Abstract

We discuss a modification of Standard ML to represent numbers of type `real` by pairs of floating point bounds, rather than by single approximating floating-point values. This approach yields a clean semantics for computer arithmetic, allowing the latter to be encompassed by formal methods, without sacrificing executability. We describe an experimental implementation using the Edinburgh Standard ML core language interpreter.

Formalizing SML's arithmetic by tracking cumulated error bounds

Rob Harley[1]

May 1992

[1]this work was done while the author was at the University of Edinburgh and the École Polytechnique

Contents

Chapter 1

Introduction

1.1 Standard ML

ML ("Meta Language") was initially designed as a notation for describing automatic proof techniques, at Edinburgh University during the 1970s. It was subsequently developed into a general-purpose programming language but retained its formal foundations. During the 1980s it served as a platform for theoretical and experimental work on flexible type systems and parameterized modules. From 1986, reports published by the Laboratory for Foundations of Computer Science at Edinburgh gave formal semantics for a standardised dialect of the language, Standard ML, incorporating these two advances. The reports set out which grammar phrases are legal in SML and give their meaning in terms of precise mathematical objects. Then in 1990 *The Definition of Standard ML* [MTH90] was published externally, followed by *Commentary on Standard ML* [MT90].

The formalism on which SML is based is more than just a rigorous definition because properties of the language are amenable to proof. Indeed some desireable properties have been proved (see [MT90]), inspiring confidence that the design of the language is bug-free. However this author believes that the treatment of real numbers and the built-in operations on them is unsatisfactory. While it is unreasonable to expect an implementation to provide exact arithmetic, we can have no trust in an implementation that blindly approximates the ideal operations of the formalism using floating-point operations, with no regard for the consequences of rounding error. This report suggests that a practical solution is to use floating-point arithmetic but to maintain bounds on the error accumulated during a sequence of calculations so that an exception can be raised when the underlying precision is not sufficient to guarantee correct results.

1.2 The problems of floating-point

A floating-point operation closely approximates its mathematical counterpart; for instance floating-point "addition" approximates (correct) addition as follows. Given two floating-point arguments, if their sum can be represented exactly by a floating-point number then this is returned and the operation is exact; otherwise the floating-point number that is nearest to their sum is returned—the *rounding error* is the difference between this *rounded* result and the true sum. Subtraction, multiplication and division are similar. But in the case of some higher functions such as `sin` (sine) or `exp` (exponential) it would be cumbersome to ensure that rounding error was always the least possible, and implementations may return a floating-point number which is close, but not actually the nearest, to the true answer.

Although rounding errors are tiny for each operation, it is well known that rounding errors can be magnified in a sequence of floating-point operations, leading to disastrously bad results. For instance the small relative error[1] introduced in rounding `1.0 + 1E20` to `1E20` becomes 100% relative error when we subtract `1E20`, giving `0.0` instead of the correct `1.0`. This illustrates the phenomenon known as *cancellation*: when we subtract two roughly equal numbers (or add two numbers of similar magnitude and opposite signs) the operation is exact, but previous small relative errors in the arguments lead to large relative error in the answer. However multiplication or division do not lead to an explosion of relative error even in the presence of rounding.

On the other hand small *absolute* errors in the arguments of a multiplication or a division can be magnified in the answer e.g., rounding `1.0 / 3.0` to `0.333333333333333` introduces a tiny absolute error that becomes large when we multiply by `1E20`. However addition and subtraction do not cause explosion in absolute error.

[1] *absolute* error is defined as the (signed) difference between an approximating value and the correct value. *Relative* error is absolute error divided by the correct value, provided the latter is non-zero. If it is zero, then any approximating value other than zero itself is said to have "infinite" relative error.

1.3 Possible solutions

One trivial (and useless) solution would be for all arithmetic operations involving `real`s to raise exceptions. If instead we restricted ourselves to the four basic operations `+`, `-`, `*` and `/` then we could manipulate rational numbers represented exactly by pairs of integers—one each for the numerator and the denominator. This improvement is of little value as soon as we attempt to use any higher functions; for instance there is no rational representation of $\sqrt{2}$.

A further step would be to build an algebraic manipulation system into the compiler. By representing numbers symbolically it is possible to perform exact operations much as a mathematician would do by hand. Such a solution would be powerful, but catastrophically inefficient in both time and space. Even simple comparisons could involve complex proofs e.g., deciding by algebraic "symbol pushing" the output of:

```
if sin (4.0 * (arctan 1.0)) < 1.0
  then print "foo"
  else print "bar"
;
```

is definitely non-trivial. This example is handled in four operations of the interval arithmetic method described below.

1.4 Interval arithmetic

The proposed solution provides a realistic implementation of arithmetic consistent with the formal definition of SML. We represent each `real` by a pair of bounds similar to a mathematical interval of the real line. For example the pair `(GE 3.0, LT 3.25)` (i.e., the interval $[3.0, 3.25)$) represents some arbitrary real number between 3.0 (included) and 3.25 (excluded); note that both bounds must be floating-point numbers as they are stored internally.

Now in order to manipulate a `real` we simply operate on the bounds of the interval that represents it. To do this efficiently we employ a seldom-used facility available on floating-point hardware which conforms to the IEEE Standard for Binary Floating-Point Arithmetic [IEE85]: it is possible to switch the *rounding mode* from *round-to-nearest* to *round-to-positive* (or *round-to-negative*) so that floating-point operations no longer return the floating-point number nearest to the exact answer but the next floating-point number upwards (or downwards).

So if we wish to add two `reals`, we add their lower bounds in *round-to-negative* mode, and then add their upper bounds in *round-to-positive* mode. Thus we get two floating-point numbers which we store as a new interval representing the sum of the arguments. Now because of our careful rounding we are sure that this new interval records any accumulated error. The same principle applies for subtraction, but with modified operations on the bounds: for instance the lower bound of the difference is obtained as the lower bound of the first argument minus the *upper* bound of the second argument (with rounding down). Multiplication and division are treated similarly except that we have to check the signs of the bounds to decide which pairs to multiply or divide.

Now before performing each operation we can check the bounds of its arguments to ensure that the operation is valid throughout the range. For instance an attempt to divide by an interval which contains zero will result in an exception, since the implementation cannot guarantee that the exact number represented by the interval is different from zero. In a similar manner, to decide whether a number (represented by an interval) is less than another we can first check if the two overlap. If so then we should raise an exception; otherwise return `true` or `false`.

Finally, the computed bounds can be used at any stage to monitor the accuracy of results produced.

1.5 Implementation

The approach described here was tested using the Edinburgh Standard ML core language interpreter on a Sun 4; its hardware and OS provide a fast, full implementation of IEEE Standard arithmetic (see [Sun91]). The four basic operations on intervals were added in two stages: by low-level modification of the underlying abstract machine to access the IEEE rounding modes, and by SML datatypes and functions. The basic operations were then exploited to build interval arithmetic versions of the `sqrt` and `arctan` functions. The latter function provided some interesting challenges and a novel method was applied to obtain an excellent approximating polynomial giving provably small relative error with a small number of terms; the method employed a procedure for modifying a standard Chebyshev series which is usually truncated to approximate `arctan` with small *absolute* error.

Chapter 2

Arithmetic in SML

2.1 How SML's arithmetic is defined

2.1.1 Preliminary

This section describes how [MTH90][1] defines SML's built-in arithmetic functions. These are `+`, `-`, `*`, `/`, `sqrt`, `sin`, `cos`, `arctan`, `exp` and `ln`; the first three are "overloaded" to operate on values of type `int` or `real` according to context—however we are only interested in their operation on `real`s. We use distinct typefaces to distinguish between various kinds of object e.g., `exp` is an ML variable whose value is the exponentiation function, whereas *exp* is a symbol representing expressions in the formal semantics of SML.

2.1.2 What the Definition says

Now suppose we start up the New Jersey SML compiler, and at its "`- `" prompt we type "`2.0 + 3.0;`" (the "`;`" just tells SML that we would like it to process the "`2.0 + 3.0`" before we type any more). Here's what happens:

```
- 2.0 + 3.0;
val it = 5.0 : real
```

Let's examine how we could have guessed the result from the Definition. First of all the string "`2.0 + 3.0`" is really a convenient replacement for

[1] following Milner and Tofte in [MT90], we will refer to [MTH90] as "the Definition"

"`+ (2.0, 3.0)`" which is recognized as an instance of the grammar phrase *exp atexp* with *exp* being the variable called `+` and *atexp* being the pair of constants `(2.0, 3.0)`.

