx*$_{fa}$ follows:

Theorem APPROX**:**
If bounded-by xs vs and ⌊(*l*, *u*)⌋ = *approx*$_{fa}$ *prec expr vs then* *l* ≤ *interp*$_{fa}$ *expr xs* ∧ *interp*$_{fa}$ *expr xs* ≤ *u*.

The proof is by straightforward structural induction on *expr*.

From the correctness of *approx*$_{fa}$, it is easy to show the correctness of *approx*$_{ie}$:

Theorem APPROX_INEQUALITY**:**
If bounded-by vs bs and approx$_{ie}$ *prec eq bs then interp*$_{ie}$ *eq vs*.

3.6 Taylor Series

We compute the lower and upper bounds of transcendental functions using Taylor series. For $\ln$ and $\arctan$, we had to prove that their Isabelle/HOL definitions (which are defined as the inverse of $\exp$ and $\tan$) are equivalent to their Taylor series on their convergence areas. The other functions are already defined as Taylor series in Isabelle/HOL.

Although series are generally not computable, lower and upper bounds for a specific precision can often be computed as follows. Let f be the transcendental function for which we want to compute lower and upper bounds. The series is defined on an interval R_f, which need not be the entire reals:

$$x \in R_f \qquad \Longrightarrow \qquad f(x) = \sum_{i=0}^{\infty} (-1)^i \cdot \frac{1}{a_i} \cdot x^i \qquad (1)$$

For all functions f under consideration, a_i is a positive monotonic null sequence. Furthermore,[3] we assume that $x \in \{0 \,..\, 1\}$. In the next section, we will see how to circumvent this restriction. To compute the bounds, we consider the *partial sums* S_n of the Taylor series:

$$S_n = \sum_{i=0}^{n} (-1)^i \cdot \frac{1}{a_i} \cdot x^i$$

For all functions under consideration, the Taylor series is an alternating power series, and therefore S_{2n} is a lower bound and S_{2n+1} is an upper bound.

From (1) we know that $\lim_{n\to\infty} S_n = f(x)$. Since a_i is a monotonic positive null sequence, $\{S_{2n+1} \,..\, S_{2n}\}$ forms a sequence of nested intervals. Moreover, since $\lim_{n\to\infty} S_n = f(x)$, we know that $f(x)$ is always in these intervals. Hence, for an even n, the partial sum S_n is a upper bound, and for an odd n, the partial sum S_n is a lower bound of $f(x)$. The difference between the two partial sums is

$$S_{2n} - S_{2n+1} = \frac{1}{a_{2n+1}} \cdot x^{2n+1} \qquad (2)$$

The error on the upper bound is $S_{2n} - f(x) \leq (1/a_{2n+1}) \cdot x^{2n+1}$. This inequality enables us to determine n for the desired precision p.

For $\sin$, $\cos$, and $\exp$, it can be shown that $1/a_{2n+1} \leq 1/2^{2n+1}$. We show that S_p is a upper bound of precision p.

Let p' be a precision such that $2 \cdot p' \geq p$. Since $x \in \{0..1\}$, x can be written as $m \cdot 2^e$, with $1 \leq m < 2$ and $e \leq 0$. Using these notations and (2), we obtain the following upper bound estimate:

$$\begin{aligned} S_p - f(x) &\leq S_{2p'} - S_{2p'+1} \\ &= \frac{1}{a_{2p'+1}} \cdot x^{2p'+1} \\ &\leq \frac{1}{2^{2p'+1}} \cdot x \\ &\leq \frac{1}{2^{p+1}} \cdot m \cdot 2^e \\ &\leq 2^{e-p} \end{aligned}$$

The estimation of the lower bound works in a similar way.

