DEPARTMENT OF INFORMATICS

TECHNISCHE UNIVERSITÄT MÜNCHEN

Bachelor's Thesis in Informatics

Multiple Precision Floating Point Arithmetic in Isabelle/HOL

Fabian Hellauer

DEPARTMENT OF INFORMATICS

TECHNISCHE UNIVERSITÄT MÜNCHEN

Bachelor's Thesis in Informatics

Multiple Precision Floating Point Arithmetic in Isabelle/HOL

Multiple Precision Floating Point Arithmetic in Isabelle/HOL

Author:	Fabian Hellauer
Supervisor:	Prof. Tobias Nipkow
Advisor:	Fabian Immler
Submission date:	March 15, 2016

I confirm that this bachelor's thesis is my own work and I have documented all sources and material used.

München, March 15, 2016 Fabian Hellauer

Abstract

Many problems, e.g. in geometry or topology systems, need large computations with long sequences of arithmetic operations to be solved. Machine computation using hardware floating point instructions, like the one specified by the "Institute of Electrical and Electronics Engineers" (IEEE) 754 standard[1], can offer a huge speedup. However, using the IEEE-floats directly would often create the need for a complicated numerical analysis due to them being affected by round-off. However, this downside can be removed at the cost of a few extra instructions: Alongside the result, the round-off error of an addition or subtraction can be computed. Storing and using it in further operations thus makes the computation error-free. We give a simple data format in Isabelle/HOL [8] that uses this approach to provide fast algorithms for error-free addition and subtraction.

Acknowledgments

Fabian Immler provided a highly professional guidance and an incredible amount of support. His versatility for providing solutions in different aspects of scientific work inspired me to invest much time and energy aiming for a deeper understanding of automated theorem proving.

Tobias Nipkow's lectures taught me concepts of functional programming and motivated me to take the course in Semantics and apply to his research group for a bachelor's thesis.

Contents

1 Introduction

When attacking computational problems by machine, often times fixed precision numbers have to be used: Arbitrary precision numbers are very slow, mostly because they don't use the hardware floating point operations that are available to modern systems. If the finite set of these machine numbers suffice (or an approximation within their range), the use of hard-wired operations can speed up the computation by a large factor. However, they introduce the problem of round-off, which, when not handled, will affect the output's precision in complex ways.

1.1 Round-off

Round-off occurs when the result of a floating-point operation cannot be represented as a datum of the same format. This happens in particular if the size of exponent and mantissa is fixed, as it is case for floats defined by the IEEE standard for floating-point arithmetic[1]. Round-off also occurs when casting measured or infinitely precise data to such a limited precision format.

Dealing with round-off If round-off affected arithmetic is used in a long sequence of operations, the result will only approximate within a certain range. Correctness proofs for assertions to this range will require a tedious numerical analysis of the algorithms which is very complex to do formally.

Avoiding round-off Another approach is to avoid round-off altogether. However, using an implementation of infinitely precise rationals might severely slow down the code's execution due to them not making use of the hardware floating point operations that modern machines provide.

Floating Point expansions The goal of this approach is to combine the fast execution of IEEE floats with an error-free arithmetic. The idea is to execute the round-affected operations and compute the errors alongside to make the exact result available. The accumulated errors are stored alongside with the IEEE approximation for the result. Error-free addition, subtraction and multiplication can be provided within the numbers representable in this way (a finite superset to IEEE floats).

1.2 Problem statement

Isabelle[8] already provides the arbitrary precision format *real* in its *Complex-Main* library. The widely popular "IEEE-floats"[1] are modelled in an Isabelle theory *IEEE-Floating-Point/IEEE* available at the "Archive of Formal Proofs" (AFP)[12].

The task for this bachelor thesis is to use this formalization and ideas from the literature to present a "multiple precision float arithmetic" in Isabelle/HOL. This corresponds to the "floating point expansion" approach explained above, where many ideas for such formats have been proposed. It is sometimes called "multiple term"[5] strategy, but most authors use a term involving "expansion"[9][10].