Next the typechecker makes sure the phrase is not nonsense. Briefly, it finds that `+` has type `real * real -> real` (in other words it is a function which takes a pair of real numbers as argument and returns a `real` result) and that `(2.0, 3.0)` has type `real * real` (it is a pair of `real`s, as required by `+`). Thus the application of `+` to `(2.0, 3.0)` will give a result of type `real`.

The meaning of SML programs is given in terms of mathematical objects using a set of inference rules. Each rule is written as a set of premises over a statement, and tells us that whenever the premises are all satisfied we can deduce that the statement is true. The rule which deals with the phrase we are interested in is rule (116) from the Definition:

$$\frac{E \vdash \mathit{exp} \Rightarrow b \qquad E \vdash \mathit{atexp} \Rightarrow v \qquad \mathrm{APPLY}(b, v) = v'}{E \vdash \mathit{exp}\ \mathit{atexp} \Rightarrow v'} \tag{116}$$

The symbols this rule employs need explaining: E represents an *environment* which is used to look up the values of ML's variables, and b is any element of the set of *basic values*. In our example b is the mathematical addition function (found as the value of the variable `+` by looking in E), and v is the pair of reals `(2.0, 3.0)`. Note that we write `+` for both the variable `+` and the function which is its value; we also write `2.0` for the number which is the value of the *special constant* called `2.0`, although such a subtle distinction need not delay the reader!

The rule tells us that when *exp* evaluates to a basic value b, and *atexp* evaluates to some value v then *exp atexp* evaluates to v', where $\mathrm{APPLY}(b, v) = v'$.

Plugging in the values from our example, we find that *exp atexp* evaluates to $\mathrm{APPLY}(b, v) = \mathrm{APPLY}(\texttt{+}, \texttt{(2.0, 3.0)})$. Appendix D of the Definition describes the behaviour of APPLY. Quoting the relevant item for the four basic operations:

- The arithmetic functions `/`, `*`, `+`, `-` all return the results of the usual arithmetic operations, or exception packets ...

So as we expected, $\mathrm{APPLY}(\texttt{+}, \texttt{(2.0, 3.0)}) = \texttt{5.0}$.

Other operations are defined in a similarly succinct manner, by just mapping them to their mathematical equivalents. For instance (writing $\mathrm{APPLY}(\texttt{sqrt}, \mathrm{r})$

in the more natural form `sqrt`(r) etc.):

- `sqrt`(r) returns the square root of r, or the packet [`Sqrt`] if r is negative.
- `sin`(r), `cos`(r) return the result of the appropriate trigonometric functions.

...

2.2 What goes wrong with floating-point implementations

So far so good; we can try typing some more examples (taking care to use "~" to negate things as SML always assumes that "-" typed in by the user is a *binary* minus and therefore expects two arguments):

```
- ~1.5 * 1000.0;
val it = ~1500.0 : real
- 3.75 / 2.5;
val it = 1.5 : real
- 1.0 / 3.0;
val it = 0.333333333333333 : real
```

But this last result is wrong! Computer users are used to "roughly correct" arithmetic and may not notice this sort of error. An implementation which employs floating-point is useable for some purposes. But if we want to think of arithmetic operations as mathematical functions over the reals, then we require at the very least that an implementation raise an exception to warn when it cannot continue to give the results required by the Definition.

We certainly don't want to modify the Definition to describe floating point in mathematical terms: floating-point is a means, not an end. It is fast, but a result with unknown error bounds is totally meaningless, so we must modify the implementation. But let us first show how easily floating-point goes badly wrong; the result for `1.0 / 3.0` above had small absolute and relative errors, but it is easy to give examples where either form of error is much larger. For instance:

```
- it - 0.33333333333333;
val it = 3.33066907387547E~15 : real
```

The value of `it` is that of the previous expression evaluated at top level (here `0.333333333333333`); subtracting an almost equal value causes massive cancellation—the result is large relative error and a string of meaningless digits. Other problems occur due to the limited range of floating point representation i.e., results whose magnitudes are too large can overflow even when they occur as intermediate values in a calculation whose result is well within the range. Perhaps the simplest example of the occurence of overflow is in a large constant:

```
- 1E400;

uncaught exception BadReal
```

Finally, numbers with very small magnitude underflow to `0`, causing unwanted `Division` exceptions—except that in the case of New Jersey SML a bug causes the compiler to quit:

```
- 3E~308/2.0;
underflow should not trap
dea.polytechnique.fr%
```

(The line beginning `dea...` is a shell prompt). We move to the Edinburgh SML core language interpreter, where underflow does not cause a crash! Here overflow doesn't raise an exception, but returns an IEEE "signed infinity" as shown below. Unfortunately, division by `0.0` does too, so `1.0 / 0.0 = 1E400` returns `true`: the implementation disagrees with fundamental mathematical notions, and the consequence is nonsense.

The last example below shows that higher functions such as `sin` and `arctan` suffer from similar weaknesses: the result of the calculation should be `0.0`. Although absolute error is small, the relative error is infinite.

```
- 3E~308 / 2.0;
> 1.5E~308 : real
- 1E~400;
> 0.0 : real
- 1E308 * 2.0;
> Infinity.0 : real
- 1.0 / 0.0 = 1E400;
> true : bool
- sin (4.0 * (arctan 1.0));
> 1.224606353822377E~16 : real
```

Chapter 3

The mathematical aspects of interval arithmetic

> Interval arithmetic takes only about twice as long as ordinary arithmetic, and it provides truly reliable error estimates. Considering the difficulty of mathematical error analyses, this is indeed a small price to pay. ...the final estimates obtained with interval arithmetic will tend to be pessimistic; and iterative numerical methods have to be redesigned if we want to deal with intervals. The prospects for effective use of interval arithmetic look very good, however, and efforts should be made to increase its availability. — *Donald E. Knuth*, [Knu81].

3.1 Basic operations on intervals

Following [Knu81], we introduce symbols to designate mathematical operations analagous to floating-point operations in round-to-negative and round-to-positive modes. For instance we require that $x \mathbin{\nabla\!\!\!\!\!+} y$ be less than or equal to $x + y$. This property characterizes functions $\nabla\!\!\!\!\!+$ which are sufficient for our purposes—we ignore the practical requirement that $x \mathbin{\nabla\!\!\!\!\!+} y$ be "close" to $x + y$ etc. The necessary properties are:

$$\forall x, y \in \mathbf{R}. \qquad x \mathbin{\nabla\!\!\!\!\!+} y \leq x + y \ \wedge\ x \mathbin{\triangle\!\!\!\!\!+} y \geq x + y \tag{3.1}$$

$$\forall x, y \in \mathbf{R}. \qquad x \mathbin{\nabla\!\!\!\!\!-} y \leq x - y \ \wedge\ x \mathbin{\triangle\!\!\!\!\!-} y \geq x - y \tag{3.2}$$

$$\forall x, y \in \mathbf{R}. \qquad x \mathbin{\triangledown} y \leq xy \ \wedge \ x \mathbin{\triangle} y \geq xy \tag{3.3}$$

$$\forall x \in \mathbf{R}, \forall y \in \mathbf{R} \setminus \{0\}. \qquad x \mathbin{\triangledown} y \leq x/y \ \wedge \ x \mathbin{\triangle} y \geq x/y \tag{3.4}$$

3.1.1 Addition

Let $I_1 = (x_l, x_u)$ and $I_2 = (y_l, y_u)$ be two (open) intervals: $x_l < x_u \ \wedge \ y_l < y_u$. In the case of addition let I_3 be the "interval sum" $I_1 + I_2$ defined by:

$$I_1 + I_2 \stackrel{\text{def}}{=} \{z \in \mathbf{R} : \ \exists x \in I_1, \exists y \in I_2.\ z = x + y\} \tag{3.5}$$

We wish to define an operation $\oplus$ on intervals using just the operations of 3.1–3.4 such that $I_3 \subseteq I_1 \oplus I_2$. The method suggested in the Introduction is:

$$I_1 \oplus I_2 \stackrel{\text{def}}{=} (x_l \mathbin{\triangledown} y_l, x_u \mathbin{\triangle} y_u) \tag{3.6}$$

The proof of $I_3 \subseteq I_1 \oplus I_2$ is easy. For any $z \in I_3$ we have $z = x + y$ for some $x \in I_1$ and $y \in I_2$. Then $x_l < x < x_u$ and $y_l < y < y_u$ so clearly $x_l + y_l < x + y < x_u + y_u$. Then by property 3.1, $x_l \mathbin{\triangledown} y_l < x + y < x_u \mathbin{\triangle} y_u$; thus $z \in I_1 \oplus I_2$.

