Fakultät für Informatik
der Technischen Universität München

Diplomarbeit in Informatik

Proving Real-Valued Inequalities by Computation in Isabelle/HOL

Johannes Hölzl

FAKULTÄT FÜR INFORMATIK
DER TECHNISCHEN UNIVERSITÄT MÜNCHEN

Diplomarbeit in Informatik

Proving Real-Valued Inequalities by Computation in Isabelle/HOL

Beweisen Reelwertiger Ungleichungen mit Berechnung in Isabelle/HOL

Supervisor	:	Prof. Tobias Nipkow, Ph.D.
Advisor	:	Prof. Tobias Nipkow, Ph.D.
Advisor	:	Dr. Amine Chaieb
Submission Date	:	April 22, 2009

I assure the single-handed composition of this diploma thesis only supported by declared resources.

Munich, April 22, 2009

(Johannes Hölzl)

Abstract

In this thesis we present an automatic proof method on real valued formulas. It translates the formulas into interval arithmetic calculations on floating point numbers. The resulting formulas are then evaluated by utilizing the code generator. These computations are entirely verified in Isabelle/HOL itself.

To reach that goal, we extend the theory with several missing analytical results about trigonometrical functions, as well as derivation rules for power series. A major new development are the boundary computations for $\sqrt{}$, π, sin, cos, arctan, exp and ln. Finally the correctness of these computations is verified in Isabelle/HOL.

Acknowledgements

I am very grateful to Tobias Nipkow, for introducing me to the field of theorem proving, and for all his advice and support. I also want to thank Amine Chaieb for his advice and for reading draft versions of my thesis. A big thank-you goes to Lukas Bulwahn for giving me lots of advice after reading a draft version and for the interesting cocoa breaks. Florian Haftmann helped me a lot with advice to his code generator framework. Thanks are due to the entire Isabelle group at TU München for advice and interesting lunch breaks. A special thanks goes to Brigitta Strigl for proof-reading my thesis.

Contents

Chapter 1

Introduction

1.1 Motivation

To verify that a computer system fulfills its specification, different techniques are used. The one most widely used is writing test cases and verifying that the system behaves correctly according to this tests. This inherently only verifies the correctness for a limited number of different input data. For complex systems deployed in critical environments this often is not enough.

Here are three famous errors that show how disastrous a failure in such systems can be:

- A software bug in the realtime clock of the Patriot missile defense deployed by the US military led to a small drift of the measured time. In the First Gulf War, this caused a long running system to miss an Iraqi SCUD missile, which killed 28 people [3].
- The Ariane 5 rocket exploded at its first flight due to an overflow in a variable storing the horizontal velocity. One of the controlling computers on board switched itself off. This caused self destruction by the supervising computer [15].
- Some Intel Pentium processors had a bug in the `FDIV` instruction, so that the result had a much worse precision than anticipated. Since the discovery of the bug Intel guarantees to replace each Pentium processor with that bug up to today [22].

The last two failures are each estimated to have cost around USD 500 million [11]. Even worse, Thomas Hales [8] assumes that:

> On average, a programmer introduces 1.5 bugs per line while typing. Most are typing errors that are spotted at once. About one bug per hundred lines of computer code ships to market without detection.

To lower this bug rate and to avoid such disastrous bugs, formal verification among other methods is used. Interactive theorem provers are used to assist the user in formal verification and to mechanically check the proofs done by the user.

Such provers are not only used for formal verification, but also to formalize mathematical proofs themselves. These formalizations are again used to verify programs. One important aspect here is the real analysis. In all the bugs above, verification in a system with a theory of real analysis would have been helpful to discover them. Fortunately, theorem provers like Isabelle/HOL already provide a comprehensive library of formalization of real analysis.

However, especially in software verification it is not enough to only have generic theorems about analysis available. Often explicit calculation is needed. Daumas [5] cites the following formula used in the verification of a flight control system:

$$\frac{3 \cdot \pi}{180} \leq \frac{g}{v} \cdot \tan\left(\frac{35 \cdot \pi}{180}\right) \leq \frac{3.1 \cdot \pi}{180} \tag{1.1}$$

where v is the velocity of the aircraft and g is the gravitational force. We only have constants so the inequality can be computed.

Other examples include the verification of precomputed number tables used in arithmetic libraries. Here the actual algorithms are often easy to verify, but without guaranteing correctness of the tables used the verification is incomplete.

The difficulty with these problems is that calculation on large numbers is necessary. Performing this calculations in small deductive steps with a theorem prover is possible but very slow, and often a special proof method is needed. Fortunately we can do the calculations in the external ML runtime environment, and import the results back into our theorem prover. We can prove the correctness of our calculation even better by verifying the correctness of our computation functions.

1.2 Contributions

The main contributions of the work presented in this thesis are as follows:

1. Formalizing upper and lower bound approximations of the most important elementary transcendental functions (sin, cos, arctan, exp, and ln) in Isabelle/HOL. We also implement approximation functions for the square root and the power to a natural number. The precision of this approximation is preset by the user.

2. Providing an evaluation function to approximate the result of a real arithmetic expression as interval, bounding the exact result.

3. Providing a tactic usable in Isabelle/Isar to automatically verify real inequality with bounded variables.

4. Extending the analytical results available in Isabelle/HOL. The most important parts here are the theorems equating the arc tangent and the logarithm with the respective Taylor expansion.

1.3 Overview

Chapter 2 will give an overview of the basics needed to understand this work. It gives a brief intro to Isabelle/HOL and interval arithmetic.

Chapter 3 discusses the decisions taken in this work and shows some alternative approaches.

Chapter 4 shows how the different functions were specified and which theorems needed to be proved. All the formulas in this chapter are checked by Isabelle/HOL.

Chapter 5 gives a timing analysis, an overview of related work from other authors and, in which way this work can be extended in the future.

1.4 Introductory Example

In this section we give a small overview of the implemented proof method. We go through each step performed by it. For one we show the executed proof method and the resulting subgoals.

Example ARCTAN15***:***

To show that $\arctan 1.5$ is less than one we could also use $\arctan \frac{\pi}{2} = 1$ and apply the monotonicity of arcus tangent. But then $2 \cdot 1.5 < \pi$ would be needed and we need to write down the entire proof. But when the user utilizes the `approximation` method the proof is only one step.

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

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

Reification: The proof method uses functions implemented in Isabelle/HOL itself. These functions operate on HOL data structures itself. So our first step is to transform the formula into data structures. Here *uneq* is the interpretation function. The rules in *uneq-equations* define the semantic of our arithmetic. They map the formula onto the data structure.

apply (*reify uneq-equations*)

  1. uneq (*Less* (*Arctan* (*Mult* (*Num 15*) (*Inverse* (*Num 10*)))) (*Num 1*)) []

Rewrite as approximation: As *uneq* maps the data structure directly to operations on real numbers it is not possible to execute it. Hence as second step the goal is rewritten to use the approximation function *uneq′*. We use UNEQ_APPROX: ⟦*bounded-by vs bs*; *uneq′ prec eq bs*⟧ ⟹ *uneq eq vs* to rewrite the formula.

UNEQ_APPROX is the central theorem in this thesis. It states the correctness of our approximation. By applying it to the reified formula we get a call to *uneq′*. It computes the formula using interval arithmetic on floating point numbers. When the inequality on these intervals holds it also holds on real numbers.

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

  1. bounded-by [] []
  2. uneq′ 10 (*Less* (*Arctan* (*Mult* (*Num 15*) (*Inverse* (*Num 10*)))) (*Num 1*)) []

Proof boundaries: When variables are used in the formula boundaries of these variables are required. In this example no variables are used hence the list of boundaries is empty and it is trivially solved.

apply *simp*

  1. uneq′ 10 (*Less* (*Arctan* (*Mult* (*Num 15*) (*Inverse* (*Num 10*)))) (*Num 1*)) []

Evaluate: The *uneq′* function is computeable, e.g. it can transformed into a ML program by the code generator and executed. When the result of the computation is true the theorem is stated. Otherwise it can not be stated using the given precision. Probably it is not possible to verify it using approximation and another method is necessary.

apply *eval*
done

All these steps are now combined into the `approximation` proof method. Its parameter specifies the precision used by the computations.

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

Chapter 2

Preliminaries

2.1 Isabelle

To verify mathematical proofs or properties about computer programs, more and more often theorem provers are used [8].

Isabelle [20] is such an interactive theorem prover. Isabelle implements the LCF approach to theorem proving. It forces each proof to be done step by step in a simple deductive system. These steps are performed by a very small kernel, the only code base which needs to be trusted. Of course, this code base is extended to the entire ML runtime environment, i.e. the compiler and libraries as well as the operating system on which the Isabelle system runs.

2.1.1 Proof methods

Isabelle provides a rich set of functions to prove theorems, called *proof methods*. On the ML level the proof methods are called *tactics*. The simplest ones just call the kernel to perform one inference rule. Since it would be a very tedious task for the user to manually perform each of these steps, *tacticals* are used to combine simple tactics to bigger and more complex tactics. The tactics provided by Isabelle range from simple rule application over substitution to algebraic solvers. The users can also write their own ML functions using these tactics, utilizing tacticals to combine tactics into a new proof method.

One of the most important proof method is the *simplifier*. It rewrites theorems with provided equations until the theorem is trivial or no further simplification rules are available. However, it is also possible for a tactic to specify a set of simplification rules to be used by the simplifier. This is very important when a special class of formulas needs to be solved or at least simplified.

There also exist special solvers for HOL formulas of a specific syntactic format, e.g. Presburger arithmetic or polynomial equations. Those solvers can be externel programs not generating proofs. To incorporate such solvers the LCF kernel needs to be circumvented, allowing proof methods to testify theorems without deducing them step for step in Isabelle. A proof method using such an external tool is called an *oracle*. An example is the evaluation of theorems by code generation described later in this chapter.

2.1.2 Isabelle/Isar

An Isabelle theory file distinguishes between the outer and the inner syntax. The outer syntax describes the commands that tell the Isabelle system what to do next. This language is called

Isabelle/Isar [24]. The inner syntax is the term language in which the definitions and theorems are specified.

Isabelle/Isar provides commands to define new constants, recursive functions, inductive predicates and types. Isar also provides a language for mathematical proofs. There are two styles available in which proofs can be written. The user can write a *goal-oriented* proof by applying tactics to its goal state. This is similar to the ML level, where tactics connected by tacticals are used to discharge theorems. Such proofs are not very readable and hard to maintain.

The main goal of the Isabelle/Isar language is to provide means to write structured mathematical proofs, similar to textbook proofs. To do this Isar offers a rich set of commands. Here the user can write down proofs in a very clear way.

The theory files developed for this thesis are entirely written using the structural language. Especially when formalizing proofs from mathematical textbooks this has the advantage to follow closely the structure of the textbook proof. Therefore proofs are easier to grasp.

2.2 Isabelle/HOL

Unlike other theorem provers like Coq, PVS, or HOL Light, which only support one object logic, Isabelle provides a framework to support several logics on top of its meta-logic. The most important object logic in Isabelle is HOL (Higher Order Logic). It is also used as the formalization logic in this thesis. HOL supports most concepts found in functional programming languages. In this section we describe the most important notations used in this thesis.

2.2.1 Terms and Types

Isabelle/HOL has a polymorphic type system with type classes.

Types are constructed from the following elements:

Base types: Base types like *bool*.

Type constructors: Polymorphic types like $'a\ list$, $'a \times 'b$ or $'a\ set$.

Type variables: Type variables can be instantiated with arbitrary types. Name of type variables start with a prime.

Function types: Special type constructors which represent functions. They are written with a double dashed arrow $'a \Rightarrow 'b$. There is also a special form when a function returns an option (i.e. $'a \Rightarrow 'b\ option$). The *option* type constructor is hidden and only the top half of the last arrow is shown: $'a \rightharpoonup 'b$.

For defining constants, predicates, functions and stating theorems terms are used. Terms are written in the usual λ-calculus style. The following additional expressions are used in this thesis:

- *if* A *then* B *else* C
- $case\ x\ of\ A \Rightarrow \ldots \mid B\ y\ z \Rightarrow \ldots$
- $let\ x = \ldots\ ;\ y = \ldots\ in\ \ldots$

2.2.2 Functions

Functions in Isabelle/HOL need to be defined totally. Isabelle/HOL supports the definition of recursive functions, similar to the definition of functions in a functional programming language [14]. However, since all functions need to be total, termination and completeness must be proved. The function package provides automation for these tasks.

Most function definitions in this thesis use simple recursion patterns on the structure of the input values. Hence termination and completeness of such functions is trivial. In a few cases we require a case analysis on the input value, which is not automatically but can often be proved in a couple of lines.

2.2.3 Data types

One easy way to define new types in Isabelle is to use the `datatype` command [2]. This is similar to defining new data types in Haskell or ML. A new type is defined by enumerating all constructors and their argument types. The `datatype` command automatically introduces among other things new symbols for the constructors and new rules for induction and case distinction over the new type. It is also possible to use the *case* expression for destruction of datatypes.

There are a lot of data types already available[1]:

bool represents boolean values. The constants True and False as well as the if operator *if* A *then* B *else* C are already available. There are also the usual logical operators like $A \wedge B$, $A \vee B$ and $\neg\ A$ available.

$'a \times 'b$ represents a pair of values. There are the usual functions like *fst* and *snd* available to retrieve the first and second value, respectively.

$'a$ *list* represents a list of values. HOL has a very powerful theory about lists. We use lists only to represent a list of variables for the arithmetic. List literals are either written as literals (i.e. $[1,\ 2,\ 3]$) or as construction of a head and a tail *head # tail* and the empty list []. It is also possible to access a list element at a specific index with $lst_{[index]}$.

$'a$ *option* represents an optional value. This is usually used to represent the result of a calculation which can fail. When the calculation has no result or is undefined, *None* is returned, otherwise the value x is returned as $\lfloor x \rfloor$. When it is sure that a calculation yields a valid result *the* is used. It returns the enclosed value *the* $\lfloor X \rfloor = X$.

2.2.4 Sets

An important concept in mathematics are sets. The type of a set is represented by $'a$ *set*. It represents a set over values of type $'a$. Internally it is just a function $'a \Rightarrow bool$. Unlike sets in programming languages the sets in Isabelle/HOL can be of infinite cardinality. Hence it is not possible to use them for code generation.