1.3 Contributions

We explain different aspects of the "floating point expansion" approach and provide the data format *mpf*, which stands for "multiple precision float". A *mpf* can represent the full range of IEEE floats at their maximum precision, as opposed to floats themselves, where magnitude and absolute precision depend inversely on each other. We implement error-free addition and subtraction within these numbers.

We use the formal setting of Isabelle/HOL to specify and prove the algorithms correct, but we make sure all of them can easily be executed by adapting Isabelle's Standard ML (SML) code generation for IEEE-floats.

2 Specification in Isabelle/HOL

2.1 IEEE in Isabelle

The IEEE standard for floating point arithmetic (IEEE 754-2008)[1] is already modelled in Lei Yu's AFP entry *IEEE-Floating-Point/IEEE*. The formalization is quite general to accommodate for the many different allowed formats that arise when different bit sizes for mantissa and exponent are combined (the decimal formats are omitted). However, a strong precedence for the "binary64" format and the "roundTiesToEven" rounding mode can be observed. The operations using this format and rounding rule (called *float-format* respectively *To-nearest* in the theory) are wrapped into definitions with simpler names, e.g.

definition *plus-float* :: *float* ⇒ *float* ⇒ *float* **where** *a* + *b* = *Abs-float* (*fadd float-format To-nearest* (*Rep-float a*) (*Rep-float b*))

In the case of the format, this is justifiable by "binary64" being widely popular and hardware-implemented on most systems.

In the case of the rounding mode, the IEEE standard defines "roundTiesToEven" to be the default ([1]p. 16).

We also use this format and rounding mode, which enables us to use the code printing defined in the theory *Code-Float* from the same AFP entry. The "roundTiesToEven" rule is explicitly required for the TwoSum-properties ([7] sect. 4.3.3) and thus for all the presented algorithms.

2.2 Notation

In theory *IEEE*, the float operations use the + and − sign via
instantiation *float*

For this thesis, we want the IEEE-operations to use a different symbol, and reserve the + operator for exact number formats. Fortunately, Isabelle provides spare symbols and the **abbreviation** command, which makes sure even the output uses the new notation. We thus state at the beginning of our **theory**:

— Use another notation for the possibly inexact IEEE-operations.
abbreviation *round-affected-plus* :: *float* ⇒ *float* ⇒ *float* (**infixl** ⊕ *65*) **where**
  round-affected-plus a b ≡ *a* + *b*

abbreviation *round-affected-minus* :: *float* ⇒ *float* ⇒ *float* (**infixl** ⊖ *65*) **where**
  round-affected-minus a b ≡ *a* − *b*

Afterwards,

term $a + (b{::}\mathit{float})$

outputs

$a \oplus b$ :: *IEEE.float*

2.3 Expanding basic operations

The core idea is to provide an error-free form of the basic operations between IEEE-floats. Since we want the output to be floats as well, and rounding occurs for almost all input value pairs, the only way to do so is to use the round-affected IEEE operation and then computing the error, also represented as floats. For the basic operations, this will turn out to be exactly another IEEE-float.

2.3.1 Addition

We compute $a \oplus b$ together with y the error value. y will have the sign $-$ or $+$ corresponding to whether $a \oplus b$ is above resp. below the exact mathematical result of $a + b$. It can be computed by the following sequence, first described by Ole Møller in 1965[6]:

```
definition TwoSum :: float ⇒ float ⇒ float × float where
  TwoSum a b = (let
    x = a ⊕ b;
    $b_v$ = x ⊖ a;
    $a_v$ = x ⊖ $b_v$;
    $b_r$ = b ⊖ $b_v$;
    $a_r$ = a ⊖ $a_v$;
    y = $a_r$ ⊕ $b_r$
    in (x, y))
```

Here, we compute a value y such that $a + b = x + y$, where $x = a \oplus b$. The following lemma states the latter:

```
lemma TwoSum-correct1: TwoSum a b = (x, y) ⟹ x = a ⊕ b
  — x is defined in the first line of TwoSum and not changed thereafter.
  by (auto simp: TwoSum-def Let-def)
```