3.1.2 Other basic operations

The operation $\ominus$ satisfying $I_1 - I_2 \subseteq I_1 \ominus I_2$ is defined in a similar manner:

$$I_1 \ominus I_2 \stackrel{\text{def}}{=} (x_l \mathbin{\triangledown} y_u, x_u \mathbin{\triangle} y_l) \tag{3.7}$$

The operation $\otimes$ satisfies $I_1 \times I_2 \subseteq I_1 \otimes I_2$, but it is slightly more complicated:

$$\begin{aligned} I_1 \otimes I_2 \quad \stackrel{\text{def}}{=} \quad & (\min(x_l \mathbin{\triangledown} y_l, x_l \mathbin{\triangledown} y_u, x_u \mathbin{\triangledown} y_l, x_u \mathbin{\triangledown} y_u), \\ & \max(x_l \mathbin{\triangle} y_l, x_l \mathbin{\triangle} y_u, x_u \mathbin{\triangle} y_l, x_u \mathbin{\triangle} y_u)) \end{aligned} \tag{3.8}$$

Finally $I_1 \oslash I_2 \stackrel{\text{def}}{=} I_1 \otimes I_2^{-1}$ whenever I_2^{-1} exists:

$$I_2^{-1} \stackrel{\text{def}}{=} (1 \mathbin{\triangledown} y_u, 1 \mathbin{\triangle} y_l) \text{ when } y_l \leq y_u < 0 \ \vee \ 0 < y_l \leq y_u \tag{3.9}$$

3.1.3 Cost of implementation

Clearly floating-point addition in round-to-negative mode implements ∇_{+}, except that if a large negative result overflows there may be NO floating-point number less than it. This problem is easily fixed by introducing the pseudo-number "-Infinity.0" to represent an arbitrarily large negative bound; likewise floating-point addition in round-to-positive mode implements Δ_{+}, and "Infinity.0" is introduced to handle overflow in large positive results.

In our implementation the introduction of these pseudo-numbers involves no loss of efficiency, because we use the IEEE infinities which the floating-point hardware handles directly. However careful attention is necessary to ensure we never encounter the nonsense illustrated above. We certainly must disallow division by `0.0` (which just involves setting a hardware flag to cause a trap as soon as such an operation is attempted). Our implementation never adds infinities having opposite signs or subtracts infinities having the same sign—in fact another hardware trap called "Invalid" is used for a bit of *defensive programming*, ensuring we never do so accidentally.

Now we claim that interval operations can be implemented so that they are roughly twice as expensive as ordinary floating-point arithmetic. In fact each basic interval operation can be implemented with just a few sign checks and two floating-point operations, except in one (rare) case when four are necessary.

Firstly, reference to equations 3.6 and 3.7 shows that this claim is trivially justified for $\diamond_{+}$ and $\diamond_{-}$. However a naive implementation of $\diamond_{\times}$ using 3.8 would require eight multiplications. By considering the signs of the bounds this count can be reduced: for instance if all the bounds are positive, $I_1 \diamond_{\times} I_2 = (x_l \nabla_{\times} y_l, x_u \Delta_{\times} y_u)$. Two multiplications are sufficient except when $x_l < 0 < x_u$ and $y_l < 0 < y_u$ —four are necessary in this case (see Question 4.2.2–24 of [Knu81]). In applications of interval arithmetic, bounds on a number will usually be of the same sign; thus the average number of multiplications will not be much greater than two.

Sign checks reduce interval division to two floating-point divisions in *all* of the six possible cases:

$$I_1 \diamond_{\div} I_2 = \begin{cases} \left.\begin{array}{ll} (x_l \nabla_{\div} y_l, x_u \Delta_{\div} y_u) & \text{if } x_l \le x_u < 0 \\ (x_l \nabla_{\div} y_l, x_u \Delta_{\div} y_l) & \text{if } x_l < 0 \le x_u \\ (x_l \nabla_{\div} y_u, x_u \Delta_{\div} y_l) & \text{if } 0 \le x_l \le x_u \end{array}\right\} & \text{and } 0 < y_l \le y_u; \\ \left.\begin{array}{ll} (x_u \nabla_{\div} y_l, x_l \Delta_{\div} y_u) & \text{if } x_l \le x_u < 0 \\ (x_u \nabla_{\div} y_u, x_l \Delta_{\div} y_u) & \text{if } x_l < 0 \le x_u \\ (x_u \nabla_{\div} y_u, x_l \Delta_{\div} y_l) & \text{if } 0 \le x_l \le x_u \end{array}\right\} & \text{and } y_l \le y_u < 0. \end{cases}$$

Thus the claim that interval operations are around twice as expensive as floating-point arithmetic is justified.

3.2 Interval square roots

We move on to the square root function—and find that it is surprisingly easy to implement an interval version of it. To approximate the square root of a floating-point number k, the usual algorithm is to pick a (positive) x_0 and converge towards the positive root of $x^2 - k = 0$ with a few iterations of Newton's method: $x_{n+1} = (x_n + \frac{k}{x_n})/2$. Now rather than adapting this algorithm to manipulate intervals directly, we proceed as follows. Given an interval (k_l, k_u), we first check the sign of k_l: if $k_l < 0$ then raise the exception `Sqrt`. Otherwise we use a routine (described below) to get tight upper bounds l and u on $\sqrt{k_l}$ and $\sqrt{k_u}$ respectively, and return the interval $(k_l \mathbin{\bigtriangledown} l, u)$; the case $k_l = 0$ (and possibly $k_u = 0$) is handled in the obvious way.

The routine for calculating a tight upper bound x satisfying $x \geq \sqrt{k}$ works in two steps; first we use the IEEE floating-point `sqrt` function to set $y = \mathtt{sqrt}(k)$. Next we do *one* Newton iteration (adapted to round upwards) on y:

$$x = (y \mathbin{\triangle\!\!\!\!+} k \mathbin{\triangle} y) \mathbin{\triangle} 2$$

The reasons for doing this are two-fold:

- if y is "close" to $\sqrt{k}$ then x is too;
- but most importantly, $x \geq (y + k/y)/2 \geq \sqrt{k}$. The first inequality holds because of our chosen rounding direction, while the second is proved (with k non-negative and y positive) by $(y + k/y)/2 \geq \sqrt{k} \;\Leftrightarrow\; y^2 + k - 2y\sqrt{k} \geq 0 \;\Leftrightarrow\; (y - \sqrt{k})^2 \geq 0$ which certainly holds.

That's it! We have an interval square root algorithm which uses two floating point `sqrt`s and a few basic operations.

3.3 Trigonometric and exponential functions

The remaining SML arithmetic functions are `sin`, `cos`, `arctan`, `exp` and `ln`. We concentrate on polynomial approximation; we show why truncated Taylor series are not particularly good for our purpose, and introduce the far superior Chebyshev series. However Chebyshev series are typically truncated to provide small absolute error, whereas floating-point (as opposed to fixed-point) representation attempts to minimise relative error. A novel method for adapting some Chebyshev series is derived. Contrary to other methods, which are typically iterative numerical algorithms, we derive explicit expressions for series which can be truncated to yield excellent approximating polynomials in the sense of relative error. To the knowledge of the author, these expressions have not previously been published in the relevant literature.

3.3.1 Taylor series

The Taylor series for arctan is:

$$\arctan x = \sum_{n=0}^{\infty} (-)^n \frac{x^{2n+1}}{2n+1} \qquad (-1 < x \leq 1) \tag{3.10}$$

To calculate $\arctan x$ for $x \notin (-1, 1]$ the following identities can be used:

$$\begin{aligned} \arctan x &= \frac{\pi}{2} - \arctan(1/x) \qquad (x > 0) & (3.11) \\ \arctan(-x) &= -\arctan x & (3.12) \end{aligned}$$

However the series converges slowly unless $|x|$ is much less than 1; the slowest convergence occurs for $x = 1$: truncating after nine terms gives an error of magnitude greater than 0.0276, or about 1 in 36. To guarantee a small truncation error throughout $[0, 1]$ say, a huge number of terms would be necessary. Taylor series for sin, cos and exp converge much better than this, but we have chosen to concentrate on the arctan function as its Taylor series behaves particularly badly, forcing us to explore less elementary methods of calculation (ln would have been similarly challenging).