To compute the partial sums S_m, we use the Horner scheme. Let $b_n = a_{m-n-1}$, and let $h(m, x)$ be the function defined by the equations

$$\begin{aligned} h(0, x) &= 0 \\ h(n+1, x) &= \frac{1}{b_n} - x \cdot h(n, x) \end{aligned}$$

Clearly, $h(m, x)$ equals S_m. If $h(m, x)$ is computed as a floating-point number, it is not possible to compute the coefficients $1/b_n$ exactly. Hence, we must split the computation function into two parts: one for the lower bound, $\underline{h}$, and one for the upper bound, $\overline{h}$. The polynomials computed in the recursive call are subtracted from the current coefficient. Since subtraction swaps the bounds, $\underline{h}$ and $\overline{h}$ are mutually recursive:

$$\begin{aligned} \overline{h}(0, x) &= 0 \\ \overline{h}(n+1, x) &= \overline{1/b_n} - x \cdot \underline{h}(n, x) \\ \underline{h}(0, x) &= 0 \\ \underline{h}(n+1, x) &= \underline{1/b_n} - x \cdot \overline{h}(n, x) \end{aligned}$$

[3] Exceptionally, for exp, we must assume $x \in \{-1 \,..\, 0\}$ instead.

3.7 Approximating Transcendental Functions

In section 3.5, we glossed over the implementation of the lower and upper bounds for transcendental functions. They are formalized as follows:

Power series The transcendental function is represented as a power series approximated as a finite polynomial, which in turn is computed using the Horner scheme. Unfortunately, the approximation can only be used on a limited convergence area, as we saw in the previous section.

Constants We use the approximation of the power series to compute π and $\ln 2$.

Bounding functions Trigonometrical identities enable us to compute the transcendental functions on the entire definition range. To accomplish this, we use the power series and the computed constants.

To illustrate how to define the bounding functions for transcendental functions, we will restrict our attention to $\arctan$. The other functions are implemented in a somewhat similar way.

For $\arctan$, we must prove that its definition is equivalent to its Taylor series on its convergence area:

Lemma ARCTAN_SERIES**:**
If $|x| \leq 1$ *then* $\mathit{arctan}\ x = \sum_{k=0}^{\infty} (-1)^k \cdot \frac{1}{k \cdot 2 + 1} \cdot x^{k \cdot 2 + 1}$.

To evaluate this series, we introduce an auxiliary function $h_n(x)$ computing its partial sums:

$$h_n(x) = \sum_{k=0}^{n-1} (-1)^k \cdot \frac{1}{k \cdot 2 + 1} \cdot x^k$$

The exponent of x does not match the one in the Taylor series, but by massaging it slightly we can make it comply:

$$x \cdot h_n(x^2) = \sum_{i=0}^{n-1} (-1)^k \cdot \frac{1}{2 \cdot k + 1} \cdot x^{k \cdot 2 + 1}$$

The $h_n(x)$ function can be computed using the Horner scheme by mutual recursive lower and upper bound functions. Since the input value to h_n is squared, x is always non-negative.

lb-arctan-horner prec n 1 x computes the lower bound and *ub-arctan-horner prec n 1 x* the upper bound of $h_n(x)$ to the precision *prec*:

Definition *ub-arctan-horner* :: *nat* ⇒ *nat* ⇒ *nat* ⇒ *float* ⇒ *float*:

```
ub-arctan-horner prec 0 k x =  0
ub-arctan-horner prec (n + 1) k x =
  rapprox-rat prec 1 k − x · lb-arctan-horner prec n (k + 2) x
```

Definition *lb-arctan-horner* :: *nat* ⇒ *nat* ⇒ *nat* ⇒ *float* ⇒ *float*:

```
lb-arctan-horner prec 0 k x =  0
lb-arctan-horner prec (n + 1) k x =
  lapprox-rat prec 1 k − x · ub-arctan-horner prec n (k + 2) x
```

Lemma ARCTAN-0-1-BOUNDS**:**

If $0 \leq x$ *and* $x \leq 1$ *then*
$\mathit{arctan}\ x \in \{x \cdot \mathit{lb\text{-}arctan\text{-}horner\ prec}\ (\mathit{get\text{-}even}\ n)\ 1\ (x \cdot x)\ ..\ x \cdot \mathit{ub\text{-}arctan\text{-}horner\ prec}\ (\mathit{get\text{-}odd}\ n)\ 1\ (x \cdot x)\}$.