The other property needs the preconditions that both the input and the output represent real numbers (as opposed to the special values *NaN* and $\pm\infty$). This is checked by the predicate *Finite*. We use the exact arithmetic of *real* and the conversion *Val* :: *float* => *real* from *IEEE-Floating-Point/IEEE*.

```
lemma TwoSum-correct2:
  fixes a b x y :: float
  assumes Finite a
  assumes Finite b
```

assumes *Finite* $(a \oplus b)$
assumes *out*: $(x, y) = \mathit{TwoSum}\ a\ b$
shows $\mathit{Val}\ a + \mathit{Val}\ b = \mathit{Val}\ x + \mathit{Val}\ y$
sorry

We assume the lemma by **sorry** for this thesis. Notice that a formal proof using the theorem prover Coq[2] is available online[11]

2.3.2 Subtraction

For $a - b$, we can compute:

definition *TwoDiff* :: *float* ⇒ *float* ⇒ *float* × *float* **where**
TwoDiff a b = *TwoSum a* (*float-neg b*)

To drop the additional negation step, we could instead perform:

$\mathit{TwoDiff}'\ a\ b = (\mathit{let}$
$x = a \ominus b;$
$b_v = x \ominus a;$
$a_v = x \oplus b_v;$
$b_r = b_v \ominus b;$
$a_r = a \ominus a_v;$
$y = a_r \oplus b_r$
$\mathit{in}\ (x, y))$

according to Shewchuk[10]. Note that we still have a + on the right side of the equation to make the "unevaluated sum approach" work:

lemma *TwoDiff-correct2*:
fixes *a b x y* :: *float*
assumes *Finite a*
assumes *Finite b*
assumes *Finite* $(a \ominus b)$
assumes *out*: $(x, y) = \mathit{TwoDiff}\ a\ b$
shows $\mathit{Val}\ a - \mathit{Val}\ b = \mathit{Val}\ x + \mathit{Val}\ y$
oops

2.3.3 Optimizations

Dekker[3] shows that in some situations, sequences of three operations suffice for + and −. In order to not further increase the amount of unproven lemmas (or complicate them), we drop this optimization and the *TwoDiff'* sequence for this thesis. This keeps the possibilities for errors at a minimum.

2.4 MPF definitions

The new data format is designed to implement the idea of storing all the errors as an unevaluated sum. It is defined as follows:

— Define the "Multiple Precision Float"
type-synonym *mpf* = *float* × *float list*
fun *approx* :: *mpf* ⇒ *float* **where**
  approx (*a*, *es*) = *a*
fun *errors* :: *mpf* ⇒ *float list* **where**
  errors (*a*, *es*) = *es*

where the tuple of a *float* and a *float list* should be seen together as a non-empty float list, ordered by decreasing magnitude. The approximation *approx* is just stored separately to avoid having to check for an empty list on access. Its quality depends on the executed algorithms (proofs about it can be found at [10]).

The mpf's represented value is the infinite precise sum of all its components:

fun *Val-mpf* :: *mpf* ⇒ *real* **where**
  Val-mpf (*a*, *es*) = *Val a* + *listsum* (*map Val es*)

Note that multiple *mpf*s can represent the same value. Many of these are invalid if we enforce the "non-overlapping" property proposed by Shewchuk[10] on the float list. However, since it needs to read out the bit representation of the IEEE float, there is no easy way to check this condition using the AFP-formalization. We could instead decrease the number of valid representations by not allowing zero components in the list's tail:

fun *valid* :: *mpf* ⇒ *bool* **where**
  valid (*a*, *es*) = (*case Iszero a of*
    True ⇒ *es* = [] |
    False ⇒ *Finite a* ∧ *list-all* (λf. *Isdenormal f* ∨ *Isnormal f*) *es*)

where λf. *Isdenormal f* ∨ *Isnormal f* returns *False* for zero-floats:

lemma *Iszero fl* ⟹ ¬(λf. *Isdenormal f* ∨ *Isnormal f*) *fl*
  using *float-distinct*
by (*metis Isnormal-def Iszero-def is-normal-def is-zero-def order-less-irrefl*)

Since zero components don't contribute to the mpf's value, omitting them is an easy way to save storage by decreasing the list size. The problem with this property is that the algorithms don't preserve the constraint by default. As Shewchuk puts it:

"A complicating characteristic of all the algorithms for manipulating expansions is that there may be spurious zero components scattered throughout the output expansions, even if no zeros were present in the input expansions."