Figure 1 shows the absolute error curve when we approximate arctan by truncating its Taylor series after nine terms (the little "teeth" along the curve are due to the graphing package!) The error is $O(x^{19})$ so for small x the approximation is excellent—we would have to change the scale to distinguish any error for $|x| \leq \frac{3}{4}$. But as x approaches 1 the error grows rapidly, so the maximum error is very large.

3.3.2 Optimal polynomial approximation

We now define the notion of distance between functions on an interval. This will allow us to measure how good an approximation to a given function is. The (L_∞) norm $\|g\|_\infty$ of a function g over some interval I measures the maximum deviation of $g(x)$ from 0 for $x \in I$. It is given by:

$$\|g\|_\infty \stackrel{\text{def}}{=} \sup_{x \in I} |g(x)|$$

The distance between a function f (such as arctan) and an approximating polynomial f' over an interval I (such as $[-1,1]$) is the *largest* magnitude of absolute error over I:

$$\mathrm{d}(f, f') \stackrel{\text{def}}{=} \|f'(x) - f(x)\|_\infty$$

As shown in Figure 1, the distance $d(\arctan, f')$ (where f' is the truncated Taylor series) is very large, though the biggest error only occurs for x near ± 1.

Much better polynomial approximations (of the same degree) can be obtained by choosing coefficients so as to spread the error more evenly throughout the interval $[-1,1]$. In fact the following holds for any continuous function over a closed bounded interval: for each n there is a unique best approximating polynomial of degree n, characterized by the so called *equal-ripple* property. The latter property states that the error assumes at least $n+2$ extreme values of equal magnitude and alternating sign as x ranges over the chosen interval. Intuitively, if the extrema were not arranged in such a manner we could perturb the coefficients to decrease the biggest extrema in exchange for an increase in the lesser extrema (look ahead to Figure 4 for the error curve of an approximation which "almost" satisfies the equal-ripple property).

A precise and more general formulation is given by the following theorem due to Chebyshev, quoted from [Fik68].

> Let $u(x)$ denote a function continuous in a closed, finite interval $[a,b]$, and let $v(x)$ denote a function continuous and nonzero in $[a,b]$. Let V_n denote the set of polynomials of degree $\leq n$. There exists a unique polynomial $P_n^*(x)$ in V_n such that
>
> $$\max_{x \in [a,b]} \left| \frac{P_n^*(x)}{v(x)} - u(x) \right| = \min_{P_n(x) \in V_n} \max_{x \in [a,b]} \left| \frac{P_n(x)}{v(x)} - u(x) \right|$$
>
> Let $P_n(x)$ denote a polynomial in V_n. Then $P_n(x)$ is $P_n^*(x)$ if and only if there exist $N \geq n+2$ points in $[a,b]$,

$$x_1^* < x_2^* < x_3^* < \cdots < x_N^*$$

such that

$$\frac{P_n(x_k^*)}{v(x_k^*)} - u(x_k^*) = (-1)^k \mu^* \quad \text{for} \quad k = 1, 2, 3, \ldots, N,$$

where

$$|\mu^*| = \max_{x \in [a,b]} \left| \frac{P_n(x)}{v(x)} - u(x) \right|.$$

In the expressions above, we can replace $u(x)$ by the function $f(x)$ to be approximated and replace $v(x)$ by 1. Then clearly the expression $P_n(x)/v(x) - u(x) = P_n(x) - f(x)$ and the statement is about absolute error. Instead we can replace $u(x)$ by 1 and $v(x)$ by $f(x)$, and then $P_n(x)/v(x) - u(x) = (P_n(x) - f(x))/f(x)$, so the equal-ripple property also holds for relative error.

For a proof see Chapter 3 of [Ric64] (Fike suggests [Ach56]).

(see Figure 1)

3.3.3 Chebyshev polynomials

The Chebyshev polynomials T_n are defined by:

$$T_n(x) \stackrel{\text{def}}{=} \cos(n \arccos x) \qquad (x \in [-1,1]) \tag{3.13}$$

(see Figures 2 and 3 for the graphs of $T_3(x)$ and $T_6(x)$ on $x \in [-1,1]$). To verify that this equation indeed defines a family of polynomials, let $\theta \stackrel{\text{def}}{=} \arccos x$ and note that:

$$\begin{aligned}
T_0(x) &= \cos 0 = 1 && (3.14)\\
T_1(x) &= \cos\theta = x && (3.15)\\
T_{n+1}(x) &= \cos(\theta + n\theta)\\
&= \cos\theta\cos(n\theta) - \sin\theta\sin(n\theta)\\
&= 2\cos\theta\cos(n\theta) - \cos(n\theta - \theta)\\
&= 2xT_n(x) - T_{n-1}(x) && (3.16)
\end{aligned}$$

So by induction $T_n(x)$ is a polynomial of degree n for all $n \geq 0$. Furthermore for $n \geq 1$ the leading coefficient of $T_n(x)$ is 2^{n-1}. From Definition 3.13 it is clear that $\sup_{x\in[-1,1]} |T_n(x)| = 1$, so for any $n \geq 1$ the normalized polynomial $\tilde{T}_n(x) = T_n(x)/2^{n-1}$ deviates from 0 by 2^{1-n} over $[-1,1]$. In fact this is the least possible for any monic polynomial (one whose leading coefficient is 1) of degree n.

Finally, as x varies from -1 to 1, $\arccos x$ varies from $-\pi$ to 0. Then by 3.13, for $n \geq 1$ we have that $T_n(x)$ alternates between ± 1 exactly $n+1$ times.

(see Figures 2 and 3)

3.3.4 Chebyshev series

Now if we have a *Chebyshev series* of the form:

$$\arctan x = \sum_{n=0}^{\infty} a_n T_n(x) \tag{3.17}$$

then truncating to the N-term polynomial $f'(x) = \sum_{n=0}^{N-1} a_n T_n(x)$ introduces an absolute error $\epsilon_N(x) = -\sum_{n=N}^{\infty} a_n T_n(x)$. For analytic functions such as arctan the a_n decrease rapidly in magnitude, so $\epsilon_N(x)$ is dominated by $-a_N T_N(x)$. This term alternates $N+1$ times between $\pm a_N$, and since $f'(x)$ is of degree $N-1$, the equal-ripple property is nearly satisfied (as in Figure 4).

It is certainly not obvious that an approximation for which the equal-ripple property "almost" holds is a good approximation in any sense. However from [Fik68] again:

> Let $P_n(x)$ denote a polynomial of degree $\leq n$; and let $x_1, x_2, \ldots, x_{n+2}$ denote points such that
>
> $$a \leq x_1 < x_2 < \cdots < x_{n+2} \leq b$$
>
> If
>
> $$\frac{P_n(x_k)}{v(x_k)} - u(x_k) = (-1)^k \mu_k, \qquad k = 1, 2, 3, \ldots, n+2,$$
>
> where $\mu_1, \mu_2, \ldots, \mu_{n+2}$ denote nonzero numbers having like signs, then
>
> $$\min_{1 \leq k \leq n+2} |\mu_k| \leq \max_{x \in [a,b]} \left| \frac{P_n^*(x)}{v(x)} - u(x) \right| .$$

Refering back to Figure 4, choose $\mu_1, \mu_2, \ldots, \mu_{n+2}$ to be the values of x corresponding to the extrema of the curve. Then the above theorem says that the best polynomial approximation (of the same degree) has maximum error at least equal to the height of the smallest extremum of the curve. However it clearly has maximum error no greater than the height of the largest extremum of the graph. Thus, informally, a sufficent condition for an approximation to be "good" is that its error curve alternates $n+2$ times, and has its smallest extremum almost as high as its largest.

3.3.5 Orthogonality

Given a function we wish to approximate, how do we obtain the Chebyshev coefficients a_n for a series such as 3.17? To answer this question we borrow some ideas from the theory of orthogonal polynomials.