To approximate $\arctan$ for values outside of the power series's convergence area, we must calculate π. An easy-to-prove and reasonably fast approach is to use Machin's formula:

Lemma MACHIN**:** $\frac{pi}{4} = 4 \cdot \mathit{arctan}\ \frac{1}{5} - \mathit{arctan}\ \frac{1}{239}$

Since MACHIN only needs $\arctan$ in the interval $\{-1\ ..\ 1\}$, we can use it together with *lb-arctan-horner* and *ub-arctan-horner* to calculate π. To determine the number of terms needed to obtain the required precision, we observe that $(1/5)^2 < 2^{-4}$ and $(1/239)^2 < 2^{-14}$; hence, with each computed term we gain 4 bits for $\arctan(1/5)$ and 14 bits for $\arctan(1/239)$. Therefore, it suffices to consider *prec div 4 + 1* and *prec div 14 + 1* terms, respectively:

Definition *ub-pi* :: *nat* ⇒ *float*:

```
ub-pi prec =
  let A = rapprox-rat prec 1 5; B = lapprox-rat prec 1 239
  in 4 · (4 · A · ub-arctan-horner prec (get-odd (prec div 4 + 1)) 1 (A · A) −
          B · lb-arctan-horner prec (get-even (prec div 14 + 1)) 1 (B · B))
```

Definition *lb-pi* :: *nat* ⇒ *float*:

```
lb-pi prec =
  let A = lapprox-rat prec 1 5; B = rapprox-rat prec 1 239
  in 4 · (4 · A · lb-arctan-horner prec (get-even (prec div 4 + 1)) 1 (A · A) −
          B · ub-arctan-horner prec (get-odd (prec div 14 + 1)) 1 (B · B))
```

Theorem BNDS-PI**:** *pi* ∈ {*lb-pi n* .. *ub-pi n*}

Our implementation of $\arctan x$ only caters for $x \in \{-1\ ..\ 1\}$. Furthermore, since the coefficients of the Taylor series only shrink linearly, the desired precision is honored only for $x \in \{-1/2\ ..\ 1/2\}$. To compute values outside of that interval, we exploit the following properties of $\arctan$:

Lemma ARCTAN-HALF**:** $\mathit{arctan}\ x = 2 \cdot \mathit{arctan}\ \frac{x}{1 + \mathit{sqrt}\ (1 + x^2)}$

Lemma ARCTAN-INVERSE**:** *If* $x \neq 0$ *then*
$\mathit{arctan}\ \frac{1}{x} = \frac{\mathit{sgn}\ x \cdot \mathit{pi}}{2} - \mathit{arctan}\ x$.

Putting all ingredients together, we obtain the following case distinction:

$$\arctan(x) = \begin{cases} -\arctan(-x) & \text{if } x < 0 \\ \displaystyle\sum_{n=0}^{\infty} (-1)^n \cdot \frac{1}{2 \cdot n + 1} \cdot x^{2 \cdot n + 1} & \text{if } 0 \leq x \leq \frac{1}{2} \\ \displaystyle 2 \cdot \arctan\left(\frac{x}{1 + \sqrt{1 + x^2}}\right) & \text{if } \frac{1}{2} < x \leq 2 \\ \displaystyle \frac{\pi}{2} - \arctan\left(\frac{1}{x}\right) & \text{otherwise} \end{cases}$$

From this equation, we derive the following lower-bound and upper-bound computation:

Definition *ub-arctan* :: *nat* ⇒ *float* ⇒ *float*:

```
ub-arctan prec x =
  let lb-horner =
        λx. x · lb-arctan-horner prec (get-even (prec div 4 + 1)) 1 (x · x);
      ub-horner =
        λx. x · ub-arctan-horner prec (get-odd (prec div 4 + 1)) 1 (x · x)
  in if x < 0 then − lb-arctan prec (− x)
     else if x ≤ 1/2 then ub-horner x
          else if x ≤ 2
               then let x' = float-divr prec x
                                (1 + the (lb-sqrt prec (1 + x · x)))
                    in if 1 < x' then ub-pi prec · 1/2 else 2 · ub-horner x'
               else ub-pi prec · 1/2 − lb-horner (float-divl prec 1 x)
```