For sets the usual operators are available, like $A \cup B$, $A \cap B$ and $a \in A$. There are also set literals available, written as $\{a,\ b,\ c\}$. To define intervals $\{a\ ..\ b\}$ is used, here the type $'a$ needs to be ordered.

[1] *bool* and $'a \times 'b$ are not introduced using the `datatype` command. But they can be used just like normal datatypes.

2.2.5 Numbers

Isabelle/HOL provides *nat*, *int*, and *real* number types. Most arithmetic operators are implemented as type classes, there is nearly no difference in writing formulas about natural numbers, integers or real numbers.

The following operations available in HOL are used in this thesis:

- Ordering: $A < B$ and $A \leq B$
- Basic arithmetic operations: $A + B$, $A - B$ and $A \cdot B$. For *int* and *real* also $- A$.
- Power function: A^n Here n is a natural number, i.e. the inverse of A is not needed.
- A *div* B and A *mod* B is the result and remainder of the integer division on *nat* and *int*.
- $\frac{A}{B}$ defines the division on the real numbers.
- Summation $\sum\limits_{i=n}^{m} f\ i$: Calculates the finite sum $f(n) + f(n+1) + \cdots + f(m)$
- Conversion functions between numbers: *real x*, *int x*, and *nat x*. This needs to be written in the Isabelle/HOL syntax. The type checker is unable to infere them automatically, but for clarity they are omitted in the formulas presented in this thesis.

2.2.6 Real Analysis

The fundamentals of mathematical analysis are also already formalized in Isabelle/HOL [6]. In this section we describe the parts of the formalized analysis used in this thesis.

Sequences are formalized as functions from natural numbers to real numbers: $nat \Rightarrow real$. We use $a \longrightarrow_\infty x$ to describe that a sequence a converges against x. Sometimes we need to state that the sequence a is monotone, e.g. either each member of the sequence is lesser or equal to all previous members, or each member of the sequence is greater or equal to all previous members. We denote monotonicity of a sequence a with *monoseq a*.

To state that x' is the derivation of f at x we write *DERIV* $f\ x :> x'$.

To formalize the trigonometric functions power series are needed. Series are just like sequences defined as ($nat \Rightarrow real$) functions. We write *summable a* to state that the series converges. $\sum\limits_{i=0}^{\infty} a\ i$ is the limit of the series a.

2.3 Reflection and Reification

To prove arithmetic formulas we utilize reflection. First the formula is reified, i.e. its transformed into a data structure evaluated by an interpretation function. The interpretation function is then replaced by an approximation. Finally the approximation is executed, the result states the correctness of the formula.

Example REIFICATION APPLIED TO A SMALL ARITHMETIC***:***
— The *Arith* data type specifies the syntactical form of our formulas:
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:

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

Now we implement a function in HOL itself to analyse the provided *Arith* data structure and prove the correctness of the analysis in Isabelle.

When performing reflection we first reify the term. Here the HOL term is transformed into a data structure of type δ applied to an interpretation function $\llbracket\ \_\ \rrbracket_{\_} :: \delta \Rightarrow \tau\ \mathit{list} \Rightarrow \tau$. Here τ is the type of the original formula. t is the original term, t' denotes the data structure representing, and xs is the variable assignment of the variables occuring in t.

Reflection is then performed in two steps:

Reification Deduces an equation $t = \llbracket\ t'\ \rrbracket_{xs}$. The rewrite rules for $\llbracket\ \_\ \rrbracket_{\_}$ are applied in reverse direction. Hence this equation can be proved by just using the rewrite rules.

Evaluation Proves $\llbracket\ t'\ \rrbracket_{xs}$ by evaluation. $\llbracket\ \_\ \rrbracket_{\_}$ is replaced by an approximation function. This function is then evaluated by code generation.

Chaieb [4] implemented a generic reification mechanism. To use this mechanism the user must provide rewrite equations for $\llbracket\ \_\ \rrbracket_{\_}$. They specify how the HOL terms are mapped to values of type δ. These rewrite equations are usually the definitional equations of $\llbracket\ \_\ \rrbracket_{\_}$. It is also possible to specify different interpretation functions $\llbracket\ \_\ \rrbracket_{\_}$ for different types δ. This allows to further restrict the syntactic structure of the original formula.

Here we describe a simplified version of how this generic reification mechanism is used. First the equations for $\llbracket\ \_\ \rrbracket_{\_}$ are provided:

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

Here x_i are variables, C_j are constructors of δ, and P_k are arbitrary terms. Each line specifies how each constructor is interpreted. It is important to note that there can be different equations with the same constructor on the left-hand side. The generic reification now tries to unify a P_k with the term to reify. When P_k matches the corresponding constructor, C_k is used to build up the resulting data structure. Now we repeat the previous step with the arguments to P_k. If the term to reify is only a variable x, the last equation is used, and x is inserted into the variable assignment xs.

Reflection was first used by Moore and Boyer [18] which they called Metafunctions. For a more detailed overview of reflection and reification see Chaieb [4].

2.4 Evaluation with code generation

Isabelle provides a facility to generate executable code from rewrite rules [7]. It generates code for Standard ML [17], Haskell [12] or Objective Caml. But these are not the only usable target languages; each language where reduction of pure terms are viewed as equational deduction [7] can be used as a target language.

The primary goal of the code generator is to automatically generate executable code from specifications. But there are large part of HOL which are not executable, henceforth a subset of HOL is specified as the executable elements of HOL specifications. This includes data types, inductive predicates and recursive functions. In this thesis only recursive functions and data types are used for code generation. When a new function is introduced using the `definition` or `fun` commands the rewrite rules used by the definition are added to the code generator. But it is also possible to overwrite this rules by other equations.

Moreover the code generator also allows the overriding of Isabelle constants by constants defined in the target language. The available setup maps natural numbers and integers to arbitrary precision integers in ML. This needs to be done with a lot of care, since there is no proof providing the equivalence between the Isabelle version and the ML version of a function. But it provides a significant speedup when used with large integers, especially when using the *div* and *mod* operators.

The code generation framework in Isabelle is not only used to generate executable specifications, but can also be used as a proof method. It provides an oracle which proves executable theorems by evaluating them in ML. Together with reification it is easy to develop an entire decision procedure in Isabelle/HOL and execute it on the ML level. On the one hand the LCF approach is lost, but on the other hand the confidence in the proof is established if we trust the code generator and the ML environment.

This last point needs to be emphasised. Since Isabelle itself is implemented in ML, we must already trust large parts of the ML environment. But especially the arbitrary precision integers used in this thesis are used in a way normally not utilized by Isabelle itself. For example we use the large numbers, i.e. numbers greater or equal to 2^{31}. They are differently represented than the smaller numbers. And we use the integer division on these large numbers.

2.5 Interval arithmetic

We want to prove inequality of real values. It is not possible to compute real values exact as there are uncountable many. Hence to computable represent a real value x we use two floating point numbers (l, u) where $l \leq x \leq u$. We distinguish between the computable interval (l, u) and the set of reals represented by them $\{l..u\}$[2]. The former is used in computations, the later is used in theorems and proofs.

For each computable function $\hat{f}$ which computes the boundaries of $f(x)$, we need to show that it satisfies:

$$\forall x \in \{l..u\}.\ (l', u') = \hat{f}(l, u) \longrightarrow f(x) \in \{l'..u'\} \tag{2.1}$$

For binary operations $x \oplus y$ the computable operator $x \mathbin{\hat{\oplus}} y$ needs to satisfy:

$$\forall x \in \{l_x..u_x\}.\forall\ y \in \{l_y..u_y\}.\ (l', u') = (l_x, u_x) \mathbin{\hat{\oplus}} (l_y, u_y) \longrightarrow x \oplus y \in \{l'..u'\} \tag{2.2}$$

Some functions are not defined on the entire real numbers, e.g. the logarithm which is only defined for positive values. Here we use the result type *float option* × *float option*, so that a

[2] Here we do not use the usual mathematical notation $[l, u]$ but the notation Isabelle specific with braces.

bounding function can return *None* as an undefined value. We could also use *(float × float) option*, which is inconvenient in our case, since most function have separate upper and lower bound calculations. Each function itself yields *None* when the input value is not in the domain.

Addition and subtraction is very easy to handle in interval arithmetic:

$$(l_a, u_a) + (l_b, u_b) = (l_a + l_b, u_a + u_b) \tag{2.3}$$

$$-(l, u) = (-u, -l) \tag{2.4}$$

Multiplication is more demanding. We need to consider the different cases of the signs of the upper and lower bounds of the operands. To avoid this, we introduce two auxiliary functions:

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

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

This auxiliary functions are called *float-pprt* and *float-nprt* in Isabelle/HOL. They allow us to write the upper and lower bounds of the product just as a sum of all cases:

$$\begin{array}{rcl}(l_a, u_a) \cdot (l_b, u_b) & = & (l_a^- \cdot u_b^+ + u_a^- \cdot u_b^- + l_a^+ \cdot l_b^+ + u_a^+ \cdot l_b^-, \\ & & \; u_a^+ \cdot u_b^+ + l_a^+ \cdot u_b^- + u_a^- \cdot l_b^+ + l_a^- \cdot l_b^-)\end{array} \tag{2.5}$$

The calculation of the inverse is easy again, no auxiliary functions are needed. The function only needs to check if zero is not in the input interval:

$$0 \notin \{l..u\} \qquad \longrightarrow \qquad \frac{1}{(l, u)} = \left(\frac{1}{u}, \frac{1}{l}\right) \tag{2.6}$$

Applying interval arithmetic on monotone functions works as expected. If the function f is increasing and monotone we have: $f(\,(l, u)\,) = (f(l), f(u))$. If the function is decreasing we need to exchange the bounds: $f(\,(l, u)\,) = (f(u), f(l))$. Most functions we want to compute are monotonic on the entire real numbers. We have a computable function $\overline{f}$ to get the upper bound and $\underline{f}$ to get the lower bound. As approximated interval we just use $(\underline{f}(l), \overline{f}(u))$.

The only non-monotonic functions we want to compute are the sine and cosine functions. Thus the cosine is only computed on $\{-\pi..0\}$ and $\{0..\pi\}$, the sine only on $\{-\frac{\pi}{2}..\frac{\pi}{2}\}$. Both functions are monotone in these intervals and thus (2.1) holds for them.

If we compute the result of a formula using interval arithmetic, we know that the exact real value is in the bounds of the interval. But to prove an inequality it is not enough to have upper and lower bounds of both sides. The two resulting intervals need to be disjunct. For example we want to show: $\sin(0) < \cos(0)$. Here the interval computation for $\sin(x)$ and $\cos(x)$ could always return $\{-1..1\}$ which is a valid result, but it can not be used to show the inequality.

Hence the computations need to obey a *precision* specified by the user. The precision states how many bits of a floating point number are correctly computed. When the user specifies the precision p and computes $f(x) = m \cdot 2^e$, with $0 \leq m \leq 1$ using $\overline{f(x)}$ and $\underline{f(x)}$ as bounds, the following assertions should hold:

$$\begin{array}{rcl}\overline{f(x)} - f(x) & \leq & 2^{e-p} \\ f(x) - \underline{f(x)} & \leq & 2^{e-p}\end{array}$$

Of course, this is not guaranteed for cos and sin outside of $\{-\pi..\pi\}$ and $\{-\frac{\pi}{2}..\frac{\pi}{2}\}$, respectively. This only holds for elementary functions, when the functions are combined to formulas this assertions does not hold anymore. The precision describes the magnitude of the error. Each function and operation also influences the error. Hence in the case of multiplication or exponential function the error grows very fast, e.g. the precision worsens. Fortunately in most cases it is easy for the user to guess the necessary precision applied to each operation to get the desired result.

2.6 Taylor series

To compute the upper and lower limits of the transcendental functions we use Taylor series. Luckily most of the transcendental functions are defined as Taylor series in Isabelle/HOL. For the right input domain of ln and arctan the equivalence to their Taylor series is shown. As series are infinite they are often not computable. But we only need upper and lower bounds up to a specific precision which is computeable.

In the following section we consider the Taylor series of a function f. Often the series is not defined on the entire real numbers, but only on an interval R_f:

$$x \in R_f \qquad \Longrightarrow \qquad f(x) = \sum_{i=0}^{\infty} (-1)^i \cdot \frac{1}{a_i} \cdot x^i \tag{2.7}$$

For each function f we consider a_i to be always a positive monotonic null sequence. To simplify it further we assume that x is in $\{0..1\}$. To compute the bounds we consider the *partial sums* S_n of the Taylor series:

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

From (2.7) we know that $\lim_{n\to\infty} S_n = f(x)$. Since a_i is a monotonic positive null sequence we know that $\{S_{2n+1} \,..\, S_{2n}\}$ forms a sequence of nested intervals. Since the limit of S_n is $f(x)$ we know that $f(x)$ is always in this intervals. Hence we know that for an even n S_n is a upper bound and for an odd n S_n is a lower bound of $f(x)$. Also the difference between the two partial sums is:

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

Now we can estimate the error of the upper bound: $S_{2n} - f(x) \le \frac{1}{a_{2n+1}} \cdot x^{2n+1}$. From this formula we can calculate n depending on the precision used. This is necessary as we use the Horner scheme to compute the partial sums. Here we need to know the length of the polynomial beforehand when calling the function evaluating the Horner scheme.

For example assume we can estimate $\frac{1}{a_{2n+1}} \le \frac{1}{2^{2n+1}}$. p is the precision without loss of generality we assume p is even. Now we show that S_p is a upper bound in the precision p:

Since p is even, we have a p' such that $p' = 2 \cdot p$. From $0 \le x \le 1$ we have a $1 \le m < 2$ and a $e \le 0$ such that $x = m \cdot 2^e$. Now we can estimate the upper bound:

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

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

Now we introduce $h_a(m, x)$ which computes S_m using the Horner Scheme. We define $b_n = a_{m-n-1}$. This is necessary as the index passed to h_b is reversed. In section 4.3 we will see how b_n is calculated.