A family $\{\pi_n(x)\}$ of polynomials are *orthogonal* with respect to the weight function $w(x)$ on $[a,b]$ iff:

$$\int_{x=a}^{b} w(x)\pi_m(x)\pi_n(x)\,dx = c_n\delta_{mn} \tag{3.18}$$

for some constants c_n (δ_{mn} is 1 for $m=n$, 0 otherwise). Now assume an expansion $f(x)=\sum_{n=0}^{\infty} a_n\pi_n(x)$ exists. Formally, we see that integrating $\int_{x=a}^{b} w(x)f(x)\pi_n(x)\,dx$ will pick out $c_n a_n$. So we can find the a_n by:

$$a_n = \frac{1}{c_n}\int_{x=a}^{b} w(x)f(x)\pi_n(x)\,dx \tag{3.19}$$

The Chebyshev polynomials are orthogonal on $[-1,1]$ with respect to $w(x)=\frac{1}{\sqrt{1-x^2}}$:

$$\begin{aligned}
\int_{x=-1}^{1}\frac{T_m(x)T_n(x)}{\sqrt{1-x^2}}\,dx &= \int_{\theta=-\pi}^{0}\cos(m\theta)\cos(n\theta)\,d\theta \qquad (\theta=\arccos x) && (3.20)\\
&= \begin{cases} 0 & \text{if } m\neq n;\\ \pi & \text{if } m=n=0,\ (\int_{\theta=-\pi}^{0} 1\,d\theta);\\ \frac{\pi}{2} & \text{if } m=n\neq 0,\ (\frac{1}{2}\int_{\theta=-\pi}^{0} 1+\cos(2m\theta)\,d\theta). \end{cases} && (3.21)
\end{aligned}$$

Hence the a_n are given by:

$$\begin{aligned}
a_0 &= \frac{1}{\pi}\int_{x=-1}^{1}\frac{f(x)}{\sqrt{1-x^2}}\,dx && (3.22)\\
a_n &= \frac{2}{\pi}\int_{x=-1}^{1}\frac{f(x)T_n(x)}{\sqrt{1-x^2}}\,dx \qquad (n>0) && (3.23)
\end{aligned}$$

In general these integrals cannot be evaluated; but for many standard functions, expressions for the integrals are known. A Chebyshev series for arctan is:

$$\arctan x = k \sum_{n=0}^{\infty} (-)^n \frac{\lambda^n}{2n+1} T_{2n+1}(x) \tag{3.24}$$

where $k = 2\sqrt{2} - 2$ and $\lambda = 3 - 2\sqrt{2}$. Truncating after N terms gives $\epsilon_N(x) = -k \sum_{n=N}^{\infty} (-)^n \frac{\lambda^n}{2n+1} T_{2n+1}(x)$. Each term $a_n T_{2n+1}(x)$ has magnitude bounded by $k \frac{\lambda^n}{2N+1}$, so $|\epsilon_N(x)|$ is bounded by the geometric series sum $\frac{k\lambda^N}{(1-\lambda)(2N+1)}$.

We choose $N = 9$ and find this bound to be between 6.7×10^{-9} and 6.8×10^{-9}. The graph of Figure 4 shows the absolute error curve scaled up by 10^9. In fact $\epsilon \in (6.6 \times 10^{-9}, 6.7 \times 10^{-9})$, so the bound is fairly tight. Compare this with the error for the truncated Taylor series!

3.3.6 The problem of relative error

Figure 5 shows the relative error for the truncated Chebyshev series, scaled by 10^9. Recall that relative error is absolute error over the correct value; so for small x, and hence small values of $\arctan x$, absolute error is greatly magnified to give large relative error. The same phenomenon occurs near roots of any (continuous) function we wish to approximate.

For a function such as cos on $[-\frac{\pi}{4}, \frac{\pi}{4}]$ there is no need for any adaptation of the standard Chebyshev series. This is because cos has range $[\frac{1}{\sqrt{2}}, 1]$ on the stated interval so absolute and relative error are roughly proportional (to within 42%); an approximation which nearly satisfies the equal-ripple property for absolute error will also nearly satisfy it for relative error. In applications the number of terms retained is chosen such that relative error due to truncation is the same size as relative error due to floating-point rounding, but anyway the latter varies by a factor of 2 between the beginning and the end of a *binade* (the interval between two successive powers of two).

(see Figures 4 and 5)

3.3.7 Adapting the Chebyshev series for relative error

As discussed above, best approximations in the sense of relative error behave similarly to best approximations in the sense of absolute error; in particular the equal-ripple property is retained. Unfortunately the techniques which make use of this fact are iterative numerical algorithms (e.g., the method of ascent or Remes' exchange algorithm) which converge towards excellent approximating polynomials, but do not yield algebraic expressions for the approximations, nor in general do they *prove* tight bounds on relative error. Here we present a method which does just this in some important cases.

Given any function f with an explicit Chebyshev series, truncation of the series does not give a polynomial approximation with small relative error *near roots of the function*. But it is possible to obtain such a polynomial near each root r for which the following property holds: for some $d > 0$, $\forall x.\ |x| < d \Rightarrow f(r-x) = f(r+x)$ i.e., when the function is locally symmetric about the root. Now shift the root to the origin if necessary so the shifted function is odd. We can then choose an interval about the origin small enough for the function to be roughly linear and scale to $[-1, 1]$. The function is then of the form:

$$f(x) = \sum_{n=0}^{\infty} a_n T_{2n+1}(x) \tag{3.25}$$

As before we shall concentrate on arctan on $[-1, 1]$ although other functions such as sin or sinh (not built into SML) are amenable to the same treatment.

In our earlier discussion of arctan we could have restricted attention to the interval $[0, 1]$ by employing symmetry of the arctan function. We did not do so; for arctan is odd, so that its Chebyshev series on an interval symmetric about the origin is also odd. The adaptation proposed here gives a modified series of the form:

$$\arctan x = x \sum_{n=0}^{\infty} b_n T_{2n}(x) \tag{3.26}$$

The leading factor x is crucial to what follows. The arctan function is "roughly linear" on $[-1, 1]$ in the sense that $(\arctan x)/x$ is "roughly constant": its range is $[\frac{\pi}{4}, 1]$ so the maximum is about 27% more than its minimum. We take the infinite sum on the RHS of 3.26 and truncate it to get an excellent polynomial approximation to $(\arctan x)/x$ in the sense of absolute error; we then multiply the polynomial by x to get a very good approximation to $\arctan x$ *in the sense of relative error*.

The coefficients b_n are obtained by the following definition, obtained by *telescoping* series 3.25 using 3.16.

$$b_n \stackrel{\text{def}}{=} \begin{cases} \sum_{i=0}^{\infty}(-)^i a_i & \text{if } n = 0; \\ 2\sum_{i=0}^{\infty}(-)^i a_{n+i} & \text{if } n > 0. \end{cases} \tag{3.27}$$

provided these sums converge. For arctan, $a_n = k(-)^n \frac{\lambda^n}{2n+1}$ (k and λ as in equation 3.24). The new expansion for $\arctan x$ is:

$$\arctan x = xk \sum_{n=0}^{\infty}(-)^n(2-\delta_{0n}) \sum_{m=n}^{\infty} \frac{\lambda^m}{2m+1} T_{2n}(x) \tag{3.28}$$

Both the inner and outer sums converge for all $x \in [-1,1]$, as can be seen by bounding the absolute values of the terms by geometric series. Figure 6 shows the absolute error for this series truncated to nine terms. Note that the maxima and minima have magnitudes which vary almost linearly with $\arctan x (\approx x)$. Thus upon division by $\arctan x$, we get a relative error curve very close to the optimum equal-ripple curve—see Figure 7. In particular the worst relative error is less than 13% of that obtained with the non-adapted Chebyshev series (Figure 5). Note also that the extrema are of very similar magnitude, hence the best possible polynomial approximation of of like (or lower) degree has maximum error only slightly less than that which we have obtained[1].

3.3.8 Adapted Chebyshev expansions for sin, sinh and arcsin

The following standard expansions are derived in Section 4.3 of [Sny66]:

$$\sin(zx) = 2\sum_{n=0}^{\infty}(-)^n J_{2n+1}(z) T_{2n+1}(x)$$

$$\sinh(zx) = 2\sum_{n=0}^{\infty} I_{2n+1}(z) T_{2n+1}(x)$$

where J_λ and I_λ are the Bessel functions of the first kind and the modified Bessel functions, respectively. Note that we would pick a fixed z ($z = 1$ say), and let x vary so that the Bessel functions would be evaluated just once (the

[1] unfortunately we can't supply the numerical value as the VAX at the Ecole Polytechnique, where our data are stored, is down...

Taylor series for the Bessel functions could be used for this purpose as they converge very quickly unless z is large).