Definition *lb-arctan* :: *nat* ⇒ *float* ⇒ *float*:

```
lb-arctan prec x =
  let ub-horner =
        λx. x · ub-arctan-horner prec (get-odd (prec div 4 + 1)) 1 (x · x);
      lb-horner =
        λx. x · lb-arctan-horner prec (get-even (prec div 4 + 1)) 1 (x · x)
  in if x < 0 then − ub-arctan prec (− x)
     else if x ≤ 1/2 then lb-horner x
          else if x ≤ 2
               then 2 · lb-horner
                      (float-divl prec x
                        (1 + the (ub-sqrt prec (1 + x · x))))
               else let x' = float-divr prec 1 x
                    in if 1 < x' then 0 else lb-pi prec · 1/2 − ub-horner x'
```

For $x < 0$, we exploit the identity $\arctan x = -\arctan -x$, which swaps the bounds. For $0 \leq x \leq 1/2$, we directly call our power series implementation of $\arctan$. For $1/2 < x \leq 2$, we invoke the power series implementation with $x' = x/\left(1 + \sqrt{1 + x^2}\right)$. Mathematically, $x' \leq 1$, but because *float-divr* over-approximates division, we can obtain $x' > 1$. In that case, we simply use the mathematical upper bound $\pi/2$ as the upper bound. Finally, for $x > 2$, we exploit the identity $\arctan x = (\pi/2) - \arctan(1/x)$.

These functions compute correct $\arctan x$ bounds for all values of x:

Theorem BNDS-ARCTAN**:**

If $(l, u) =$ (*lb-arctan prec lx*, *ub-arctan prec ux*) $\wedge\ x \in \{lx\ ..\ ux\}$
then $l \leq \arctan x \wedge \arctan x \leq u$.

3.8 Evaluation Using the Code Generator

To evaluate the lower-bound and upper-bound functions, it could be tempting to use the Isabelle simplifier, which rewrites terms by applying equations in a left-to-right manner. However, this is slow and might even fail due to technical details of the implementation of numerals in Isabelle.

The alternative is to convert the functions to ML and execute them. Isabelle provides a facility to generate executable code from HOL specifications [Haftmann and Nipkow 2007]. The code generator supports a large fragment of HOL that includes datatypes, inductive predicates, and recursive functions. It turns out that our entire development fits comfortably within that fragment.

The code generator lets us map HOL constants directly to ML functions, bypassing their definition. This must be done with great care, since there is no proof of equivalence between the HOL constant's definition and the ML function. Our setup maps natural numbers and integers to ML's arbitrary-precision integers. For large integers (which arise when representing high-precision floating-point numbers in the format $m \cdot 2^e$), this provides a huge speedup, especially for the *div* and *mod* operators.

The code generator can be used as an oracle, which proves executable theorems by evaluating them in ML. Used in conjunction with reification, it enables us to develop a decision procedure in Isabelle/HOL and execute it in ML. Since the code generator circumvents Isabelle's LCF kernel, we must explicitly trust the code generator, the ML environment, and our mapping from HOL integers and natural numbers to ML arbitrary-precision integers.

This last point must be emphasized. Since Isabelle is implemented in ML, we already trust large parts of the ML environment. However, normal Isabelle applications do not require integers larger than 2^{31}, which typical ML implementations represent differently. We cannot exclude the possibility that there is a bug in, say, the ML implementation of division on large integers that does not affect the LCF kernel but that nonetheless compromises the correctness of our decision procedure.

4. Related Work

To our knowledge, the first proof method for approximating transcendental functions in an interactive theorem prover is [Harrison 1996]. The method relied on exact arithmetic to calculate transcendental functions in the HOL theorem prover. In exact arithmetic, each real value x is represented by a function $f_x(p)$ that computes an integer representing the first p decimal places of the exact real value. By definition, $f_x(p)$ always satisfies $|f_x(p) - 2^p x| < 1$, so no interval arithmetic is needed. In addition, all computations are evaluated in the logic without code generation, with the benefit that all theorems are completely proved using the LCF kernel.