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

If $h_b(m, x)$ is computed as floating point number, it is not possible to compute the coefficients $\frac{1}{b_n}$ exactly. Hence we need to split the computation function into two parts one for the lower bound and one for the upper bound. This is no problem, as we already compute the upper or lower bound of the Taylor series. The polynomials computed in the recursive call are subtracted from the current coefficient. As subtraction swaps the bounds we have the mutual recursive functions $\overline{h_b}$ and $\underline{h_b}$:

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

Unfortunately the domain R where the power series is convergent and forms nested intervals is only a small part of the functions entire domain. So it is necessary to do a case analysis on the input value and apply some transformations before doing the calculation using $f(x)$. As this case analysis and transformation depends on the function itself it is shown at the formalization of each function in section 4.4.

Chapter 3

Design considerations

In this chapter we give some justifications for the techniques used in this thesis.

3.1 Interval arithmetic

Instead of using intervals to represent the resulting number, another possible approach is to use a single value. This value can be represented as a rational number, floating point number or even as a function computing the bits of the real number. This approach has the advantage that each operator is evaluated only once. However, the computed value can never be the exact value; we always only have a result near it.

To have useful results when such an approach is used, an estimation of the error is needed. Hence for each elementary operation and transcendental function the precision must be specified and proved. When using interval arithmetic we just need to show that we compute some upper and lower bounds. Our proof method only fails if the needed precision is not reached.

Another disadvantage compared to interval arithmetic: We can not specify an arbitrary range for a variable. If we look at the equation (1.1) in the introduction, the variable v specifies the velocity of the aircraft. We probably do not want to specify an exact velocity, but some range in which the velocity of the aircraft is. It is the same with the gravity g where we do not even know the exact value. We can only measure it up to some precision.

3.2 Floating point numbers

The bounds of the intervals need a finite representation. Here we usually have three options: rational numbers, floating point numbers or exact arithmetic. In this thesis we use floating pointer numbers. A floating point number x is represented by two integers: the mantissa m and the exponent e.

$$x = m \cdot 2^e \qquad \text{Representation of floating point numbers}$$

A rational number x is represented by two integers: the numerator a and the denominator b.

$$x = \frac{a}{b} \qquad \text{Representation of rational numbers}$$

Rational numbers have the advantage that they are closed under all elementary arithmetic operators. Hence the result of formulas just using these operators are exactly computed. This is not possible with floating point numbers. They are not closed under division. It is not always possible to choose m and e in such a way that $a/b = m \cdot 2^e$. However, if a precision p is given, it is possible to compute an upper and lower bound of a/b within $a/b - 2^{-p}$ and $a/b + 2^{-p}$.

Why did we choose floating point numbers in spite of this problem? First, as the exact real values are always approximated by an interval with upper and lower bound this is no problem. Second, for transcendental functions to be computable we need to specify a precision (i.e. the amount of correct bits). This precision is not only used by the computations, but also to limit the length of the numbers. Here we save a lot of memory and speed up the computations. With floating point numbers this is easy to do. The mantissa is cut off at the specified length.

If we use rational numbers instead, the size of the resulting numbers grow very fast. A naïve implementation of addition multiplies with the divisors. The size of the new divisor doubles. Even reducing it afterwards to its simplest form is often not enough. This is the case when computing transcendental functions where the reciprocal of the factorial is used. Hence the divisor grows very fast. In such cases we need again to approximate an upper and lower bound, which is much more complicated than the rounding of floating point numbers.

The third approach is to use exact arithmetic. Here a number is represented as a function computing the real value up to a specified amount of digits. A number x in exact arithmetic is represented by the function $f_x(p)$. The following must hold:

$$|f_x(p) - x \cdot 2^p| < 1$$

Here we have the problem to show this theorem for each operation. Another problem is to handle very small or very large numbers. If floating point numbers are used this can be circumvented by using the exponent. This is not possible in exact arithmetic, as we would need to determine the equality of two real numbers in order to correctly implement subtraction.

3.3 Taylor series

We use the Taylor series of each transcendental function to approximate it. sin, cos and exp are defined as Taylor series in Isabelle/HOL. No further theorems are needed to use these functions. Other methods, such as continued fractions, would need additional theorems to be proved. With the use of the Horner scheme the Taylor series is very easy to implement on floating point numbers.

For the Taylor series it is easy to estimate when the precision is reached. It is not necessary to show difficult termination proofs or estimations of precision. Taylor series are always used in a range where they are alternating power series. Hence we can precompute the amount of necessary members until the specified precision is reached, as described in section 2.6.

An alternative to using Taylor series would be using the CORDIC (COordinate Rotation DIgital Computer) algorithm. This algorithm is often used to efficiently compute transcendental functions in software and hardware. It only needs a table, and for each precision bit an addition and a shift operation is executed. Harrison [10] outlines how this algorithm is implemented in HOL. Here the CORDIC algorithm is implemented and verified for the logarithm.

In spite of the popularity of the CORDIC algorithm we chose to not implement it. The first problem is to verify the table of precomputed values. This would require to verify the correctness of this table where we again need to compute the function values. This could be done using the Taylor series again. Another problem is the missing shift operation in ML on the integer type. We can use multiplication and division, but then we lose the speed gain of the CORDIC algorithm. And finally we can no longer compute the values up to arbitrary precision, as the table used by the algorithm needs an entry for each precision bit.

3.4 Horner scheme

All transcendental functions we want to compute are now represented as Taylor series. We can not compute the entire Taylor series directly, as it is infinite. For each function we can calculate the number of members which are needed to reach a specific precision. This finite polynomial is now computable. The functions computing the boundaries are implemented using the Horner scheme.

From the viewpoint of numerical stability there is no difference between using the Horner scheme and using the expanded form, as multiplication and addition are distributive, commutative and associative on our floating point numbers. Hence the results of both forms are equivalent. The advantage of using the Horner scheme is the performance increase. The amount of addition operations is the same; however, the amount of multiplication operations is greatly reduced. If we used a naïve implementation of the power operator, we would need $n^2 + n$ operations for the monomial form.

3.5 Reflection

When we prove theorems on numbers operations such as additions and multiplications are performed by using rewrite rules. Since the numbers are represented as binary numbers the time needed to perform additions is linear in the numbers of digits, and the time needed to perform multiplications is quadratic. Unfortunately, each such rewrite operation needs to be threaded through the Isabelle kernel to get confidence in the correctness of the operations.

An alternative way is to use reflection. Here we use the ML integer library to perform our computations. The speed-up is only a constant factor, but this factor is very large. The operations on the microprocessor operate on 32-bit (or 64-bit) integers in one instruction. The rewrite steps involve a large number of instructions including memory access to manage the tree representing the term.

Nevertheless the computation could be performed by rewriting and Harrison argues in favor of this approach [9]. Unfortunately, for this approach we need to implement our own decision procedure which passes the right rewrite rules to the Isabelle kernel to get a decent speed. To avoid this we used reflection, as introduced in [18]. This allows us to have a fast decision procedure whose only trade-off is to have the trusted code base extended by the code generator.

Chapter 4

Formalizations

In this chapter we give an overview of the Isabelle/HOL formalizations done in this thesis. We build on the real analysis theory developed by Fleuriot [6] and the floating point library developed by Obua [21]. We begin with describing the extensions developed for these theories. The main part covers the approximation and interpretation functions.

All **definition**s in this chapter are as such defined in Isabelle/HOL and hence are syntactically correct and type-checked. The termination is proven for each recursive function. All **lemma**s and **theorem**s presented here are proved with Isabelle/HOL.

4.1 Additional analytical theorems

For most transcendental functions we can use MacLaurin's lemmas to use their Taylor series. However, to compute upper and lower bounds of the arc tangent or the logarithm we need the following equations:

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

Lemma LN_SERIES**:** *If* $0 < x$ *and* $x < 2$ *then* $\mathit{ln}\ x = \sum_{n=0}^{\infty} (-1)^n \cdot \frac{1}{n+1} \cdot (x-1)^{n+1}$.

Formalizing these lemmas requires some analytical theorems not yet proven in Isabelle/HOL. The following theorems were shown by using textbook proofs given by Königsberger [13]. First we need the summability of these Taylor series. This is provided by the alternating series test[1]. When using this test we get additional conclusions showing the bounding property of the partial sums.

Lemma SUMMABLE-LEIBNIZ**:** If $a \longrightarrow_\infty 0$ and $\mathit{monoseq}\ a$, then:

1. $\mathit{summable}\ (\lambda n.\ (-1)^n \cdot a\ n)$

   We know that each alternating series whose coefficients either decrease or increase towards zero is convergent.

2. $0 < a\ 0 \longrightarrow (\forall\, n.\ \sum_{i=0}^{\infty} (-1)^i \cdot a\ i \in \{\sum_{i=0}^{2 \cdot n - 1} (-1)^i \cdot a\ i\ ..\ \sum_{i=0}^{2 \cdot n} (-1)^i \cdot a\ i\})$

3. $a\ 0 < 0 \longrightarrow (\forall\, n.\ \sum_{i=0}^{\infty} (-1)^i \cdot a\ i \in \{\sum_{i=0}^{2 \cdot n} (-1)^i \cdot a\ i\ ..\ \sum_{i=0}^{2 \cdot n - 1} (-1)^i \cdot a\ i\})$

   The exact result of the infinite sum is bounded by two consecutive partial sums.

[1] Called "Leibniz Kriterium" in german, hence the lemmas name.

Here the last two conclusions are very important for our goals. They are the main theorems used to show the upper and lower bounds of arc tangent and logarithm. Those two Taylor series are introduced by the geometric sum of the derivatives. Hence we need to show the differentiability of power series. Every power series can be differentiated on its convergence area. Unfortunately, we need the lemma $\lim_{n\to\infty} \sqrt[n]{n} = 1$ to show this. Instead of proving that, we introduce the following lemma which is simpler but equally useful for our purposes:

Lemma DERIV_POWER_SERIES'**:** If the following premises hold

1. $\bigwedge x.\ x \in \{-R{<}..{<}R\} \Longrightarrow \mathit{summable}\ (\lambda n.\ f\, n \cdot (n+1) \cdot x^n)$
2. $x_0 \in \{-R{<}..{<}R\}$

then $\mathit{DERIV}\ (\lambda x.\ \sum_{n=0}^{\infty} f\, n \cdot x^{n+1})\ x_0 :> \sum_{n=0}^{\infty} f\, n \cdot (n+1) \cdot {x_0}^n$.

These analytical results are now available in the regular HOL image.

4.2 Floating point arithmetic

The floating point numbers are formalized in the `ComputeFloat` theory developed by Obua for the Flyspeck II project [21]. It implements floating point numbers as a tuple of two integers. Integers are mapped to arbitrary precision integers in ML when compiled by the code generator. Hence operations on them are the native integer operations in ML.

Floating point numbers are introduced as a data type. They are represented as mantissa and exponent (here also called scale):

Definition float: *datatype float* = ***Float*** *int int*

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

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

To display the floating point values in a more readable way in this thesis, numbers of the form *Float 1 i* are printed as 2^i, e.g. $\mathit{Float}\ 1\ {-2} = \frac{1}{4}$.

In some functions we need to operate on the bit length of a number, e.g. when computing the size of the mantissa. Hence we introduce *bitlen*, which returns the number of bits needed to represent an integer. Instead of giving its definition we show the most important property for our purposes:

Lemma BITLEN-BOUNDS**:** *If* $0 < x$ *then* $2^{\mathit{bitlen}\ x - 1} \le x \wedge x < 2^{\mathit{bitlen}\ x}$.

In order to map the floating point numbers into the real numbers, we define the function *Ifloat*, which defines the mapping from *float* to *real*. Note that this mapping is not injective, i.e. $2{\cdot}2^0 = 1{\cdot}2^1$. As the power operator in HOL only allows natural numbers as exponents we use the *pow2* function, implementing the power of 2 to an integer. We use guillemot braces to denote this function:

Definition *pow2* :: *int* ⇒ *real*: $\mathit{pow2}\ e = (\mathsf{if}\ 0 \le e\ \mathsf{then}\ 2^e\ \mathsf{else}\ \mathit{inverse}\ 2^{-e})$

Definition *Ifloat* :: *float* ⇒ *real*: «*Float a b*» = $a \cdot \mathit{pow2}\ b$

To show the soundness of our floating point operations we need to show that «_» preserves the basic arithmetic operations on *real*:

Lemma IFLOAT_ADD, IFLOAT_MINUS, IFLOAT_SUB, IFLOAT_MULT, IFLOAT_POWER**:**
«$a + b$» = «a» + «b»
«$- a$» = $-$ «a»
«$a - b$» = «a» $-$ «b»
«$a \cdot b$» = «a» $\cdot$ «b»
«x^n» = «x»^{n}

All these operations are exact on floating point numbers, hence «_» is a homomorphism on all these operations. Their usage in proofs is very easy: to simplify a formula including «_», we can apply these equations on the term. After this simplification we get terms where «_» interprets only floating point variables or approximation functions. Hence $«a + b \cdot c - d» = «a» + «b» \cdot «c» - «d»$ is shown by the simplifier.

Unfortunately, this is not possible for the division operator. For example the value $\frac{1}{3}$ cannot be expressed as a floating point number. Instead of trying to compute the exact value we provide two functions, computing a lower and an upper bound. The user can specify the distance from the exact result to the computed result by specifying a precision.

To implement the division on floating point numbers we first compute the fractional part, e.g. divide the mantissas. In Obua's version the division for the fractional part was implemented by computing the bits iteratively until the specified precision was reached. In our implementation _ *div* _ is used as it directly compiles to the `div` operator in ML. The fractional part of two positive integers is computed by *lapprox-posrat* and *rapprox-posrat*:

Definition *lapprox-posrat* :: *nat* ⇒ *int* ⇒ *int* ⇒ *float*:

$\mathit{lapprox\text{-}posrat}\ \mathit{prec}\ x\ y = (\mathsf{let}\ l = \mathit{prec} + \mathit{bitlen}\ y - \mathit{bitlen}\ x;\ d = x \cdot 2^l\ \mathit{div}\ y\ \mathsf{in}\ \mathit{normfloat}\ (\mathit{Float}\ d\ (-\ l)))$

Definition *rapprox-posrat* :: *nat* ⇒ *int* ⇒ *int* ⇒ *float*:

$\mathit{rapprox\text{-}posrat}\ \mathit{prec}\ x\ y =$
$(\mathsf{let}\ l = \mathit{prec} + \mathit{bitlen}\ y - \mathit{bitlen}\ x;\ X = x \cdot 2^l;\ d = X\ \mathit{div}\ y;\ m = X\ \mathit{mod}\ y$
$\ \mathsf{in}\ \mathit{normfloat}\ (\mathit{Float}\ (d + (\mathsf{if}\ m = \mathit{0}\ \mathsf{then}\ \mathit{0}\ \mathsf{else}\ \mathit{1}))\ (-\ l)))$

We also introduce *rapprox-rat* and *lapprox-rat*, which decide on the sign of both operands which approximation to choose. Using them, *float-divl* and *float-divr* finally implement the division on floating point numbers.