Clearly both of these expansions satisfy the conditions we require to obtain the adapted expansions which can be truncated for small relative error. The new expansions are:

$$\sin(zx) = 2x \sum_{n=0}^{\infty} (-)^n (2 - \delta_{0n}) \sum_{i=0}^{\infty} J_{2(i+n)+1}(z) T_{2n}(x)$$

$$\sinh(zx) = 2x \sum_{n=0}^{\infty} (2 - \delta_{0n}) \sum_{i=0}^{\infty} (-)^i I_{2(i+n)+1}(z) T_{2n}(x)$$

In [Fik68], exercise 16 of Chapter 6 yields the following expansion for $\arcsin x$:

$$\arcsin x = \frac{4}{\pi} \sum_{k=0}^{\infty} \frac{1}{(2k+1)^2} T_{2k+1}(x)$$

Note that in this case the a_n's do not decrease rapidly. Neither do we get small error upon truncation; this is because the truncation gives an almost optimal polynomial, but even the optimal polynomial approximations to arcsin are not good! Exercises 17 and 18 of [Fik68] show that any polynomial which approximates $\arcsin x$ in $[-1, 1]$ with absolute error less than 2^{-25} must be of degree at least 7529 (seven thousand five hundred and twenty nine).

The new expansion (of doubtful utility) is thus:

$$\arcsin x = \frac{4x}{\pi} \sum_{n=0}^{\infty} (2 - \delta_{0n}) \sum_{i=0}^{\infty} (-)^i \frac{1}{(2(i+n)+1)^2} T_{2n}(x)$$

(see Figures 6 and 7)

Chapter 4

Implementation

Edinburgh SML is an interpreter for the core language of SML i.e., the language with its modular facilities removed. It is designed around a "Functional Abstract Machine" (the FAM) which in practice is emulated by a small, fast byte-code interpreter written in C. Most of the system is written in SML and compiled to about 500k of byte-codes for the FAM.

4.1 Byte-codes and exceptions

We first added 8 new byte-codes using the spare numbers 237–244:

```
OpFPlusPo
OpFPlusNe
OpFDiffPo
OpFDiffNe
OpFTimesPo
OpFTimesNe
OpFDividePo
OpFDivideNe
```

The FAM was modified so that before starting to execute byte-codes it checks that directed rounding is available and enables the hardware traps for division by zero and "Invalid" operations (such as `Infinity.0 * 0.0`). Then the exception handlers were modified to take account of the new sources of floating-point exceptions (C exceptions at this level). Each was implemented so that its execution leads to the following steps: the "Inexact" hardware flag is cleared and the rounding direction is set as appropriate; then the required operation is carried out and the rounding is set back to round-to-nearest; finally the Inexact flag is tested and a pair is returned. The latter pair contains a boolean flag indicating whether the operation was exact or not, along with the answer.

4.2 The four basic operations

A new exception called `Inexact` was defined to have a real parameter; eight new operations on floating-point numbers were defined: `\+/`, `/+\`, `\-/`, `/-\`, `\*/`, `\*/`, `\//` and `//\`. Each corresponds to one of the rounded operations we have described—for instance `\+/` implements lower floating-point addition ($\nabla\mkern-13mu+$). Furthermore, whenever one of these operations involves rounding error, it raises the Inexact exception along with the rounded result.

The method of implementation was to use the new opcodes within a wrapper function which raises the Inexact exception when appropriate—see the beginning of Appendix A for details. For example on a Sun 4:

```
- 1.0 \// 2.0;
> 0.5 : real
- 1.0 \// 3.0;
Exception traceback
...
Exception: Inexact 0.3333333333333333 raised
- 1.0 //\ 3.0;
Exception traceback
...
Exception: Inexact 0.3333333333333334 raised
```

Then the four basic operations on intervals were defined using these directed rounding floating-point operations on the bounds. The methods of Section 3.1 were implemented using the optimisations suggested in 3.1.3. However in order to make best use of the underlying hardware floating-point, interval bounds were marked as being open or closed, with the SML datatypes shown here:

```
datatype LBound = GE of real | GT of real;
datatype UBound = LT of real | LE of real;
```

The constructors `GE` and `GT` represent closed and open lower bounds respectively, whereas `LT` and `LE` represent open and closed upper bounds. We illustrate the utility of this enhancement in the case of lower bounds during an interval addition. The operation `<+>` implements $\oplus$ with one improvement; suppose we wish to add two intervals with closed lower bounds, `(GE a, ...)` and `(GE b, ...)` say. Then we add `a \+/ b`; if the result `c` is exact, then we return an interval of the form `(GE c, ...)`. However if the operation is not exact (detected by handling the `Inexact` exception), then rounding down occured and we can safely return `(GT c, ...)`.

In the case where one or both of the original bounds are open, then clearly we return `(GT c, ...)` whether rounding occurs or not. The listing of Appendix A shows the implementation described so far. The parts relevant to lower bound during addition are:

```
local
  fun splitl (GE t) = (true, t) | splitl (GT t) = (false, t)
  ...
  fun lower (oper, (eqx, x), (eqy, y)) =
    if eqx andalso eqy
    then (GE (oper (x, y))) handle Inexact z => GT z
    else (GT (oper (x, y))) handle Inexact z => GT z
  ...
in
  fun op <+> ((xl, xu), (yl, yu)) =
    let val xls = splitl xl ...
        val yls = splitl yl ...
    in (lower (op \+/, xls, yls), ...) end
end;
```

The local function `splitl` simply removes the constructor `GE` or `GT` on a lower bound and replaces it by a flag to indicate closedness; `lower` is a generic function for lower bounds which operates as just outlined for the case of addition. Then the definition of `<+>` is simple indeed: it just tells `lower` to operate on the two lower bounds using the `\+/` operation. Of course the upper bound of the result is obtained in a symmetric manner with `splitu`, `upper` and `/+\`.

Some examples follow; `mk` is a one-line function which given a number `x` builds the interval `(GE x, LE x)`.

```
- (mk 1.0) </> (mk 3.0);
> (GT 0.3333333333333333,LT 0.3333333333333334) : LBound * UBound
- it <+> it;
> (GT 0.6666666666666666,LT 0.6666666666666667) : LBound * UBound
- (GE 1.0, LE 1.0) </> (GT 0.0, LE 4.0);
```

```
> (GE 0.25,LT Infinity.0) : LBound * UBound
- (GE 1.0, LE 1.0) </> (GE 0.0, LE 4.0);
Exception traceback
...
Exception: Quot  raised
```

4.3 Interval square root function

The implementation of the interval square root function `isqrt` is as described in Section 3.2 with a few small adjustments to take account of open and closed bounds. We use `unG` and `unL` to throw away the constructors of the argument intervals. This means that we sometimes return a result with closed bounds when a more complicated procedure could have given open ones.

```
val unG = fn (GE t) => t | (GT t) => t;
val unL = fn (LE t) => t | (LT t) => t;
...
local fun iter (k, x) =
    (((x /+\ ((k //\ x) handle Inexact t => t)) handle Inexact t => t)
     //\ 2.0) handle Inexact t => t
in fun isqrt (x, y) =
  let val l = unG x val u = unL y
  in if l < 0.0 then raise Sqrt
    else
      (if l = 0.0 then x
       else (GE (l \// (iter (l, sqrt l)))) handle Inexact t => (GT t),
       if u = 0.0 then y else LE (iter (u, sqrt u)))
  end
end
;
```

The constant $\sqrt{2}$ is used in the Chebyshev series for arctan. We define it by:

```
(* square root of 2 *)
val r2 = isqrt (mk 2.0);
```

The value of `r2` is printed as `(GT 1.414213562373095, LE 1.414213562373095)`. But the bounds are not in fact equal, the built-in printing function just does not show all the significant digits. If we set `val (GT l, LE u) = r2` then `u-l` is 2^{-52}. The floating-point format used is double precision IEEE which has 52 binary digits of mantissa (plus an implicit

1 digit in front), so this is the tightest possible interval for $\sqrt{2}$. We don't always get the tightest bounds though, as `isqrt (mk 3.0)` is in the same binade but gives an interval of width 2^{-51}.