For the PVS theorem prover, [Muñoz and Lester 2005] implements the *numerical* proof method using algorithms very similar to those used in our work. They also rely on code generation to efficiently compute the results. However, they use rational numbers instead of floating-point numbers. [Daumas et al. 2009] combine this approach with [Daumas et al. 2005] to implement interval splitting and Taylor series expansion (described in section 5).

[Melquiond 2008] applied essentially the same technique for Coq, but using floating-point numbers. Melquiond uses various techniques to expand the class of theorem that can be proved using his method, notably interval splitting.

[Akbarpour and Paulson 2008] followed a radically different approach to the whole problem. They transform the formula involving transcendental functions and square root into a formula consisting only of basic arithmetic operations on polynomials. This formula is then passed to a first-order prover, instead of proved by computation. Unlike the other approaches, this approach supports formulas like $|\exp(x) - 1| \leq \exp(|x|) - 1$, where x ranges over all real numbers. On the other hand, it is inefficient for computing high-precision numbers.

5. Extensions

The techniques described in this section are not implemented in Isabelle2009 but are included in the latest development version of Isabelle.

5.1 Interval Splitting

Often, a formula $\varphi(x)$ could be solved if the interval associated with the variable x were split into smaller intervals. This is typically due to the *dependency effect*. For example, if x lies in the interval $\{0\ ..\ 1\}$, the result of $x^2 - x$ using a single interval is $\{-1\ ..\ 1\}$. In contrast, if we split the interval into $\{0\ ..\ 0.5\}$ and $\{0.5\ ..\ 1\}$, we get $\{-0.5\ ..\ 0.25\} \cup \{-0.75\ ..\ 0.5\} = \{-0.75\ ..\ 0.5\}$.

In general, interval splitting [Daumas et al. 2009, Melquiond 2008] works as follows: For an input interval I associated with a variable x, we generate a set of subintervals $\{I_1, \ldots, I_n\}$ covering I. The proof method then verifies the formula for each subinterval I_i. If each formula can be proved, we know that $\forall i.\, x \in I_i \implies \varphi(x)$, hence $x \in I \implies \varphi(x)$. The proof method must be extended to accept a list of variables and associated subinterval counts, from which it automatically generates the desired theorems and proves them.

5.2 Taylor Series Expansion

We can often mitigate the dependency effect by rewriting the term so that most variables occur only once. When this does not work (e.g., for formulas containing trigonometric functions), an alternative is to use Taylor series expansion. We assume the formula has the form $\forall x \in I.\, f(x) > 0$. If f is at least n times differentiable on I and for a $c \in I$, we know from Taylor's theorem that

$$\exists t \in I.\, f(x) = \sum_{i=0}^{n-1} \frac{1}{i!} \cdot f^i(c) \cdot (x-c)^i + \frac{1}{n!} \cdot f^n(t) \cdot (x-c)^n$$

If we use I as the interval representing t, we know that $f(x)$ is between the lower and upper bound of the right-hand side. In each term $\frac{1}{i!} \cdot f^i(c) \cdot (x-c)^i$, the variable x occurs only in the exponentiation. As a result, x occurs exactly once per exponent, except for the last term $\frac{1}{n!} \cdot f^n(t) \cdot (x-c)^n$, since the interval I represents x and also t. Fortunately, if the nth derivation of f is small enough, the dependency effect is virtually eliminated.