Definition *float-divl* :: *nat* ⇒ *float* ⇒ *float* ⇒ *float*:

$\mathit{float\text{-}divl}\ p\ (\mathit{Float}\ m_1\ s_1)\ (\mathit{Float}\ m_2\ s_2) = \mathit{Float}\ \mathit{1}\ (s_1 - s_2) \cdot \mathit{lapprox\text{-}rat}\ p\ m_1\ m_2$

Definition *float-divr* :: *nat* ⇒ *float* ⇒ *float* ⇒ *float*:

$\mathit{float\text{-}divr}\ p\ (\mathit{Float}\ m_1\ s_1)\ (\mathit{Float}\ m_2\ s_2) = \mathit{Float}\ \mathit{1}\ (s_1 - s_2) \cdot \mathit{rapprox\text{-}rat}\ p\ m_1\ m_2$

Lemma FLOAT_DIVL**:** $«\mathit{float\text{-}divl}\ \mathit{prec}\ x\ y» \le \frac{«x»}{«y»}$
Lemma FLOAT_DIVR**:** $\frac{«x»}{«y»} \le «\mathit{float\text{-}divr}\ \mathit{prec}\ x\ y»$

One problem with the definition of the floating point numbers using arbitrary precision integers is the unbounded size of the mantissa. When multiplication is used the resulting bit length of the mantissa is the sum of the bit length of the operands. This is especially dangerous when the power operator is used, as this causes the bit length of the mantissa to get multiplied with the exponent. As the transcendental functions are computed using their Taylor series the resulting mantissa gets larger than the specified precision. However, this can also be the case for addition and subtraction. Here both numbers need to be aligned to have the same exponent. For example $1 \cdot 2^0 + 1 \cdot 2^{10} = 1025 \cdot 2^0$. Even when the mantissas of the operands are small the resulting mantissa depends on the exponents used. To avoid growth, the length of the numbers is cut back after each evaluation step. To do this we introduce the functions *round-up* and *round-down*:

Definition *round-up* :: *nat* ⇒ *float* ⇒ *float*:

$\mathit{round\text{-}up}\ \mathit{prec}\ (\mathit{Float}\ m\ e) =$
$(\mathsf{let}\ d = \mathit{bitlen}\ m - \mathit{prec}$
$\ \mathsf{in}\ \mathsf{if}\ \mathit{0} < d\ \mathsf{then}\ \mathsf{let}\ P = 2^d;\ n = m\ \mathit{div}\ P;\ r = m\ \mathit{mod}\ P\ \mathsf{in}\ \mathit{Float}\ (n + (\mathsf{if}\ r = \mathit{0}\ \mathsf{then}\ \mathit{0}\ \mathsf{else}\ \mathit{1}))\ (e + d)$
$\quad \mathsf{else}\ \mathit{Float}\ m\ e)$

Definition *round-down* :: *nat* ⇒ *float* ⇒ *float*:

$round\text{-}down\ prec\ (Float\ m\ e) =$
$(\mathsf{let}\ d = bitlen\ m - prec\ \mathsf{in}\ \mathsf{if}\ 0 < d\ \mathsf{then}\ \mathsf{let}\ P = 2^d;\ n = m\ div\ P\ \mathsf{in}\ Float\ n\ (e + d)\ \mathsf{else}\ Float\ m\ e)$

4.3 Horner scheme

The transcendental functions are approximated using an alternating power series computed with the Horner scheme. In the following lemma we assume that *ub* computes the upper bound and *lb* the lower bound. These functions implement the Horner scheme for a specific series. *f* is the sequence used as reciprocals of the coefficients. The calculation of *f* depends on the series itself.

F, *G*, and *s* describe the iterative calculation of *f*. *G i j* computes the next coefficient where *j* is the previous coefficient and *i* an auxiliary value computed by iterating *F* and starting with *s*. For example when computing the coefficients of the sine series we have $f\ i = fact\ (2 \cdot i + 1)$, $F\ i = i + 2$, $s = 2$ and $G\ i\ j = j \cdot i \cdot (i + 1)$. In the following table we see the values when *lb* or *ub* is called to compute the first four members of the series.

n	3	2	1	0
member	$1/f(0)$	$1/f(1) \cdot x$	$1/f(2) \cdot x^2$	$1/f(3) \cdot x^3$
$f\ i$	1	$1 \cdot 2 \cdot 3$	$1 \cdot 2 \cdot 3 \cdot 4 \cdot 5$	$1 \cdot 2 \cdot 3 \cdot 4 \cdot 5 \cdot 6 \cdot 7$
$F^i\ s$	2	4	6	8

Lemma Horner-Bounds**:** Assuming:

1. $0 \leq «x»$
2. $\bigwedge n.\ f\ (n + 1) = G\ (F^n\ s)\ (f\ n)$
3. $\bigwedge i\ k\ x.\ lb\ 0\ i\ k\ x = 0$
4. $\bigwedge n\ i\ k\ x.\ lb\ (n + 1)\ i\ k\ x = lapprox\text{-}rat\ prec\ 1\ k - x \cdot ub\ n\ (F\ i)\ (G\ i\ k)\ x$
5. $\bigwedge i\ k\ x.\ ub\ 0\ i\ k\ x = 0$
6. $\bigwedge n\ i\ k\ x.\ ub\ (n + 1)\ i\ k\ x = rapprox\text{-}rat\ prec\ 1\ k - x \cdot lb\ n\ (F\ i)\ (G\ i\ k)\ x$

Show:

1. $«lb\ n\ (F^i\ s)\ (f\ i)\ x» \leq \sum_{j=0}^{n-1} (-1)^j \cdot \frac{1}{f\ (i + j)} \cdot «x»^j$
2. $\sum_{j=0}^{n-1} (-1)^j \cdot \frac{1}{f\ (i + j)} \cdot «x»^j \leq «ub\ n\ (F^i\ s)\ (f\ i)\ x»$

For our proofs we instantiate $i = 0$ or $i = 1$ and then have the desired rule. *i* is needed for the induction proof only. The computation of the reciprocal is never exact on floating point numbers as *lapprox-rat* and *rapprox-rat* are approximations. Hence the conclusions of Horner-Bounds are inequalities instead of equations.

In some cases the input value to the polynomial is negative. In these cases the signs of the members move from the variable *x* to the coefficients. Hence it is necessary to change the minus in the Horner scheme functions to a plus. Everything else remains unchanged:

Lemma Horner-Bounds-Nonpos**:** Assuming:

1. $«x» \leq 0$
2. $\bigwedge n.\ f\ (n+1) = G\ (F^n\ s)\ (f\ n)$
3. $\bigwedge i\ k\ x.\ lb\ 0\ i\ k\ x = 0$
4. $\bigwedge n\ i\ k\ x.\ lb\ (n+1)\ i\ k\ x = lapprox\text{-}rat\ prec\ 1\ k + x \cdot ub\ n\ (F\ i)\ (G\ i\ k)\ x$
5. $\bigwedge i\ k\ x.\ ub\ 0\ i\ k\ x = 0$
6. $\bigwedge n\ i\ k\ x.\ ub\ (n+1)\ i\ k\ x = rapprox\text{-}rat\ prec\ 1\ k + x \cdot lb\ n\ (F\ i)\ (G\ i\ k)\ x$

Show:

1. $«lb\ n\ (F^i\ s)\ (f\ i)\ x» \leq \sum_{j=0}^{n-1} \frac{1}{f\ (i+j)} \cdot «x»^j$
2. $\sum_{j=0}^{n-1} \frac{1}{f\ (i+j)} \cdot «x»^j \leq «ub\ n\ (F^i\ s)\ (f\ i)\ x»$

With Horner-Bounds-Nonpos and Horner-Bounds we have rules at hand which are used to prove the correctness of the approximation of the transcendental functions.

4.4 Elementary functions

In the following sections we show the formalization of the approximation functions. As the final theorem for each function we prove a theorem satisfying (2.1).

4.4.1 Square root

Unlike the transcendental functions the square root is not calculated using a power series, but using Newton iteration. First we define *sqrt-iteration* to approximate the upper bound:

Definition *sqrt-iteration* :: *nat* ⇒ *nat* ⇒ *float* ⇒ *float*:

$sqrt\text{-}iteration\ prec\ 0\ (Float\ m\ e) = Float\ 1\ ((e + bitlen\ m)\ div\ 2 + 1)$
$sqrt\text{-}iteration\ prec\ (m+1)\ x = (\textbf{let}\ y = sqrt\text{-}iteration\ prec\ m\ x\ \textbf{in}\ \frac{1}{2} \cdot (y + float\text{-}divr\ prec\ x\ y))$

As the first approximation step we select a power of 2 which is greater than the square root, i.e. this is a simple and good approximation as we know that $\sqrt{m \cdot 2^{2e}} = \sqrt{m} \cdot 2^e$. In each further step one Newton iteration is executed. We know for each step in the iteration that the result is an upper bound of the exact value:

Lemma sqrt_iteration-Bound**:** *If* $0 < «x»$ *then* $sqrt\ «x» < «sqrt\text{-}iteration\ prec\ n\ x»$.

Using the Newton iteration, we now define the upper and lower bounds using the specified precision as the amount of steps to iterate. Since the start value of our approximation is less than twice $\sqrt{x}$, and with each iteration we get one bit precision, it is enough to do *prec* steps to reach the desired precision.

The *ub-sqrt* and *lb-sqrt* functions also check if the input value is negative and return *None* in that case.

Definition *ub-sqrt* :: *nat* ⇒ *float* ⇀ *float*:

ub-sqrt prec x = (if *0* < *x* then ⌊*sqrt-iteration prec prec x*⌋ else if *x* < *0* then *None* else ⌊*0*⌋)

Definition *lb-sqrt* :: *nat* ⇒ *float* ⇀ *float*:

lb-sqrt prec x =
(if *0* < *x* then ⌊*float-divl prec x* (*sqrt-iteration prec prec x*)⌋ else if *x* < *0* then *None* else ⌊*0*⌋)

These functions are directly used in the approximation function to calculate the square root. So we need to show the instantiation of (2.1) for these functions:

Theorem BNDS-SQRT**:**

(⌊*l*⌋, ⌊*u*⌋) = (*lb-sqrt prec lx*, *ub-sqrt prec ux*) ∧ *x* ∈ {«*lx*» .. «*ux*»} ⟹ «*l*» ≤ *sqrt x* ∧ *sqrt x* ≤ «*u*»

For later proofs it is necessary to know that the square root approximation is always positive. This is obvious for *ub-sqrt*. However, we need to show it explicitly for *lb-sqrt*:

Lemma LB_SQRT-LOWER-BOUND**:** *If 0* ≤ «*x*» *then 0* ≤ «*the* (*lb-sqrt prec x*)».

4.4.2 Arc tangent and Pi

Before we introduce the transcendental functions we need to calculate π up to an arbitrary precision. π is introduced by means of Machin's formula, using the arc tangent. We use Machin's formula as it is an easy to prove but also fast method.

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

To compute this we look at the arc tangent series:

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

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

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

As the exponent for x is not correct we need to apply a transformation to $h_n(x)$:

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

The $h_n(x)$ is computed by mutual recursive upper and lower bound functions implementing the Horner Scheme. Since the arc tangent series is an alternating power series, the partial sum up to an odd n is the upper limit, and up to an even n is the lower limit. The input value to h_n is squared, hence x is always positive.

lb-arctan-horner prec n 1 x computes the lower 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 $\mathit{lb\text{-}arctan\text{-}horner} :: \mathit{nat} \Rightarrow \mathit{nat} \Rightarrow \mathit{nat} \Rightarrow \mathit{float} \Rightarrow \mathit{float}$:

$\mathit{lb\text{-}arctan\text{-}horner\ prec\ 0\ k\ x} = \mathit{0}$
$\mathit{lb\text{-}arctan\text{-}horner\ prec}\ (n+1)\ k\ x = \mathit{lapprox\text{-}rat\ prec\ 1\ k} - x \cdot \mathit{ub\text{-}arctan\text{-}horner\ prec\ n}\ (k+2)\ x$

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

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

Since MACHIN only needs the arc tangent in the interval $\{-1 .. 1\}$, we can use the arc tangent series and this equation to calculate π. To estimate the precision we know that $\left(\frac{1}{5}\right)^2 < 2^{-4}$ and $\left(\frac{1}{239}\right)^2 < 2^{-14}$, hence with each computed member we gain 4 and 14 bits respectively. Hence we can divide the precision by 4 and 14 to have the required precision but also a fast method to calculate π:

Definition $\mathit{ub\text{-}pi} :: \mathit{nat} \Rightarrow \mathit{float}$:

$\mathit{ub\text{-}pi\ prec} =$
$(\mathsf{let}\ A = \mathit{rapprox\text{-}rat\ prec\ 1\ 5};\ B = \mathit{lapprox\text{-}rat\ prec\ 1\ 239}$
$\ \mathsf{in}\ 4 \cdot (4 \cdot A \cdot \mathit{ub\text{-}arctan\text{-}horner\ prec}\ (\mathit{get\text{-}odd}\ (\mathit{prec\ div}\ 4 + 1))\ 1\ (A \cdot A) -$
$\qquad B \cdot \mathit{lb\text{-}arctan\text{-}horner\ prec}\ (\mathit{get\text{-}even}\ (\mathit{prec\ div}\ 14 + 1))\ 1\ (B \cdot B)))$

Definition $\mathit{lb\text{-}pi} :: \mathit{nat} \Rightarrow \mathit{float}$:

$\mathit{lb\text{-}pi\ prec} =$
$(\mathsf{let}\ A = \mathit{lapprox\text{-}rat\ prec\ 1\ 5};\ B = \mathit{rapprox\text{-}rat\ prec\ 1\ 239}$
$\ \mathsf{in}\ 4 \cdot (4 \cdot A \cdot \mathit{lb\text{-}arctan\text{-}horner\ prec}\ (\mathit{get\text{-}even}\ (\mathit{prec\ div}\ 4 + 1))\ 1\ (A \cdot A) -$
$\qquad B \cdot \mathit{ub\text{-}arctan\text{-}horner\ prec}\ (\mathit{get\text{-}odd}\ (\mathit{prec\ div}\ 14 + 1))\ 1\ (B \cdot B)))$

Theorem BNDS-PI**:** $\mathit{pi} \in \{«\mathit{lb\text{-}pi}\ n» .. «\mathit{ub\text{-}pi}\ n»\}$

To expand the arc tangent to the entire real numbers, we apply the following transformations if x is outside of the interval $\{-1 .. 1\}$. To reach the desired precision, the input value needs to be even below $\frac{1}{2}$ as the coefficients only shrink linearly. This goal is achieved by applying the following transformations:

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* $\mathit{arctan}\ \frac{1}{x} = \frac{\mathit{sgn}\ x \cdot \mathit{pi}}{2} - \mathit{arctan}\ x$.