As he shows by an example, they even occur in the middle of output lists that provably have all non-zero-components sorted. He also states:

"Unfortunately, accounting for these zero components could complicate the correctness proofs significantly."[10]

In other words: We **could** modify the algorithms to drop the zero component on-the-fly, but the extra branch would drastically increase the proof size. We instead settle for an even weaker property:

fun *Finite-mpf* :: *mpf* ⇒ *bool* **where**
  Finite-mpf (*a*, *es*) ⟷ *Finite a* ∧ *list-all Finite es*

Using this property and the well-known *TwoSum-correct2* property described above (⟦*Finite ?a*; *Finite ?b*; *Finite* (*?a* ⊕ *?b*); (*?x*, *?y*) = *TwoSum ?a ?b*⟧ ⟹ *Val ?a* + *Val ?b* = *Val ?x* + *Val ?y*, unproven in this thesis), we will be able to prove computations error-free in the next section.
Here is a proof that valid implies Finite:

lemma *valid-finite*: *valid* (*a*, *es*) ⟹ *Finite-mpf* (*a*, *es*)
  apply (*simp split*: *bool.splits*)
  using *float-cases-finite float-distinct* **apply** *fastforce*
  by (*metis* (*no-types*, *lifting*) *Ball-set Finite-def*)

2.5 Using the option monad

We embed *TwoSum* to make sure overflow will be noticed:

definition *safe-TwoSum a b* =
  (*let r* = *TwoSum a b in*
    if Finite (*fst r*) ∧ *Finite* (*snd r*)
    then Some r
    else None)

The same for const TwoDiff:

definition *safe-TwoDiff a b* =
  (*let r* = *TwoDiff a b in*
    if Finite (*fst r*) ∧ *Finite* (*snd r*)
    then Some r
    else None)

If a value is returned, both output values are real numbers:

lemma *safe-TwoSum-finite*:
  assumes *safe-TwoSum a b* = *Some* (*s*, *e*)
  shows *safe-TwoSum-finite1*: *Finite s*
  and *safe-TwoSum-finite2*: *Finite e*
  using *assms*
  by (*auto simp*: *safe-TwoSum-def Let-def split*: *split-if-asm*)

The *TwoSum* lemmas can now be expressed like this:

lemma *safe-TwoSum-correct1*:
  safe-TwoSum a b = *Some* (*x*, *y*) ⟹ *x* = *a* ⊕ *b*
  by (*auto simp*: *safe-TwoSum-def Let-def TwoSum-correct1 split*: *split-if-asm*)

```
lemma safe-TwoSum-correct2:
  fixes a b x y :: float
  assumes Finite a Finite b Finite (a ⊕ b)
  assumes out: safe-TwoSum a b = Some (x, y)
  shows Val a + Val b = Val x + Val y
  using assms
by (auto intro!: TwoSum-correct2 simp: safe-TwoSum-def Let-def split: split-if-asm)
```

The lemmas can be easily transposed for *TwoDiff* (not shown here).

2.6 Grow-mpf

Remeber that our mpf is sorted by decreasing magnitude. When adding a single float value to an mpf, we need to propagate changes through the whole list to preserve the ordering, outputting the larger values to the head. We use Shewchuk's algorithm (grow-expansion[10]) to add a single float value to an mpf, using only the error-free transformations via *safe-TwoSum*. As his research shows, starting this transformation at the values with low magnitude increases the quality of the approximation (which is the component with highest magnitude). For our mpfs (that work with lists instead of arrays), this means we start at the tail:

```
fun safe-grow-mpf-rec :: mpf ⇒ float ⇒ mpf option where
  safe-grow-mpf-rec (a, []) f =
    do {
      (x, y) ← safe-TwoSum f a;
      Some (x, [y])
    } |
  safe-grow-mpf-rec (a, e # es) f =
    do {
      (a', es') ← safe-grow-mpf-rec (e, es) f;
      (x, y) ← safe-TwoSum a' a;
      Some (x, y # es')
    }
```

To demonstrate the *TwoSum* sequence carried out by grow-mpf, we use the following graphic:

At this point, we could implement the zero removal explained before, by modifying the last lines of the blocks:

```
fun safe-grow-mpf-rec-no-0 :: mpf ⇒ float ⇒ mpf option where
  safe-grow-mpf-rec-no-0 (a, []) f =
    do {
      (x, y) ← safe-TwoSum f a;
      if Iszero y then Some (x, []) else Some (x, [y])
    } |
  safe-grow-mpf-rec-no-0 (a, e # es) f =
    do {
      (a', es') ← safe-grow-mpf-rec-no-0 (e, es) f;
```

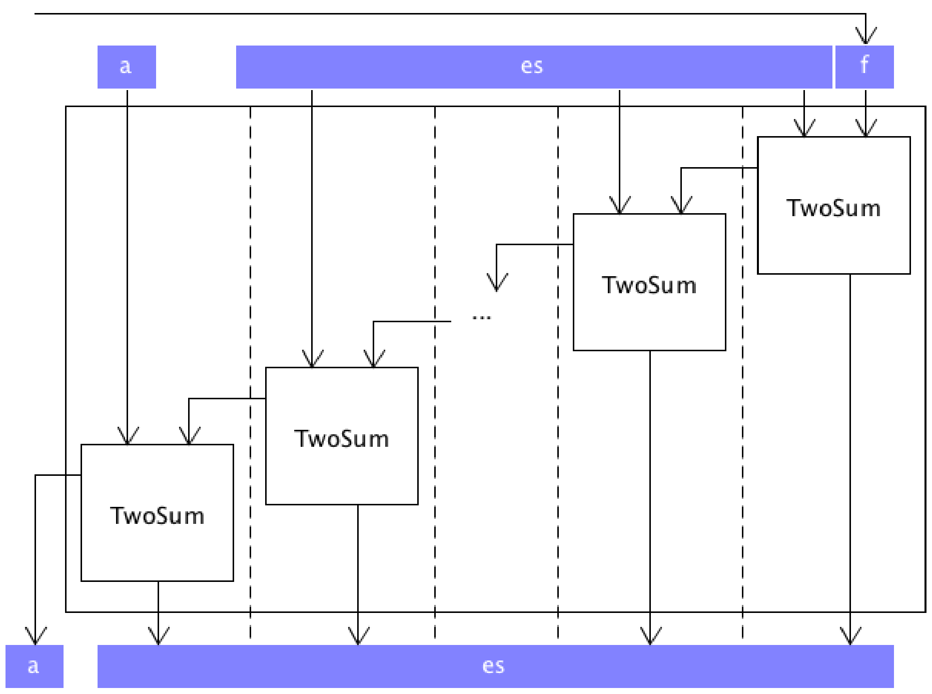

Figure 2.1: The float f is added to the mpf (a, es). TwoSum is represented by a box where the larger value, x, is output to the left side and the smaller one, y, to the bottom. On the top, the function call is passed on. The returned mpf (bottom) is built from right to left.

```
    (x, y) ← safe-TwoSum a' a;
    if Iszero y then Some (x, es') else Some (x, y # es')
  }
```

However, we don't pursue this idea further due to the mentioned problems.

We rename the induction cases, to stress the fact that in the induction step, a value is inserted at position 2 of the non-empty list:

lemmas *safe-grow-mpf-induct* = *safe-grow-mpf-rec.induct*[*case-names no-error in-between*]

2.6.1 Preserving Properties

Since *safe-TwoSum* returns *None* for non-finite floats, *safe-grow-mpf-rec mpf x* = *Some r* is a very strong assertion. We can use this to prove an important property of grow-mpf:

First, we need a recursive version of our predicate:

lemma *rec-finite*: *Finite-mpf* $(a, e \# es) \longleftrightarrow$ *Finite a* $\wedge$ *Finite-mpf* (e, es)

```
  by simp
```