5.3 Argument Reduction

In the current implementation, the precision for $\sin x$ and $\cos x$ is only guaranteed for input values in $\{-\pi/2 \mathinner{..} \pi/2\}$ and $\{-\pi \mathinner{..} \pi\}$, respectively. For other inputs, the pessimal interval $\{-1 \mathinner{..} 1\}$ is returned. The Taylor series return correct results for all input values, but to obtain the desired precision we must compute more terms as the absolute value of x increases. A better approach would be to use argument reduction. Since $\sin$ and $\cos$ are periodic functions, we may shift any real x by π times an integer k:

$$x' = x - k \cdot \pi \Longrightarrow \sin x' = \sin x$$

To compute k, we can simply divide x by π and cut of all digits after the decimal point. For interval arithmetic, we must choose a single k for both the lower and the upper bound. If the difference between them is greater than π, we can immediately return $\{-1 \mathinner{..} 1\}$. Otherwise, we must determine whether the two bounds are in the same monotonic interval of the function or there is an optimum between the two points.

References

Behzad Akbarpour and Lawrence C. Paulson. MetiTarski: an automatic prover for the elementary functions. In Serge Autexier, John Campbell, Julio Rubio, Volker Sorge, Masakazu Suzuki, and Freek Wiedijk, editors, *AISC/MKM/Calculemus*, volume 5144 of *Lecture Notes in Computer Science*, pages 217–231. Springer, 2008. ISBN 978-3-540-85109-7. URL `http://dblp.uni-trier.de/db/conf/aisc/aisc2008.html#AkbarpourP08`.

Amine Chaieb. *Automated methods for formal proofs in simple arithmetics and algebra*. PhD thesis, Technische Universität München, Germany, April 2008. URL `http://www4.in.tum.de/~chaieb/pubs/pdf/diss.pdf`.

Marc Daumas, Guillaume Melquiond, and César Muñoz. Guaranteed proofs using interval arithmetic. In *Proceedings of the 17th IEEE Symposium on Computer Arithmetic, ARITH-17*, pages 188–195, Cape Cod, Massachusetts, 2005. IEEE Computer Society.

Marc Daumas, David R. Lester, and César Muñoz. Verified real number calculations: A library for interval arithmetic. *IEEE Transactions on Computers*, 58(2):226–237, 2009. ISSN 0018-9340. doi: http://doi.ieeecomputersociety.org/10.1109/TC.2008.213.

Florian Haftmann and Tobias Nipkow. A code generator framework for Isabelle/HOL. Technical Report 364/07, Department of Computer Science, University of Kaiserslautern, August 2007.

John Harrison. *Theorem Proving with the Real Numbers*. PhD thesis, University of Cambridge, 1996.

John Harrison and Laurent Théry. Extending the HOL theorem prover with a computer algebra system to reason about the reals. In Jeffrey J. Joyce and Carl Seger, editors, *Proceedings of the 1993 International Workshop on the HOL theorem proving system and its applications*, volume 780 of *Lecture Notes in Computer Science*, pages 174–184, UBC, Vancouver, Canada, 1993. Springer-Verlag.

Johannes Hölzl. Proving real-valued inequalities by computation in Isabelle/HOL. Diploma thesis, Institut für Informatik, Technische Universität München, 2009.

David R. Lester and Paul Gowland. Using PVS to validate the algorithms of an exact arithmetic. *Theoretical Computer Science*, 291:203–218, January 2003.

Guillaume Melquiond. Proving bounds on real-valued functions with computations. In Alessandro Armando, Peter Baumgartner, and Gilles Dowek, editors, *Proceedings of the 4th International Joint Conference on Automated Reasoning*, volume 5195 of *Lectures Notes in Artificial Intelligence*, pages 2–17, Sydney, Australia, 2008.

J Strother Moore and Robert S. Boyer, editors. *Metafunctions: Proving Them Correct and Using Them Efficiently as New Proof Procedures*, chapter 3, pages 103–213. Academic Press, London, 1981.

César Muñoz and David R. Lester. Real number calculations and theorem proving. In J. Hurd and T. Melham, editors, *Proceedings of the 18th International Conference on Theorem Proving in Higher Order Logics, TPHOLs 2005*, volume 3603 of *Lecture Notes in Computer Science*, pages 195–210, Oxford, UK, 2005. Springer-Verlag.

Steven Obua. *Flyspeck II: The Basic Linear Programs*. PhD thesis, Technische Universität München, Germany, 2008. URL `http://www4.in.tum.de/~obua/phd/thesis.pdf`.