4.4 Arctangent function

The first (experimental) implementation uses a truncated Taylor series, and is only for the interval $[0, 1]$. Given a number `n` of terms to use, we build a stack of intervals representing the terms of the series, by recursion; then we make an interval which bounds the sum of the discarded terms. The latter operation is easy as we have an alternating series with terms decreasing in magnitude, so the sum is between zero and the value of the first discarded term. Finally take this error interval and add on the terms, from the last back to the first, using `<+>`. Adding in this order has the advantage that we avoid adding many small numbers to a large one, and so reduce rounding error.

```
fun err (x, y) =
  (if unG x > 0.0 then GE 0.0 else x,
   if unL y < 0.0 then LE 0.0 else y)
;

(* Taylor series *)
local fun terms f (p as (st, tup)) n =
    if n = 0 then (err st)
    else st <+> (terms f (f tup) (n-1))
  fun f (v, r, c) = (v </> (mk c), (v <*> r, r, c+2.0))
in fun iatan1 x n =
  let val x2 = x <*> x
  in terms f (x, (<~> x <*> x2, <~>x2, 3.0)) n
  end
end
;
```

We can attempt to estimate π using this. For instance $\pi = 4 \arctan 1$. However as we know the Taylor series converges very badly for arguments close to 1, and even 50 terms is useless (see below). The following *Machin-like* formula gives high precision with just seven terms:

$$\pi = 48 \arctan \frac{1}{18} + 32 \arctan \frac{1}{57} - 20 \arctan \frac{1}{239}$$

```
- (mk 4.0) <*> (iatan1 (mk 1.0) 50);
> (GT 3.12159465259101,LT 3.161198612987051) : LBound * UBound
- fun ipi n =
  (mk 48.0)<*>(iatan1((mk 1.0)</>(mk 18.0))n)<+>
  (mk 32.0)<*>(iatan1((mk 1.0)</>(mk 57.0))n)<->
  (mk 20.0)<*>(iatan1((mk 1.0)</>(mk 239.0))n)
;
= = = = > val ipi = fn : int -> (LBound * UBound)
- ipi 7;
> (GT 3.141592653589793,LT 3.141592653589794) : LBound * UBound
```

Finally, a version of arctan using the modified Chebyshev expansion was implemented. The coefficients b_n are calculated in similar manner to the Taylor series except that the error term is found as the sum of a geometric series which bounds the true error. For efficiency we calculate the b_n once and store the coefficients in a list:

```
(* Tchebycheff series coefficients *)
local
  fun coef (pow, lam, n) 0 = [err (pow </> (mk n) </> ((mk 1.0) <-> lam))]
    | coef (pow, lam, n) k =
    let val li = (coef (pow <*> lam, lam, n+2.0) (k-1))
    in (pow </> (mk n) <+> (hd li))::li end
  val lam = (mk 3.0) <-> (mk 2.0) <*> r2
  fun chang [] _ = []
    | chang (h::t) 1 = h::(chang t ~1)
    | chang (h::t) ~1 = (<~> h)::(chang t 1)
in val coli = chang (coef (mk 1.0, lam, 1.0) 50) 1
end
;
```

Then `val coli` is:

```
[(GT 1.063912033650002, LT 1.063912033650002),
 (GT ~0.06391203365000214, LT ~0.06391203365000193),
 (GT 0.006721075232065375, LT 0.006721075232065415),
 (GT ~0.0008336249274935132, LT ~0.0008336249274935059),
 (GT 0.0001121058013011393, LT 0.0001121058013011406),
 (GT ~1.58222704989079E~5, LT ~1.582227049890768E~5),
 ...
] : (LBound * UBound) list
```

Then the `iatan2` function takes as parameter an interval and a number of terms to use. We use the fact that arctan is monotonic to get tighter bounds than would be possible by operating directly on the interval argument. For if its bounds are not very close, then calculating high powers by interval multiplication would cause the bounds on the powers to spread apart rapidly. However if we work out the arctan of the argument's lower bound while rounding down, and the arctan of the upper bound while rounding up, then we neatly avoid the divergence phenomenon.

Note also that the approximating polynomial is never used in the standard form $a_0 + a_1 x + a_2 x^2 + \ldots + a_N x^N$. To avoid numerical error involved in converting to this form, we evaluate the $T_{2n}(x)$ directly using a method similar to Horner's method for standard-form polynomials. Further details can be found in Section 3.1 of [Sny66].

Chapter 5

Conclusion

This work was motivated by the author's desire to give precise mathematical meanings to SML programs that use real numbers. SML is renowned, deservedly, as the paradigm of formally defined programming languages. Unfortunately however, in its implementation it falls somewhat short of this ideal. In this particular case it has been shown that in relation to its treatment of arithmetic it leads to unsatisfactory if not unsafe results, in situations which might well arise in everyday use of the system by many who place considerable trust in its reliability. The objective has therefore been to show how the formal purity of SML can be maintained without sacrificing efficiency in an inordinate way. This is done by adopting an interval arithmetic approach to floating-point arithmetic, which requires recourse to approximation methods permitting bounds on error to be determined. It is shown that naive algorithms can be modified in ways to reduce the volume of computation required to implement high precision interval operations. In particular, a method of adapting the Chebyshev polynomial was developed to this end. This interval arithmetic approach incorporating the Chebyshev adaptation has been implemented on the Edinburgh SML core language interpreter.

5.1 Possibilities for further work

Ideally the safety of interval arithmetic should be exploited to the full by replacing the floating-point arithmetic of an SML interpreter or compiler, in as transparent a manner as feasible. To this end it would be necessary to modify the language so it parsed constants in program text or users' input as intervals. The remaining arithmetic functions would need to be implemented, along the lines described in this report. A few other comparatively simple operations (such as comparisons, conversions to or from integers and printing) would require modification.

An SML implementation which used interval arithmetic in a transparent manner would be an ideal platform for formal work, for instance as part of a system for executing specifications in the executable subset of Extended ML.

5.2 Thanks

The author thanks Dr. Don Sannella for friendly supervision and many useful suggestions as to the content of the report, especially when the emphasis was moved from Extended ML to the present interval arithmetic work.

Bibliography

[Ach56] N. I. Achieser. *The Theory of Approximation.* Ungar, 1956.

[Fik68] C. T. Fike. *Computer Evaluation of Mathematical Functions.* Prentice-Hall Series in Automatic Computation. Prentice-Hall, Inc., 1968.

[IEE85] IEEE, Inc. *IEEE Std 754-1985. IEEE Standard for Binary Floating-Point Arithmetic*, 1985.

[Knu81] Donald E. Knuth. *Seminumerical Algorithms*, volume 2 of *The Art of Computer Programming.* Addison-Wesley Publishing Company, 2nd edition, 1981.

[MT90] Robin Milner and Mads Tofte. *Commentary on Standard ML.* The MIT Press, 1990.

[MTH90] Robin Milner, Mads Tofte, and Robert Harper. *The Definition of Standard ML.* The MIT Press, 1990.

[Ric64] John R. Rice. *The Approximation of Functions.* Addison-Wesley Publishing Company, 1964.

[Sny66] Martin A. Snyder. *Chebyshev Methods in Numerical Approximation.* Prentice-Hall Series in Automatic Computation. Prentice-Hall, Inc., 1966.

[Sun91] Sun Microsystems, Inc. *Numerical Computation Guide*, 1991.