With these formulas we now use the arc tangent series to compute the arc tangent on the entire domain:

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

We now map this equation to the computation of lower and upper bounds in floating point numbers. If $x < 0$ we only call the other bound. In the second case we directly use the polynomial until the precision is reached. If $\frac{1}{2} < x \leq 2$ we check the result of the division not to be outside of the Taylor series' convergence area, e.g. $y < 1$. In the last three cases we know the input value to the recursive call to arctan and directly evaluate the Taylor series.

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 ≤ $\frac{1}{2}$ then ub-horner x
         else if x ≤ 2
              then let y = float-divr prec x (1 + the (lb-sqrt prec (1 + x · x)))
                   in if 1 < y then ub-pi prec · $\frac{1}{2}$ else 2 · ub-horner y
              else ub-pi prec · $\frac{1}{2}$ − 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 ≤ $\frac{1}{2}$ 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 inv = float-divr prec 1 x in if 1 < inv then 0 else lb-pi prec · $\frac{1}{2}$ − ub-horner inv)
```

Now we show that these functions compute the arc tangent boundaries for all floating point numbers. Hence we show the instantiation of (2.1) for these functions:

Theorem BNDS-ARCTAN**:**

$(l,\ u) = (\textit{lb-arctan prec lx},\ \textit{ub-arctan prec ux}) \land x \in \{«lx» \,..\, «ux»\} \Longrightarrow$
$«l» \leq \textit{arctan}\ x \land \textit{arctan}\ x \leq «u»$

4.4.3 Sine and Cosine

To compute the sine and cosine bounds, we again use the Taylor series. Here we use MacLaurin's lemma, already proved in Isabelle/HOL in the range $x \in \{0 \,..\, \frac{1}{2}\}$:

Lemma MacLaurin-cos**:**

If $0 < x$ *and* $0 < n$ *then*
$\exists\, t{>}0.\ t < x \land \cos x = \sum_{m=0}^{n-1} (\textsf{if } \textit{even } m \textsf{ then } \frac{-1^{m\ div\ 2}}{fact\ m} \textsf{ else } 0) \cdot x^m + \frac{cos\ (t + \frac{1}{2} \cdot n \cdot pi)}{fact\ n} \cdot x^n.$

As we only compute upper and lower bounds we use the left part of the sum. We rewrite the index to avoid the *if even m then ...* for better readability.

$$\cos x = \sum_{i=0}^{n-1} (-1)^i \cdot \frac{1}{(2 \cdot i)!} \cdot x^{2 \cdot i} + \dots$$

For computing this partial sum we introduce an auxiliary function $h_n^q(x)$. We introduce the parameter q to later reuse that function when implementing the sine function:

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

This function is implemented using the Horner Scheme to compute its upper and lower bounds. The k parameter to these functions is only internally used to calculate the coefficients. The functions are called with $k = 1$. Here *lb-sin-cos-aux prec n q 1 x* computes the lower and *ub-sin-cos-aux prec n q 1 x* the upper bound of $h_n^q(x)$ to the precision *prec*.

Definition *ub-sin-cos-aux* :: *nat* ⇒ *nat* ⇒ *nat* ⇒ *nat* ⇒ *float* ⇒ *float*:

ub-sin-cos-aux prec 0 q k x = *0*
ub-sin-cos-aux prec (*n*+ *1*) *q k x* = *rapprox-rat prec 1 k*− *x*· *lb-sin-cos-aux prec n* (*q*+ *2*) (*k*· *q*· (*q*+ *1*)) *x*

Definition *lb-sin-cos-aux* :: *nat* ⇒ *nat* ⇒ *nat* ⇒ *nat* ⇒ *float* ⇒ *float*:

lb-sin-cos-aux prec 0 q k x = *0*
lb-sin-cos-aux prec (*n*+ *1*) *q k x* = *lapprox-rat prec 1 k*− *x*· *ub-sin-cos-aux prec n* (*q*+ *2*) (*k*· *q*· (*q*+ *1*)) *x*

$h_n^q(x)$ is not directly the partial sum of the cosine. We need to set $q = 1$ and apply a transformation to x to adjust the exponents:

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

Using this and MacLaurin's lemma we show that *lb-sin-cos-aux* and *ub-sin-cos-aux* calculate the correct boundaries for the convergence radius $\frac{\pi}{2}$:

Lemma COS-BOUNDS**:**

If $0 \leq$ «*x*» *and* «*x*» $\leq \frac{pi}{2}$ *then*
cos «*x*» ∈ {«*lb-sin-cos-aux prec* (*get-even n*) *1 1* (*x*· *x*)» .. «*ub-sin-cos-aux prec* (*get-odd n*) *1 1* (*x*· *x*)»}.

We use the following case distinction when computing cos in the range $\{0..\frac{\pi}{2}\}$:

$$\cos(x) = \begin{cases} \sum_{n=0}^{\infty} (-1)^n \cdot \frac{1}{(2 \cdot n)!} \cdot x^{2 \cdot n} & \text{if} \, x < \frac{1}{2} \\ 2 \cdot \cos\left(\frac{x}{2}\right)^2 - 1 & \text{if} \, \frac{1}{2} \leq x < 1 \\ 2 \cdot \left(2 \cdot \cos\left(\frac{x}{4}\right)^2 - 1\right)^2 - 1 & \text{otherwise} \end{cases} \tag{4.1}$$

The last case is nothing else than the second case unfolded twice. This is used to avoid a real recursive call. In our implementations of the bounds we do not call *ub-cos* or *lb-cos* recursively. Instead we apply the Taylor series directly. Unfortunately the result of the Horner scheme can be negative, hence we need a guard when the second or third case is chosen. Otherwise it is not guaranteed that the result is an upper or lower bound. Hence in these cases we only return −1.

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

lb-cos prec x =
(*let horner* = λ*x*. *lb-sin-cos-aux prec* (*get-even* (*prec div 4*+ *1*)) *1 1* (*x*· *x*);
    half = λ*x*. *if x* < *0 then* − *1 else 2*· *x*· *x*− *1*
in if x < $\frac{1}{2}$ *then horner x else if x* < *1 then half* (*horner* (*x*· $\frac{1}{2}$)) *else half* (*half* (*horner* (*x*· $\frac{1}{4}$))))

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

```
ub-cos prec x =
(let horner = λx. ub-sin-cos-aux prec (get-odd (prec div 4 + 1)) 1 1 (x · x); half = λx. 2 · x · x − 1
 in if x < 1/2 then horner x else if x < 1 then half (horner (x · 1/2)) else half (half (horner (x · 1/4))))
```

Finally we implement *bnds-cos*, which returns just the range of cosine when the input is outside of the π-radius. In all other cases *lb-cos* and *ub-cos* are used:

Definition *bnds-cos* :: *nat* ⇒ *float* ⇒ *float* ⇒ *float* × *float*:

```
bnds-cos prec lx ux =
(let lpi = lb-pi prec
 in if lx < − lpi ∨ lpi < ux then (−1, 1)
    else if ux ≤ 0 then (lb-cos prec (− lx), ub-cos prec (− ux))
         else if 0 ≤ lx then (lb-cos prec ux, ub-cos prec lx)
              else (min (lb-cos prec (− lx)) (lb-cos prec ux), 1))
```

Finally we show the instantiation of (2.1) for the cosine approximation:

Theorem BNDS-COS**:**
$(l,\ u) = \mathit{bnds\text{-}cos\ prec\ lx\ ux} \wedge x \in \{«lx» \mathinner{..} «ux»\} \Longrightarrow «l» \leq \mathit{cos}\ x \wedge \mathit{cos}\ x \leq «u»$

We use the MacLaurin's lemma already proved in Isabelle/HOL to use the Horner scheme for the computation of sine:

Lemma MACLAURIN-SIN**:**

If $0 < n$ *and* $0 < x$ *then*
$\exists\, t{>}0.\ t < x \wedge \mathit{sin}\ x = \sum_{m=0}^{n-1} (\textsf{if}\ \mathit{even}\ m\ \textsf{then}\ 0\ \textsf{else}\ \frac{-1^{(m\,-\,1)\ div\ 2}}{fact\ m}) \cdot x^m + \frac{sin\ (t + \frac{1}{2} \cdot n \cdot pi)}{fact\ n} \cdot x^n.$

As we only compute upper and lower bounds we use the left part of the sum. We rewrite the index to avoid the *if even m then* ... for better readability.

$$\sin x = \sum_{i=0}^{n-1} (-1)^i \cdot \frac{1}{(2 \cdot i + 1)!} \cdot x^{2 \cdot i + 1} + \dots$$

We reuse the $h_n^q(x)$ introduced to compute the series of the cosine for the sine's series:

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

Hence the Horner scheme implemented by *ub-sin-cos-aux* and *ub-sin-cos-aux* is used to compute boundaries for sine:

Lemma SIN-BOUNDS**:**

If $0 \leq «x»$ *and* $«x» \leq \frac{pi}{2}$ *then*
$\mathit{sin}\ «x»$
$\in \{«x \cdot \mathit{lb\text{-}sin\text{-}cos\text{-}aux\ prec\ (get\text{-}even\ n)\ 2\ 1}\ (x \cdot x)» \mathinner{..} «x \cdot \mathit{ub\text{-}sin\text{-}cos\text{-}aux\ prec\ (get\text{-}odd\ n)\ 2\ 1}\ (x \cdot x)»\}.$

Then we introduce *lb-sin* and *ub-sin*. Here we use $(\mathit{sin}\ x)^2 = 1 - (\mathit{cos}\ x)^2$ to calculate the sine outside the $\frac{1}{2}$-radius. The following case distinctions are used:

$$\sin(x) = \begin{cases} -\sin(-x) & \text{if}\, x < 0 \\ \displaystyle\sum_{n=0}^{\infty} (-1)^n \cdot \frac{1}{(2 \cdot n + 1)!} \cdot x^{2 \cdot n + 1} & \text{if}\, 0 \leq x < \frac{1}{2} \\ \sqrt{1 - \cos(x)^2} & \text{otherwise} \end{cases}$$

On the bound computations we need to check in the third case that the results of the cosine bounds are not out of the cosine range. Unfortunately this happens in some cases.

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

```
lb-sin prec x =
(let sqr-diff = λx. if 1 < x then 0 else 1 − x · x
 in if x < 0 then − ub-sin prec (− x)
    else if x ≤ $\frac{1}{2}$ then x · lb-sin-cos-aux prec (get-even (prec div 4 + 1)) 2 1 (x · x)
         else the (lb-sqrt prec (sqr-diff (ub-cos prec x))))
```

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

```
ub-sin prec x =
(let sqr-diff = λx. if x < 0 then 1 else 1 − x · x
 in if x < 0 then − lb-sin prec (− x)
    else if x ≤ $\frac{1}{2}$ then x · ub-sin-cos-aux prec (get-odd (prec div 4 + 1)) 2 1 (x · x)
         else the (ub-sqrt prec (sqr-diff (lb-cos prec x))))
```

Finally we implement *bnds-sin*, which only returns the range of sine if the input is outside of the $\frac{\pi}{2}$-radius. In all other cases *lb-sin* and *ub-sin* are used:

Definition *bnds-sin* :: *nat* ⇒ *float* ⇒ *float* ⇒ *float* × *float*:

```
bnds-sin prec lx ux =
(let lpi = lb-pi prec; half-pi = lpi · $\frac{1}{2}$
 in if lx ≤ − half-pi ∨ half-pi ≤ ux then (−1, 1) else (lb-sin prec lx, ub-sin prec ux))
```

Finally we need to show the instantiation of (2.1) for the sine approximation:

Theorem BNDS-SIN**:**
If $(l,\ u) = \mathit{bnds\text{-}sin\ prec\ lx\ ux} \wedge x \in \{«lx» \ ..\ «ux»\}$ *then*
$«l» \leq \sin x \wedge \sin x \leq «u»$.

4.4.4 Exponential function

To compute the bounds of the exponential function, we again use the Taylor series. Here we use the MacLaurin's lemma for the exponential function.

Lemma MACLAURIN-EXP**:**

$$\exists\, t.\ |t| \leq |x| \wedge \mathit{exp}\ x = \sum_{m=0}^{n-1} \frac{x^m}{\mathit{fact}\ m} + \frac{\mathit{exp}\ t}{\mathit{fact}\ n} \cdot x^n$$

We only use the Taylor series to compute the exponential function in the range $x \in \{-1\ ..\ 0\}$ because only in this range it is useable as boundaries. In that case we only use the left part of the equation:

$$\exp x = \sum_{i=0}^{n-1} (-1)^i \cdot \frac{1}{i!} \cdot x^i + \ldots$$

We describe this partial sums as $h_n(x)$:

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

To compute the boundaries of $h_n(x)$ to the precision *prec* we introduce *lb-exp-horner prec n 1 1 x* for the lower and *ub-exp-horner prec n 1 1 x* for the upper bound.

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

```
ub-exp-horner prec 0 i k x = 0
ub-exp-horner prec (n+ 1) i k x = rapprox-rat prec 1 k+ x· lb-exp-horner prec n (i+ 1) (k· i) x
```

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