Now, onto the main proof:

```
lemma preserve-finite:
  assumes safe-grow-mpf-rec mpf x = Some r
  assumes Finite x Finite-mpf mpf
  shows Finite-mpf r
using assms
proof (induction mpf x arbitrary: r rule: safe-grow-mpf-induct)
— The base case is the case where the mpf is a single float with an empty error-list:
case (no-error a f)
— We apply the definition of safe-grow-mpf-rec:
from no-error.prems(1) have do {(x, y) ← safe-TwoSum f a; Some (x, [y])} = Some r
   unfolding safe-grow-mpf-rec.simps(1) .
— Since we required the result to be some value, we can give it a name:
  then obtain x y where xy: safe-TwoSum f a = Some (x, y) and r: r = (x, [y])
   by (auto simp: bind-eq-Some-conv)
— and then delegate to the corresponding property of safe-TwoSum:
  moreover from safe-TwoSum-finite[OF xy]
   have Finite x Finite y.
  ultimately show ?case
   by simp
next
case (in-between a e es f r-full)
— This case is similar except that we need to prove more floats to be finite
  note in-between.prems(1)[simplified, unfolded bind-eq-Some-conv, simplified]
  then obtain l r where goal1: safe-grow-mpf-rec (e, es) f = Some (l, r)
   and r1: do {(x, y) ← safe-TwoSum l a; Some (x, y # r)} = Some r-full
     by blast
  then obtain l2 r2 where l2: safe-TwoSum l a = Some (l2, r2) and
    r2: (l2, r2 # r) = r-full
    using r1[unfolded bind-eq-Some-conv, simplified] by auto
  from r2 have ?case = Finite-mpf (l2, r2 # r) by simp
  moreover have Finite l2
   using safe-TwoSum-finite1[OF l2].
  moreover have Finite r2
   using safe-TwoSum-finite2[OF l2].
  moreover from in-between.IH[OF goal1 in-between.prems(2)] have list-all Finite r
   using in-between.prems(3) by auto
  ultimately
   show ?case
   by simp
qed
```

Notice that the "assignments" (←) in a Monad like *mpf option* can also be written using the ≫=-operator (*bind*) and λ-notation. This is also what Isabelle's state panel will output, e.g.

do {(*x*, *y*) ← *safe-TwoSum l a*; *Some* (*x*, *y* # *r*)} becomes

safe-TwoSum l a ≫= (λ(*x*, *y*). *Some* (*x*, *y* # *r*))

etc. We perform the next proof using this style.

Again, we first need to prove a function to be equivalent to a recursive version:

```
lemma rec-val: Val-mpf (a, e # es) = Val a + Val-mpf (e, es)
  by simp
```

Now, we're good to go:

```
theorem preserve-val:
  assumes safe-grow-mpf-rec mpf x = Some r
  assumes Finite x Finite-mpf mpf
  shows Val-mpf r = Val-mpf mpf + Val x
using assms
proof (induction mpf x arbitrary: r rule: safe-grow-mpf-induct)
case (no-error a f)
  from no-error.prems(1) have safe-TwoSum f a ≫= (λ(x, y). Some (x, [y])) = Some r
    unfolding safe-grow-mpf-rec.simps(1) .
  then obtain x y where xy: safe-TwoSum f a = Some (x, y) and r: r = (x, [y])
    by (auto simp: bind-eq-Some-conv)
  from safe-TwoSum-finite1[OF xy]
  have Finite x.
  from no-error have an: Finite a by simp
  show ?case
    using safe-TwoSum-correct2[OF ‹Finite f› an - xy] ‹Finite x›
      safe-TwoSum-correct1[OF xy]
    by (auto simp: r split: prod.split)
next
case (in-between a e es f r-full)
  note in-between.prems(1)[simplified, unfolded bind-eq-Some-conv, simplified]
  then obtain l r where goal1: safe-grow-mpf-rec (e, es) f = Some (l, r)
    and r1: safe-TwoSum l a ≫= (λ(x, y). Some (x, y # r)) = Some r-full
      by blast
  then obtain l2 r2 where l2: safe-TwoSum l a = Some (l2, r2) and
     r2: (l2, r2 # r) = r-full
     using r1[unfolded bind-eq-Some-conv, simplified] by auto
  then have Val-mpf r-full = Val-mpf (l2, r2 # r) by simp
  also have ... = Val l2 + Val-mpf (r2, r)
    by (simp add: rec-val)
  also have ... = Val l2 + Val r2 + listsum(map Val r)
    by simp
  also have ... = Val l + Val a + listsum(map Val r)
    proof −
      from in-between.prems have Finite l
        using goal1 preserve-finite by auto
      moreover have Finite a
        using in-between.prems(3) by simp
      moreover have Finite (l + a)
```

```
        using l2 safe-TwoSum-correct1 safe-TwoSum-finite1 by auto
      moreover have Val l + Val a = Val l2 + Val r2
        using safe-TwoSum-correct2[OF calculation l2].
      ultimately show ?thesis
        by simp
    qed
  finally show ?case
    using in-between goal1 rec-finite by auto
qed
```