Appendix A

SML code for the four basic operations on intervals

```
(* RJH 17th May 1992: interval arithmetic *)


(**
 ** Basic floating point ops with rounding towards positive/negative;
 ** use byte-codes implemented only on Sun 4 and Archimedes
 **)

(* This exception is raised whenever the result of a operation cannot
 * be represented (exactly) by a floating-point value; it has a real
 * parameter which contains the result rounded up or down to the
 * nearest fp value (except when overflow occurs...)
 *)
exception Inexact of real;

local
  (* Make local functions which just use the new opcodes for +, -, *, /
   * with rounding to +ve/-ve (this part is EdML specific); each
   * fun is declared by a fixity directive and its identifier, then
   * the opcode number, then the size of the tuple it needs as
   * argument.  All low-level stuff is in the C source for the FAM:
   * see bcodes.h for a list of opcodes, and interp.c/system.c/fam.c
```

```
   * for their implementation
   *)
  nonfix FPlusPo 237 2
  nonfix FPlusNe 238 2
  nonfix FDiffPo 239 2
  nonfix FDiffNe 240 2
  nonfix FTimesPo 241 2
  nonfix FTimesNe 242 2
  nonfix FDividePo 243 2
  nonfix FDivideNe 244 2

  (* Then for the type-checker, we define each fun as itself (!)
   * but with full type information.  NOT recursive definition.
   *)
  fun FPlusPo (x: real, y: real): (bool * real) = FPlusPo (x, y)
  fun FPlusNe (x: real, y: real): (bool * real) = FPlusNe (x, y)
  fun FDiffPo (x: real, y: real): (bool * real) = FDiffPo (x, y)
  fun FDiffNe (x: real, y: real): (bool * real) = FDiffNe (x, y)
  fun FTimesPo (x: real, y: real): (bool * real) = FTimesPo (x, y)
  fun FTimesNe (x: real, y: real): (bool * real) = FTimesNe (x, y)
  fun FDividePo (x: real, y: real): (bool * real) = FDividePo (x, y)
  fun FDivideNe (x: real, y: real): (bool * real) = FDivideNe (x, y)

  (* Each opcode returns a pair with a boolean flag (to indicate
   * exactness) and a real result.  This function is used to convert
   * to a cleaner form: return the result if it is exact, else raise
   * the Inexact exception with the rounded result.
   *)
  fun wrap f = fn p =>
    let val (exactp, x) = f p
    in if exactp then x else raise Inexact x end
in
  (* Here we declare the funs that will be visible globally
   * When o is +, -, * or / then /o\ and \o/ are the
   * rounding-to-positive and rounding-to-negative versions resp.
   *)
  infix 6 /+\ \+/ /-\ \-/
  infix 7 /*\ \*/ //\ \//

  (* Here we define the funs by using ``wrap'' on the new opcodes.
   * If x and y are reals and z = x o y then x /o\ y should:
   *  - return z, if it has a floating point representation
   *  - or raise Inexact z', where z' is the least fp number > z if it exists
   *  - or raise Inexact Infinity.0
```

```
   * Likewise x \o/ y should:
   *  - return z, if it has a fp representation
   *  - or raise Inexact z', where z' is the largest fp number < z if it exists
   *  - or raise Inexact ~Infinity.0
   * But when o = / and y = 0.0 we should raise Quot, when x or y are infinite
   * ...
   *)
  val op /+\ = wrap FPlusPo
  val op \+/ = wrap FPlusNe
  val op /-\ = wrap FDiffPo
  val op \-/ = wrap FDiffNe
  val op /*\ = wrap FTimesPo
  val op \*/ = wrap FTimesNe
  val op //\ = wrap FDividePo
  val op \// = wrap FDivideNe
end;


(* These are the datatypes for lower and upper interval bounds *)
datatype LBound = GE of real | GT of real;
datatype UBound = LT of real | LE of real;

(* An interval has type LBound * UBound i.e., a pair of bounds;
 * e.g., the interval [1.0, 2.5) becomes (GE 1.0, LT 2.5).
 * It represents a single number which we don't know precisely,
 * although it can be thought of as a set of possible values of this
 * unknown number.
 *)

(* The forms BigPos and BigNeg can each be thought of as ``some real
 * bound too large for floating-point representation'' (possibly a
 * different number each time they are used).
 * It is fastest to implement them with the IEEE infinities; but then
 * overflow can't be trapped so we have to be very careful.  For
 * instance the typechecker would allow (GT Infinity.0) as an LBound
 *)
(* Excuse the horrible and implementation-dependant way of getting
 * Infinity.0 using overflow---you can't write it as you would 1.25
 *)
local val Infinity = 1E1000 (* anything that causes overflow *)
in val BigNeg = GT (~Infinity) val BigPos = LT Infinity end;

(* Here we define the elementary operations on intervals:
 * e.g. (GE 1.0, LE 1.0) </> (GT 0.0, LE 4.0) = (GE 0.25, Infinity.0)
```

```
 * since the first interval represents 1.0, the second represents some
 * (strictly) +ve number less than or equal to 4.0, and the third is
 * some number greater than or equal to 0.25
 * There is no checking against args being invalid, e.g. (GT 1.0, LE 1.0)
 * or (GE 2.0, LE 1.0)
 *)
infix 6 <+> <->;
infix 7 <*> </>;

local
  fun splitl (GE t) = (true, t) | splitl (GT t) = (false, t)
  fun splitu (LE t) = (true, t) | splitu (LT t) = (false, t)
  fun lower (oper, (eqx, x), (eqy, y)) =
    if eqx andalso eqy
    then (GE (oper (x, y))) handle Inexact z => GT z
    else (GT (oper (x, y))) handle Inexact z => GT z
  fun upper (oper, (eqx, x), (eqy, y)) =
    if eqx andalso eqy
    then (LE (oper (x, y))) handle Inexact z => LT z
    else (LT (oper (x, y))) handle Inexact z => LT z
  datatype SIGN = PosZ | Neg
  fun sign t = if t >= 0.0 then PosZ else Neg
in
  fun op <+> ((xl, xu), (yl, yu)) =
    let val xls = splitl xl val xus = splitu xu
        val yls = splitl yl val yus = splitu yu
    in (lower (op \+/, xls, yls), upper (op /+\, xus, yus)) end
  fun op <-> ((xl, xu), (yl, yu)) =
    let val xls = splitl xl val xus = splitu xu
        val yls = splitl yl val yus = splitu yu
    in (lower (op \-/, xls, yus), upper (op /-\, xus, yls)) end
  fun op <*> ((xl, xu), (yl, yu)) =
    let val (xls as (_, xlt)) = splitl xl val (xus as (_, xut)) = splitu xu
        val (yls as (_, ylt)) = splitl yl val (yus as (_, yut)) = splitu yu
    in
      case (sign xlt, sign xut, sign ylt, sign yut) of
        (PosZ, _, PosZ, _) => (lower(op\*/,xls,yls),upper(op/*\,xus,yus))
      | (PosZ, _, _, PosZ) => (lower(op\*/,xus,yls),upper(op/*\,xus,yus))
      | (PosZ, _, _, _) => (lower(op\*/,xus,yls),upper(op/*\,xls,yus))
      | (_, PosZ, PosZ, _) => (lower(op\*/,xls,yus),upper(op/*\,xus,yus))
      | (_, PosZ, _, PosZ) =>
        let fun minl (l1, l2) =
            let val (eq1, t1) = splitl l1 val (eq2, t2) = splitl l2
            in if t1 < t2 then l1
```

```
            else if t2 < t1 then l2 else if eq1 then l1 else l2 end
        fun maxu (u1, u2) =
          let val (eq1, t1) = splitu u1 val (eq2, t2) = splitu u2
          in if t1 > t2 then u1
            else if t2 > t1 then u2 else if eq1 then u1 else u2 end
      in (minl(lower(op\*/,xls,yus),lower(op\*/,xus,yls)),
         maxu(upper(op/*\,xls,yls),upper(op/*\,xus,yus))) end
    | (_, PosZ, _, _) => (lower(op\*/,xus,yls),upper(op/*\,xls,yls))
    | (_, _, PosZ, _) => (lower(op\*/,xls,yus),upper(op/*\,xus,yls))
    | (_, _, _, PosZ) => (lower(op\*/,xls,yus),upper(op/*\,xls,yls))
    | _ => (lower(op\*/,xus,yus),upper(op/*\,xls,yls))
  end
fun op </> ((xl, xu), (yl, yu)) =
  let val (xls as (_, xlt)) = splitl xl val (xus as (_, xut)) = splitu xu
      val sxlt = sign xlt val sxut = sign xut
      val (yls as (eql, ylt)) = splitl yl val (yus as (equ, yut)) = splitu yu
  in if ylt < 0.0 orelse (eql andalso ylt = 0.0)
    then if yut > 0.0 orelse (equ andalso yut = 0.0) then raise Quot
      else case (sxlt, sxut) of
          (PosZ, _) => ((lower (op \//, xus, yus)) handle Quot =>
                          if xut = 0.0 then GE 0.0 else BigNeg,
                          upper (op //\, xls, yls))
        | (_, PosZ) => ((lower (op \//, xus, yus)) handle Quot =>
                            if xut = 0.0 then GE 0.0 else BigNeg,
                          (upper (op //\, xls, yus)) handle Quot => BigPos)
        | _ => (lower (op \//, xus, yls),
                  (upper (op //\, xls, yus)) handle Quot => BigPos)
    else case (sxlt, sxut) of
        (PosZ, _) => (lower (op \//, xls, yus),(upper (op //\, xus, yls))
                        handle Quot => if xut = 0.0 then LE 0.0 else BigPos)
      | (_, PosZ) => ((lower (op \//, xls, yls)) handle Quot => BigNeg,
                        (upper (op //\, xus, yls)) handle Quot =>
                          if xut = 0.0 then LE 0.0 else BigPos)
      | _ => ((lower (op \//, xls, yls)) handle Quot => BigNeg,
              upper (op //\, xus, yus))
  end
fun <~> (xl, xu) =
  ((fn (LE t) => GE (~t) | (LT t) => GT (~t)) xu,
   (fn (GE t) => LE (~t) | (GT t) => LT (~t)) xl)
fun mk x = (GE x, LE x)
fun printerval (l, u) =
  let fun wr x = output (std_out, x)
  in ((fn (GE t) => (wr "["; print t) | (GT t) => (wr "("; print t)) l;
      wr ", ";
```

```
        (fn (LE t) => (print t; wr "]") | (LT t) => (print t; wr ")")) u)
    end
end;
```