```
lb-exp-horner prec 0 i k x = 0
lb-exp-horner prec (n+ 1) i k x = lapprox-rat prec 1 k+ x· ub-exp-horner prec n (i+ 1) (k· i) x
```

Theorem EXP-BOUNDS**:**

If «x» ≤ *0 then*
exp «x» ∈ {«*lb-exp-horner prec* (*get-even n*) *1 1 x*» .. «*ub-exp-horner prec* (*get-odd n*) *1 1 x*»}.

To compute the exponential function we use the following case distinction. We use *exp* $(-\,x)$ = *inverse* (*exp x*) or split the input into its integer part and its fraction. Here $\lfloor x \rfloor$ is the greatest integer smaller or equal x:

$$\exp(x) = \begin{cases} \dfrac{1}{\exp(-x)} & \text{if}\, 0 < x \\ \displaystyle\sum_{n=0}^{\infty} (-1)^n \cdot \frac{1}{n} \cdot x^n & \text{if } -1 \le x \le 0 \\ \exp\left(-\dfrac{x}{\lfloor x \rfloor}\right)^{-\lfloor x \rfloor} & \text{otherwise} \end{cases}$$

The implementation as bounding functions is straight forward. We only need to take care of the lower boundary. Here it can happen that the result of the division is negative or zero. In this case we return $\frac{1}{4} < \exp(-1)$:

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

```
ub-exp prec x =
(if 0 < x then float-divr prec 1 (lb-exp prec (− x))
 else if x < − 1
      then case floor-fl x of
             Float m e ⇒ (ub-exp-horner prec (get-odd (prec + 2)) 1 1 (float-divr prec x (− Float m e)))^(− m · 2^e)
      else ub-exp-horner prec (get-odd (prec+ 2)) 1 1 x)
```

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

```
lb-exp prec x =
(if 0 < x then float-divl prec 1 (ub-exp prec (− x))
 else let horner = λx. let y = lb-exp-horner prec (get-even (prec+ 2)) 1 1 x in if y ≤ 0 then 1/4 else y
      in if x < − 1 then case floor-fl x of Float m e ⇒ (horner (float-divl prec x (− Float m e)))^(− m · 2^e)
         else horner x)
```

Finally we need to show the instantiation of (2.1) for the approximation of the exponential function:

Theorem BNDS_EXP**:**

If (*l*, *u*) = (*lb-exp prec lx*, *ub-exp prec ux*) ∧ *x* ∈ {«*lx*» .. «*ux*»} *then* «*l*» ≤ *exp x* ∧ *exp x* ≤ «*u*».

4.4.5 Logarithm

We use the Taylor series for the logarithm:

Lemma LN_SERIES: *If* $0 < x$ *and* $x < 2$ *then* $ln\ x = \sum_{n=0}^{\infty} (-1)^n \cdot \frac{1}{n+1} \cdot (x-1)^{n+1}$.

To compute the partial sums of the logarithm series $h_n(x)$ is introduced:

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

ub-ln-horner prec n 1 x is implemented to compute the upper bound of $h_n(x)$ to precision *prec* and *lb-ln-horner prec n 1 x* for the lower bound.

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

ub-ln-horner prec 0 i x = *0*
ub-ln-horner prec (*n*+ *1*) *i x* = *rapprox-rat prec 1 i* − *x*· *lb-ln-horner prec n* (*i*+ *1*) *x*

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

lb-ln-horner prec 0 i x = *0*
lb-ln-horner prec (*n*+ *1*) *i x* = *lapprox-rat prec 1 i* − *x*· *ub-ln-horner prec n* (*i*+ *1*) *x*

To get the correct form of the logarithm series we need to apply a transformation to $h_n(x)$:

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

Lemma LN-BOUNDARIES:

If $0 \le$ «*x*» *and* «*x*» < 1 *then*
ln («*x*» + *1*) ∈ {«*x*· *lb-ln-horner prec* (*get-even n*) *1 x*» .. «*x*· *ub-ln-horner prec* (*get-odd n*) *1 x*»}.

Unfortunately this only produces results of the requested precision for inputs in the range {1 .. < 2}. We use $0 < x \Longrightarrow ln\ (inverse\ x) = -\ ln\ x$ to only apply values above 1 to the Taylor series. Now the structure of our floating point numbers is exploited to get values in the range {1 .. < 2}. So the following formula is applied:

Lemma LN-SHIFTED-FLOAT:
If $0 < m$ *then ln* «*Float m e*» = *ln 2*· (*e*+ *bitlen m*− *1*) + *ln* «*Float m* (− (*bitlen m*− *1*))».

Before implementing this formula, we need to compute *ln 2*. This is not possible with the Taylor series. Hence we first introduce the addition theorem on the logarithm:

Lemma LN-ADD: *If* $0 < x$ *and* $0 < y$ *then* $ln\ (x+ y) = ln\ x + ln\ (1 + \frac{y}{x})$.

This equation is instantiated where x is $\frac{3}{2}$ and y is $\frac{1}{2}$ (hence $\frac{y}{x} = \frac{1}{3}$). The resulting values are used to compute *ln 2* with the Taylor series:

Definition *ub-ln2* :: *nat* ⇒ *float*:

ub-ln2 prec =
(let *third* = *rapprox-rat* (*max prec 1*) *1 3*
 in $\frac{1}{2}$ · *ub-ln-horner prec* (*get-odd prec*) *1* $\frac{1}{2}$ + *third*· *ub-ln-horner prec* (*get-odd prec*) *1 third*)

Definition *lb-ln2* :: *nat* ⇒ *float*:

lb-ln2 prec =
(let *third* = *lapprox-rat prec 1 3*
 in $\frac{1}{2}$ · *lb-ln-horner prec* (*get-even prec*) *1* $\frac{1}{2}$ + *third*· *lb-ln-horner prec* (*get-even prec*) *1 third*)

Lemma LB_LN2: *«lb-ln2 prec»* $\leq$ *ln 2*

Lemma UB_LN2: *ln 2* $\leq$ *«ub-ln2 prec»*

Using this equations and LN-SHIFTED-FLOAT we now define our functions to compute the logarithm boundaries for arbitrary positive floating point numbers. To do this we use the following case distinctions. Here x_m and x_e are the mantissa and the exponent of x, respectively. $\log_2(x)$ is the integer logarithm to the base 2:

$$\ln(x) = \begin{cases} \text{undefined} & \text{if}\, x \leq 0 \\ -\ln\left(\dfrac{1}{x}\right) & \text{if}\, 0 < x < 1 \\ \displaystyle\sum_{n=0}^{\infty} (-1)^n \cdot \frac{1}{n+1} \cdot (x-1)^{n+1} & \text{if}\, 1 \leq x < 2 \\ \ln 2 \cdot (x_e + \log_2(x_m)) - \ln\left(x_m \cdot 2^{-\log_2(x_m)}\right) & \text{otherwise} \end{cases}$$

The implementation as lower and upper bounds is now straight forward. Instead of $\log_2(x_m)$ we use *bitlen* $(mantissa\ x) - 1$:

Definition *ub-ln* :: *nat* ⇒ *float* ⇀ *float*:

```
ub-ln prec x =
(if x ≤ 0 then None
 else if x < 1 then ⌊− the (lb-ln prec (float-divl (max prec 1) 1 x))⌋
      else let horner = λx. (x− 1) · ub-ln-horner prec (get-odd prec) 1 (x− 1)
           in if x < 2 then ⌊horner x⌋
              else let l = bitlen (mantissa x) − 1
                   in ⌊ub-ln2 prec · (scale x + l) + horner (Float (mantissa x) (− l))⌋)
```

Definition *lb-ln* :: *nat* ⇒ *float* ⇀ *float*:

```
lb-ln prec x =
(if x ≤ 0 then None
 else if x < 1 then ⌊− the (ub-ln prec (float-divr prec 1 x))⌋
      else let horner = λx. (x− 1) · lb-ln-horner prec (get-even prec) 1 (x− 1)
           in if x < 2 then ⌊horner x⌋
              else let l = bitlen (mantissa x) − 1
                   in ⌊lb-ln2 prec · (scale x + l) + horner (Float (mantissa x) (− l))⌋)
```

Again we now show that these computations of the upper and lower bounds of ln are correct. Finally, we show the instantiation of (2.1):

Theorem BNDS-LN:

$$(\lfloor l \rfloor,\ \lfloor u \rfloor) = (lb\text{-}ln\ prec\ lx,\ ub\text{-}ln\ prec\ ux) \wedge x \in \{«lx» .. «ux»\} \Longrightarrow «l» \leq ln\ x \wedge ln\ x \leq «u»$$

4.5 Approximation of real valued formulas

Now we have implemented the computation of the most important transcendent functions. The basic arithmetic operations are easy to implement for interval arithmetic. We directly implement them in the final approximation function. However, we first define the syntax and semantics of the arithmetic formulas.

4.5.1 Model of formulas

The syntax of our formulas is defined as a data type in Isabelle/HOL:

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 reification should produce *floatarith* values from an arithmetic HOL expression. Each constructor describes an arithmetic operation. The only exception is the *Atom* constructor. It is used to look up a variable in a provided list of values. *Ifloatarith* is used to describe the semantic of the arithmetic formulas on real numbers:

Definition *Ifloatarith* :: *floatarith* ⇒ *real list* ⇒ *real*:

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

There are some operations missing such as subtraction, division, or the tangent function. However, these operations are not needed as atomic operations. by using the definition of these operations we prove the following additional lemmas:

Lemma Ifloatarith-Additional-Arithmetic**:**

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

To also support numerals the following equations are needed:

Lemma Ifloatarith-num**:**

Ifloatarith (Num 0) vs = 0
Ifloatarith (Num 1) vs = 1
Ifloatarith (Num (number-of a)) vs = number-of a

Here is a simple example of how an arithmetic formula maps to its syntax representation as *Ifloatarith*:

Example IFLOATARITH-EXAMPLES**:** $\frac{1}{\mathit{sin}\ x} - \mathit{cos}\ y =$
$\mathit{Ifloatarith}\ (\mathit{Add}\ (\mathit{Mult}\ (\mathit{Num}\ 1)\ (\mathit{Inverse}\ (\mathit{Sin}\ (\mathit{Atom}\ 1))))\ (\mathit{Minus}\ (\mathit{Cos}\ (\mathit{Atom}\ 2))))\ [x,\ y]$

So with reification we now lift terms over real values into an *floatarith* data structure.

4.5.2 Approximation function

Our next step we is to implement the approximation function itself. This function takes the desired precision, the formula to calculate, and a list of intervals as input. As a result an optional interval is returned. When a calculation error happens, such as division by zero, *None* is returned, otherwise the result is returned. In the following definition *lift-bin*, *lift-un*, *lift-bin′* and *lift-un′* are used to lift the boundary functions into a function of (*float* × *float*) *option*. They check if the result of the previous operation was not *None*. *lift-bin′* and *lift-un′* also wrap the result with the *Some* constructor to get a (*float* × *float*) *option*.

Definition $\mathit{approx} :: \mathit{nat} \Rightarrow \mathit{floatarith} \Rightarrow (\mathit{float} \times \mathit{float})\ \mathit{list} \rightharpoonup \mathit{float} \times \mathit{float}$:

$$\begin{array}{l}
\mathit{approx'}\ \mathit{prec}\ a\ \mathit{bs} = \\
\quad (\mathsf{case}\ \mathit{approx}\ \mathit{prec}\ a\ \mathit{bs}\ \mathsf{of}\ \mathit{None} \Rightarrow \mathit{None} \mid \lfloor(l,\ u)\rfloor \Rightarrow \lfloor(\textit{round-down}\ \mathit{prec}\ l,\ \textit{round-up}\ \mathit{prec}\ u)\rfloor) \\
\mathit{approx}\ \mathit{prec}\ (\mathit{Add}\ a\ b)\ \mathit{bs} = \textit{lift-bin}'\ (\mathit{approx}'\ \mathit{prec}\ a\ \mathit{bs})\ (\mathit{approx}'\ \mathit{prec}\ b\ \mathit{bs})\ (\lambda l1\ u1\ l2\ u2.\ (l1 + l2,\ u1 + u2)) \\
\mathit{approx}\ \mathit{prec}\ (\mathit{Minus}\ a)\ \mathit{bs} = \textit{lift-un}'\ (\mathit{approx}'\ \mathit{prec}\ a\ \mathit{bs})\ (\lambda l\ u.\ (-\ u,\ -\ l)) \\
\mathit{approx}\ \mathit{prec}\ (\mathit{Mult}\ a\ b)\ \mathit{bs} = \\
\textit{lift-bin}'\ (\mathit{approx}'\ \mathit{prec}\ a\ \mathit{bs})\ (\mathit{approx}'\ \mathit{prec}\ b\ \mathit{bs}) \\
\ (\lambda a1\ a2\ b1\ b2. \\
\quad (\textit{float-nprt}\ a1 \cdot \textit{float-pprt}\ b2 + \textit{float-nprt}\ a2 \cdot \textit{float-nprt}\ b2 + \textit{float-pprt}\ a1 \cdot \textit{float-pprt}\ b1 + \\
\quad\ \textit{float-pprt}\ a2 \cdot \textit{float-nprt}\ b1,\ \textit{float-pprt}\ a2 \cdot \textit{float-pprt}\ b2 + \textit{float-pprt}\ a1 \cdot \textit{float-nprt}\ b2 + \\
\quad\ \textit{float-nprt}\ a2 \cdot \textit{float-pprt}\ b1 + \textit{float-nprt}\ a1 \cdot \textit{float-nprt}\ b1)) \\
\mathit{approx}\ \mathit{prec}\ (\mathit{Inverse}\ a)\ \mathit{bs} = \\
\textit{lift-un}\ (\mathit{approx}'\ \mathit{prec}\ a\ \mathit{bs}) \\
\ (\lambda l\ u.\ \mathsf{if}\ 0 < l \lor u < 0\ \mathsf{then}\ (\lfloor \textit{float-divl}\ \mathit{prec}\ 1\ u\rfloor,\ \lfloor \textit{float-divr}\ \mathit{prec}\ 1\ l\rfloor)\ \mathsf{else}\ (\mathit{None},\ \mathit{None})) \\
\mathit{approx}\ \mathit{prec}\ (\mathit{Sin}\ a)\ \mathit{bs} = \textit{lift-un}'\ (\mathit{approx}'\ \mathit{prec}\ a\ \mathit{bs})\ (\textit{bnds-sin}\ \mathit{prec}) \\
\mathit{approx}\ \mathit{prec}\ (\mathit{Cos}\ a)\ \mathit{bs} = \textit{lift-un}'\ (\mathit{approx}'\ \mathit{prec}\ a\ \mathit{bs})\ (\textit{bnds-cos}\ \mathit{prec}) \\
\mathit{approx}\ \mathit{prec}\ \mathit{Pi}\ \mathit{bs} = \lfloor(\textit{lb-pi}\ \mathit{prec},\ \textit{ub-pi}\ \mathit{prec})\rfloor \\
\mathit{approx}\ \mathit{prec}\ (\mathit{Min}\ a\ b)\ \mathit{bs} = \\
\textit{lift-bin}'\ (\mathit{approx}'\ \mathit{prec}\ a\ \mathit{bs})\ (\mathit{approx}'\ \mathit{prec}\ b\ \mathit{bs})\ (\lambda l1\ u1\ l2\ u2.\ (\mathit{min}\ l1\ l2,\ \mathit{min}\ u1\ u2)) \\
\mathit{approx}\ \mathit{prec}\ (\mathit{Max}\ a\ b)\ \mathit{bs} = \\
\textit{lift-bin}'\ (\mathit{approx}'\ \mathit{prec}\ a\ \mathit{bs})\ (\mathit{approx}'\ \mathit{prec}\ b\ \mathit{bs})\ (\lambda l1\ u1\ l2\ u2.\ (\mathit{max}\ l1\ l2,\ \mathit{max}\ u1\ u2)) \\
\mathit{approx}\ \mathit{prec}\ (\mathit{Abs}\ a)\ \mathit{bs} = \\
\textit{lift-un}'\ (\mathit{approx}'\ \mathit{prec}\ a\ \mathit{bs})\ (\lambda l\ u.\ (\mathsf{if}\ l < 0 \land 0 < u\ \mathsf{then}\ 0\ \mathsf{else}\ \mathit{min}\ |l|\ |u|,\ \mathit{max}\ |l|\ |u|)) \\
\mathit{approx}\ \mathit{prec}\ (\mathit{Arctan}\ a)\ \mathit{bs} = \textit{lift-un}'\ (\mathit{approx}'\ \mathit{prec}\ a\ \mathit{bs})\ (\lambda l\ u.\ (\textit{lb-arctan}\ \mathit{prec}\ l,\ \textit{ub-arctan}\ \mathit{prec}\ u)) \\
\mathit{approx}\ \mathit{prec}\ (\mathit{Sqrt}\ a)\ \mathit{bs} = \textit{lift-un}\ (\mathit{approx}'\ \mathit{prec}\ a\ \mathit{bs})\ (\lambda l\ u.\ (\textit{lb-sqrt}\ \mathit{prec}\ l,\ \textit{ub-sqrt}\ \mathit{prec}\ u)) \\
\mathit{approx}\ \mathit{prec}\ (\mathit{Exp}\ a)\ \mathit{bs} = \textit{lift-un}'\ (\mathit{approx}'\ \mathit{prec}\ a\ \mathit{bs})\ (\lambda l\ u.\ (\textit{lb-exp}\ \mathit{prec}\ l,\ \textit{ub-exp}\ \mathit{prec}\ u)) \\
\mathit{approx}\ \mathit{prec}\ (\mathit{Ln}\ a)\ \mathit{bs} = \textit{lift-un}\ (\mathit{approx}'\ \mathit{prec}\ a\ \mathit{bs})\ (\lambda l\ u.\ (\textit{lb-ln}\ \mathit{prec}\ l,\ \textit{ub-ln}\ \mathit{prec}\ u)) \\
\mathit{approx}\ \mathit{prec}\ (\mathit{Power}\ a\ n)\ \mathit{bs} = \textit{lift-un}'\ (\mathit{approx}'\ \mathit{prec}\ a\ \mathit{bs})\ (\textit{float-power-bnds}\ n) \\
\mathit{approx}\ \mathit{prec}\ (\mathit{Num}\ f)\ \mathit{bs} = \lfloor(f,\ f)\rfloor \\
\mathit{approx}\ \mathit{prec}\ (\mathit{Atom}\ i)\ \mathit{bs} = (\mathsf{if}\ i < |\mathit{bs}|\ \mathsf{then}\ \lfloor \mathit{bs}_{[i]}\rfloor\ \mathsf{else}\ \mathit{None})
\end{array}$$

Here we introduce the helper function *approx′* to cut off the length of the floating point numbers to the desired precision. The functions *round-up* and *round-down* are used to get the next upper and lower floating point numbers of the specified precision, respectively.

This is also the function where the operations from interval arithmetic (see formulas (2.3), (2.4), (2.5) and (2.6)) are implemented.

bounded-by vs bs is used to state that the variables in *vs* are bounded by the bounds in *bs*. The proof method shows *bounded-by vs bs* by a list of assumptions provided by the user. As this is done by using the simplifier, we implement *bounded-by* recursively:

Definition $\textit{bounded-by} :: \textit{real list} \Rightarrow (\textit{float} \times \textit{float})\ \textit{list} \Rightarrow \textit{bool}$:

$\textit{bounded-by}\ (v\ \#\ vs)\ ((l,\ u)\ \#\ bs) = ((«l» \leq v \wedge v \leq «u») \wedge \textit{bounded-by}\ vs\ bs)$
$\textit{bounded-by}\ []\ [] = \textit{True}$
$\textit{bounded-by}\ []\ (v\ \#\ va) = \textit{False}$
$\textit{bounded-by}\ (v\ \#\ va)\ [] = \textit{False}$

Now we want to show our final goal: The correctness of our approximations.

Theorem APPROX**:** *If* $\textit{bounded-by}\ xs\ vs$ *and* $\lfloor(l,\ u)\rfloor = \textit{approx prec arith}\ vs$ *then* $«l» \leq \textit{Ifloatarith arith}\ xs \wedge \textit{Ifloatarith arith}\ xs \leq «u»$.

The proof of the last theorem is a simple induction on the structure of the formula. For each constructor we either use already provided theorems about the basic arithmetic operations, or one of the theorems about our approximation functions.

We now have the tool to compute bounded formulas over the real numbers. However, to prove theorems we need to compute inequalities. Here again we introduce a data structure representing the syntax and we again need one function to define the semantic and one to do the approximation itself.

First we define the data structure:

Definition ApproxEq:

datatype $\textit{ApproxEq} = \textit{Less floatarith floatarith} \mid \textit{LessEqual floatarith floatarith}$

Here the semantic definition is very simple:

Definition $\textit{uneq} :: \textit{ApproxEq} \Rightarrow \textit{real list} \Rightarrow \textit{bool}$:

$\textit{uneq}\ (\textit{Less}\ a\ b)\ vs = (\textit{Ifloatarith}\ a\ vs < \textit{Ifloatarith}\ b\ vs)$
$\textit{uneq}\ (\textit{LessEqual}\ a\ b)\ vs = (\textit{Ifloatarith}\ a\ vs \leq \textit{Ifloatarith}\ b\ vs)$

The computation is also straight forward:

Definition $\textit{uneq}' :: \textit{nat} \Rightarrow \textit{ApproxEq} \Rightarrow (\textit{float} \times \textit{float})\ \textit{list} \Rightarrow \textit{bool}$:

```
$uneq'\ prec\ (Less\ a\ b)\ bs =$
    $(\mathsf{case}\ (approx\ prec\ a\ bs,\ approx\ prec\ b\ bs)\ \mathsf{of}$
     $(None,\ b) \Rightarrow False \mid (\lfloor(l,\ u)\rfloor,\ None) \Rightarrow False$
     $\mid (\lfloor(l,\ u)\rfloor,\ \lfloor(l',\ u')\rfloor) \Rightarrow u < l')$
$uneq'\ prec\ (LessEqual\ a\ b)\ bs =$
    $(\mathsf{case}\ (approx\ prec\ a\ bs,\ approx\ prec\ b\ bs)\ \mathsf{of}$
     $(None,\ b) \Rightarrow False \mid (\lfloor(l,\ u)\rfloor,\ None) \Rightarrow False$
     $\mid (\lfloor(l,\ u)\rfloor,\ \lfloor(l',\ u')\rfloor) \Rightarrow u \leq l')$
```

After the reification of inequalities we need to show that *uneq* holds for this formula. However this also follows if *uneq'* for this formula holds:

Theorem UNEQ_APPROX**:** *If* $\textit{bounded-by}\ vs\ bs$ *and* $\textit{uneq}'\ \textit{prec eq}\ bs$ *then* $\textit{uneq eq}\ vs$.

4.5.3 Implementation of the automatic tactic

Finally all parts are available to implement an automatic proof method. Proving a formula by approximation is done using the following steps:

Reification: Rewrite formula into an *ApproxEq* structure

Rewrite as approximation: Apply UNEQ_APPROX

Proof boundaries: Show variable boundaries using the assumptions

Evaluate *uneq'***:** Show inequality by evaluation

To simplify this task we provide the automatic tactic `approximation`. This tactic proves an inequality and all variable bounds are conjunctions of the lower and upper bounds in the assumptions. The user only needs to specify the required precision, which is not determined by the tactic automatically.

Example APPROXIMATION***:***

We show a simple example of the approximation method to demonstrate these steps. In the following proof the executed proof steps are followed by the resulting sub goals.

theorem assumes *3 ≤ x ∧ x ≤ 6* **shows** *sin (pi / x) > 0.4*

> *1.* $\frac{4}{10} < sin\ \frac{pi}{x}$

apply (*reify uneq-equations*)

> *1. uneq (Less (Mult (Num 4) (Inverse (Num 10))) (Sin (Mult Pi (Inverse (Atom 0))))) [x]*

apply (*rule uneq-approx*[**where** *prec=10* **and** *bs*=[(*Float 3 0*, *Float 6 0*)]])

> *1. bounded-by [x] [(3, 6)]*
> *2. uneq' 10 (Less (Mult (Num 4) (Inverse (Num 10))) (Sin (Mult Pi (Inverse (Atom 0))))) [(3, 6)]*

apply (*simp add*: *assms*)

> *1. uneq' 10 (Less (Mult (Num 4) (Inverse (Num 10))) (Sin (Mult Pi (Inverse (Atom 0))))) [(3, 6)]*

apply *eval*
done

This is done in one step by the approximation method:

theorem *3 ≤ x ∧ x ≤ 6 ⟹ sin (pi / x) > 0.4* **by** (*approximation 10*)

Chapter 5

Conclusion

5.1 Results

In this section we give an overview, how much time is spent in the proof method. All the timings were taken with the Isabelle development changeset `f65670092259` on an iMac Intel Core 2 Duo 2.4 GHz with 4 GB RAM.

The running time of the proof method can be split into two parts. One part is the compilation of the computation functions, which unfortunately can not be done just once. It needs to be done each time we call the evaluator. The compilation itself only takes a couple of seconds, independent of the formula's size and the precision.

The second part of the runtime is the computation itself. This largely depends on the precision but also on how many transcendental functions are instantiated in the formula. We assume that multiplication and division are the most time consuming elementary operations. For each member in the Taylor series of a transcendental function one multiplication and one division is performed. The amount of members calculated depends linear on the precision. The running time of the division and multiplication has a linear dependency on the length of the input values, e.g. the precision. So we have a quadratic dependency on the precision for each transcendental function.

	(5.1)	(5.2)	(5.3)	(5.4)	(5.5)	(5.6)	(5.7)	(5.8)	(5.9)
p=80	11.625	15.713	9.069	9.665	25.182	19.797	17.017	9.909	9.789
p=10	16.009	25.950	9.129	10.177	50.167	36.902	28.754	11.513	11.273
p=120	25.450	45.511	9.221	11.209	100.226	70.076	51.607	14.821	14.541
p=140	42.859	79.489	9.317	12.837	183.639	125.380	88.990	19.113	18.785
p=160	72.357	129.104	9.277	15.749	327.444	223.922	152.146	27.514	26.522
p=180	120.044	214.461	9.565	20.157	535.961	368.123	240.883	37.254	36.006
p=200	192.836	334.293	9.717	26.646	864.774	591.489	382.468	53.819	52.919

Figure 5.1: The runtime of the different formulas in seconds.

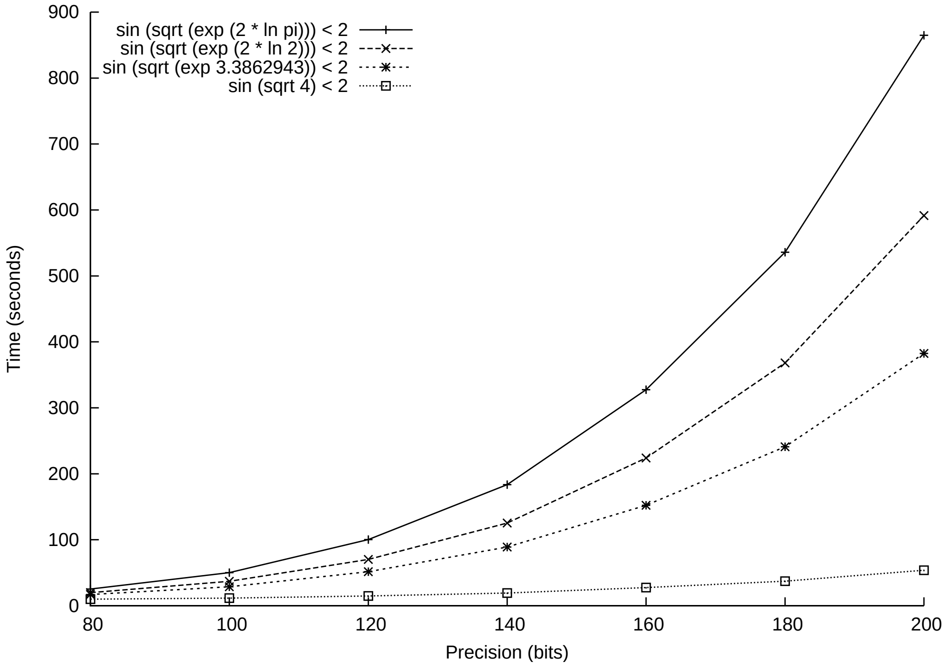

Figure 5.2: Execution time depending on formula length and precision

We use the following formulas to do time measurement with different precisions:

$$\left|\ln(2) - \frac{544531980202654583340825686620847}{785593587443817081832229725798400}\right| < \frac{1}{2^{51}} \tag{5.1}$$

$$|\exp(1.626) - 5.083499996273| < \frac{1}{10^{10}} \tag{5.2}$$

$$|\sqrt{2} - 1.4142135623730951| < \frac{1}{10^{15}} \tag{5.3}$$

$$|\pi - 3.14159265358979323851| < \frac{1}{10^{18}} \tag{5.4}$$

$$\sin(\sqrt{\exp(2 \cdot ln(\pi))}) < 2 \tag{5.5}$$

$$\sin(\sqrt{\exp(2 \cdot ln(2))}) < 2 \tag{5.6}$$

$$\sin(\sqrt{\exp 1.3862943}) < 2 \tag{5.7}$$

$$\sin(\sqrt{4}) < 2 \tag{5.8}$$

$$\sin(2) < 2 \tag{5.9}$$

The formulas (5.1), (5.2), (5.3) and (5.4) will be used to show the timing for different functions. In fig. 5.1 we see the exact timings of them. Figure 5.3 shows the quadratic dependency for transcendental functions. It also shows the nearly constant time for the square root, even on large precisions like 200 bits, is seen here.

The formulas (5.5), (5.6), (5.7), (5.8) and (5.9) are used to see the influence of the amount of transcendental functions used. In fig. 5.2 we see again the quadratic dependency, now for different numbers of transcendental functions calculated. For (5.5), (5.6) and (5.7) we also see here a linear

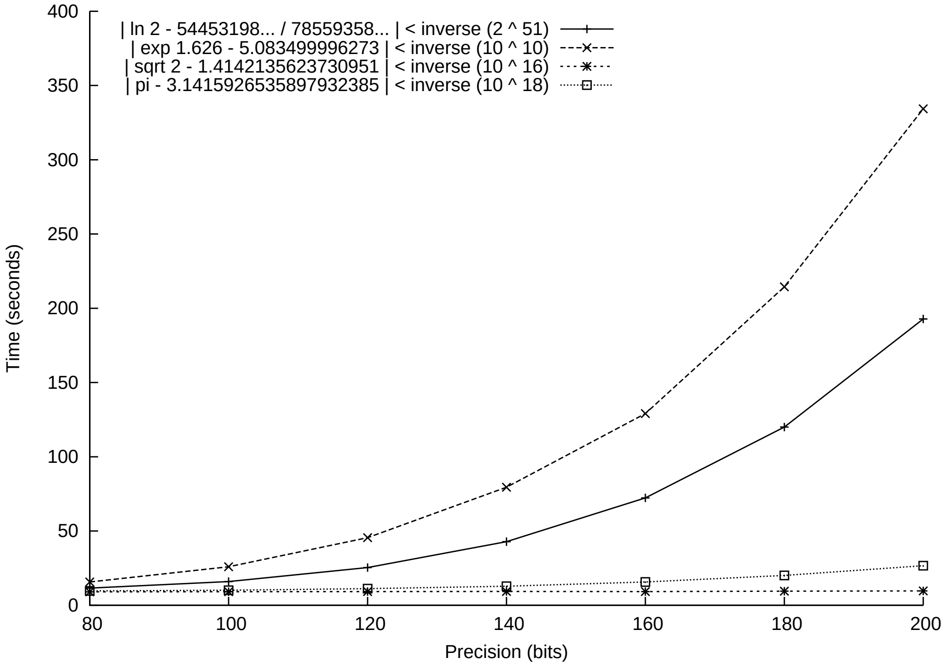

Figure 5.3: Execution time depending on function and precision

dependency of how many functions are used.

5.2 Related Work

We give an overview of other implementations of similar proof methods for interactive theorem provers.

In his PhD-thesis Harrison [10] describes how he formalizes exact arithmetic and floating point numbers in HOL and uses the exact arithmetic to calculate transcendental functions. All computations in his version are evaluated in the logic without code generation. Hence the execution needs to be passed through the LCF kernel. An example he gives is the calculations of $\ln(1 + \frac{1}{2})$ to a precision of 50 bits which takes 478 seconds. The advantage of his approach is that the trusted code base is not extended. Another difference is that he uses exact real arithmetic, e.g. each real x is represented as function $f_x(p)$ which computes an integer representing the first p decimal places of the exact real value. This also avoids the usage of interval arithmetic as the result of $f_x(p)$ always holds $|f_x(p) - 2^p x| < 1$.

For the theorem prover PVS, Daumas et al. [5] implemented the `numerical` proof method using algorithms very similar to the algorithms used in this work. They also use code generation to efficiently compute the results. Instead of using floating point numbers to represent the upper and lower bounds they use rational numbers. They do not mention any reduction of the size by approximating results with smaller numbers. There are also timings given for the computation of arctan in the precision of 8 bit, where the input value is divided into 18 sub intervals (see 5.3.1). This takes 46 seconds, i.e. 2.5 seconds per sub interval.

Melquiond [16] used the same technique to implement such a proof method for the theorem prover Coq. Instead of using rational numbers he also implemented the interval bounds as floating point numbers. Like Daumas et al. he also uses interval splitting to reduce the dependency effect. In his paper no performance measurement is provided, so no comparison with our implementation is possible. Melquiond uses techniques to expand the range of theorems which can be proved using his method, like interval splitting (described in 5.3.1) or use of Taylor models.

In his diploma thesis Varadi [23] implemented already some of the transcendental functions and the square root in Isabelle/HOL. He tried to generalize the usage of the Horner scheme by expanding each function into the first n members of the Taylor series. Where n is a number which can be specified by the user to describe the needed precision. A similar technique was used to expand the square root and generate one basic arithmetic expression, only consisting of basic arithmetic operations.

To evaluate these basic arithmetic expressions he used the approximation function in `Compute-Float` from Obua. On the one hand this allowed simple proofs, since even division can be lifted into the division operation in the basic arithmetic. On the other hand these expressions can get very big due to the dependency effect, rendering the results unusable in some cases. Hence we used a more direct approach and implemented the Horner functions directly in terms of operations on floating point numbers instead of using intervals. With the `HornerScheme` theory we could also abstract the Horner scheme when using it to define the computation of transcendental functions.

A different approach to proving inequalities of real formulas take Akbarpour and Paulson [1]. Again they transform the formula with transcendental functions and square root into a formula only consisting of basic arithmetic operations approximating transcendental functions by polynomials. This formula is then not proved by computation but is put into a first order prover. This approach can not be used to efficiently verify values such as π to an arbitrary precision, but to prove formulas like $|\exp(x) - 1| \leq \exp(|x|) - 1$ where over the entire set of real numbers is quantified.

5.3 Future Work

In this section we mention future extensions to the formalization of the computation of transcendental functions and proof methods developed in this work.

5.3.1 Interval splitting

Often a formula $F(x)$ could be solved when the variable x is split up into smaller intervals. This is often due to the *dependency effect*. For example when x is in $(0, 1)$ the result of $x - x$ is $(-1, 1)$. When the interval is split up into $(0, 0.5)$ and $(0.5, 1)$ we get for both sub intervals the result $(-0.5, 0.5)$, for smaller sub intervals this gets better.

We could detect such easy occurrences and remove them by some rewrite rules. Unfortunately there are more complicated formulas where the *dependency effect* occurs. For example $x^2 - x$ gives a much better result when rewritten as $(x - \frac{1}{2})^2 - \frac{1}{4}$ and it can also occur in the transcendental functions itself too. Hence some formulas are better proved when it is shown for intervals of a partition.

To do this automatically Daumas et al. [5] and Melquiond [16] implemented a technique called interval splitting. For an input interval I of variable x we generate a set of sub intervals $\{I_1, \ldots, I_n\}$ covering the interval I, e.g. $I \subseteq \bigcup_{i=1}^{n} I_i$. The proof method now verifies the formula for each sub interval I_i. If each one is correct we know $\forall i.\, x \in I_i \implies F(x)$ hence $x \in I \implies F(x)$. The proof method would be extended to accept a list of variable names and sub interval counts and then automatically generate the desired theorems and prove them.

5.3.2 Argument reduction

In the current version the precision for sin and cos is only guaranteed when the input value is in $(-\frac{\pi}{2}, \frac{\pi}{2})$ and $(-\pi, \pi)$ respectively. Otherwise the interval $(-1, 1)$ is returned, which is always correct but does not meet the precision. The Taylor series returns the correct result for all input values, but we no more guarantee that we return the upper or lower bound of the exact value and we need to compute a lot of members of the series to get the desired precision. A better approach is to use argument reduction. Since sin and cos are periodic functions we shift any real x by an integer k times π:

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

To compute k we just need to divide x by π and cut of all digits after the decimal point. For interval arithmetic we must choose only one k for both the upper and lower bound. When the difference between them is bigger than π we can already return $(-1, 1)$. Otherwise we need to figure out if both values are in one monotonic part of the function or if there is a minima or maxima between the two points. Fortunately this is just a simple case distinction.

We also need to compute π to an arbitrary precision, to have a correct k up to the last digit before the comma. In [19] is a description of this algorithm to compute k.

5.3.3 Performance enhancements

Currently the computation functions are compiled each time the proof method is called. This can be avoided when we not use the evaluation oracle of the code generator but implement our own oracle which calls the generated functions. This would save 10 seconds needed by each invokation. The disadvantage is that the trusted code base needs to be extended by the oracle's code base.

Another slowdown factor is the problem to not cut of the multiplication. The length of the result is the sum of both operands. When the power to n is computed, which is needed for the Taylor series of a transcendental function, the length of the resulting number is $n \cdot \text{bitlen}\ x$. At least for the Horner scheme we could implement a size limitation of each member after the multiplication. Unfortunately the cut off is also very expensive, since in ML only division is available. Fast bit shifting operations are not available for integer numbers.

Bibliography

[1] B. Akbarpour and L. C. Paulson. MetiTarski: an automatic prover for the elementary functions. In S. Autexier, J. Campbell, J. Rubio, V. Sorge, M. Suzuki, and F. Wiedijk, editors, *AISC/MKM/Calculemus*, volume 5144 of *Lecture Notes in Computer Science*, pages 217–231. Springer, 2008.

[2] S. Berghofer and M. Wenzel. Inductive datatypes in HOL - lessons learned in formal-logic engineering. In Y. Bertot, G. Dowek, A. Hirschowitz, C. Paulin, and L. Théry, editors, *Theorem Proving in Higher Order Logics, 12th International Conference, TPHOLs'99, Nice, France, September, 1999, Proceedings*, volume 1690 of *Lecture Notes in Computer Science*, pages 19–36. Springer, 1999.

[3] R. V. Carlone. Patriot Missile Defense: Software Problem Led to System Failure at Dhahran, Saudi Arabia. Report B-247094, US General Accounting Office: Information Management and Technology Division, 1992.

[4] A. Chaieb. *Automated methods for formal proofs in simple arithmetics and algebra.* PhD thesis, Technische Universität München, Germany, April 2008.

[5] M. Daumas, D. Lester, and C. Muñoz. Verified real number calculations: A library for interval arithmetic. *IEEE Transactions on Computers*, 58(2):226–237, 2009.

[6] J. D. Fleuriot. On the mechanization of real analysis in Isabelle/HOL. In M. Aagaard and J. Harrison, editors, *Theorem Proving in Higher Order Logics, 13th International Conference, TPHOLs 2000, Portland, Oregon, USA, August 14-18, 2000, Proceedings*, volume 1869 of *Lecture Notes in Computer Science*, pages 145–161. Springer, 2000.

[7] F. Haftmann and T. Nipkow. A code generator framework for Isabelle/HOL. Technical Report 364/07, Department of Computer Science, University of Kaiserslautern, August 2007.

[8] T. C. Hales. Formal Proof. *Notices of the American Mathematical Society*, 55:1370–1380, 2008.

[9] J. Harrison. Metatheory and Reflection in Theorem Proving: A Survey and Critique. Technical Report CRC-053, SRI Cambridge, Millers Yard, Cambridge, UK, 1995.

[10] J. Harrison. *Theorem Proving with the Real Numbers.* PhD thesis, University of Cambridge, 1996.

[11] J. Harrison. Formal Proof – Theory and Practice. *Notices of the American Mathematical Society*, 55:1395–1406, 2008.

[12] S. P. Jones. *Haskell 98 Language and Libraries: The Revised Report.* Cambridge University Press, May 2003.

[13] K. Königsberger. *Analysis 1.* Springer, Berlin, 5 edition, 2001.

[14] A. Krauss. *Automating Recursive Definitions and Termination Proofs in Higher-Order Logic.* PhD thesis, Technische Universität München, Germany, 2009.

[15] J. L. Lions and et al. Ariane 5: Flight 501 failure. Report by the inquiry board, European Space Agency, 1996.

[16] G. Melquiond. Proving bounds on real-valued functions with computations. In A. Armando, P. Baumgartner, and G. 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.

[17] R. Milner, M. Tofte, and D. Macqueen. *The Definition of Standard ML.* MIT Press, Cambridge, MA, USA, 1997.

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

[19] K. C. Ng. Argument reduction for huge arguments: Good to the Last Bit. Technical report, SunPro, 1992.

[20] T. Nipkow, L. C. Paulson, and M. Wenzel. *Isabelle/HOL: A Proof Assistant for Higher-Order Logic.* Springer, 2002. LNCS Tutorial 2283.

[21] S. Obua. *Flyspeck II: The Basic Linear Programs.* PhD thesis, Technische Universität München, Germany, 2008.

[22] V. R. Pratt. Anatomy of the pentium bug. In *TAPSOFT '95: Proceedings of the 6th International Joint Conference CAAP/FASE on Theory and Practice of Software Development*, pages 97–107, London, UK, 1995. Springer-Verlag.

[23] C. Varadi. A framework for real number calculations in Isabelle/HOL. Diploma thesis, Technische Universität München, 2008.

[24] M. Wenzel. *The Isabelle/Isar Reference Manual.* Technische Universität München.