Note that the proof tactics for *preserve-finite* and *preserve-val* are very similar (identical up to the **obtain** commands in both induction cases). They could be combined by stating

```
  shows preserve-finite: Finite-mpf r
  and preserve-val: Val-mpf r = Val-mpf mpf + Val x
```

in a single lemma with the same assumptions. However, to actually remove redundancy in the proofs, both goals would have to be combined again via

```
 unfolding atomize-conj
```

Since the second result depends partly on the first one, many fact names (or alternatively: large HOL predicates combining unrelated facts) would accumulate during the proof. To maintain readability, we stick to the two-proof-solution. Instead of stating both goals in one lemma, we collect the results afterwards:

```
lemmas safe-grow-mpf-correct =
  preserve-finite
  preserve-val
```

2.6.2 Tail recursive version

We can also implement grow-mpf in a tail-recursive way. For simplicity, we drop the overflow-check via the *option* monad for now.

```
fun grow-mpf-it :: float list ⇒ float ⇒ float list ⇒ mpf where
  grow-mpf-it [] f hs = (f, hs) |
  grow-mpf-it (e # es) f hs = (let
    (x, y) = TwoSum f e
    in grow-mpf-it es x (y # hs))
```

This transformation was comparably easy because grow-mpf only needs one linear pass as the graphic has shown.

```
fun grow-mpf-tr :: mpf ⇒ float ⇒ mpf where
  grow-mpf-tr (a, es) f = (let
    (a′, es′) = grow-mpf-it (rev es) f [];
    (x, y) = TwoSum a′ a
    in (x, y # es′))
```

An interesting realisation is that the list recursion can instead be implemented using fold:

```
fun grow-mpf-step :: float ⇒ mpf ⇒ mpf where
  grow-mpf-step f (a, es) = (let
    (x, y) = TwoSum a f
  in (x, y # es))

fun grow-by-fold :: mpf ⇒ float ⇒ mpf where
  grow-by-fold (a, es) f = foldr grow-mpf-step (a # es) (f, [])
```

We expect compilers that optimize tail recursion to also optimize *foldr*. Thus, it is no longer necessary to make the functions tail recursive if they can be expressed by fold.

We prepare an equivalence proof for *grow-mpf-tr* and *grow-by-fold* by providing some lemmas about *grow-mpf-tr* and *op* @.

```
lemma grow-it-append-accumulator:
  grow-mpf-it as f (hs @ hs') = (let
      (a, es) = grow-mpf-it as f hs
    in (a, es @ hs'))
  apply (induction as arbitrary: f hs hs')
  apply simp-all
  apply (metis (no-types, lifting) Cons-eq-appendI case-prod-beta)
  done

lemma grow-it-append-expansion:
  grow-mpf-it (as @ es) f hs = (let
    (a', es') = grow-mpf-it as f hs
  in grow-mpf-it es a' es')
  apply (induction as arbitrary: f hs)
  by (simp-all add: prod.case-eq-if)
```

Its effect on the wrapper *grow-mpf-tr* is

```
lemma grow-append-rev:
  grow-mpf-tr (a, es @ es') f = (let
    (a'', es'') = grow-mpf-it (rev es') f [];
    (a', es') = grow-mpf-it (rev es) a'' es'';
    (x, y) = TwoSum a' a
    in (x, y # es'))
     by (simp add: case-prod-beta grow-it-append-expansion)
```

In case of an increase by a singleton, this can be simplified:

```
lemma grow-snoc-rev:
  (grow-mpf-tr (a, es @ [h]) f) = (let
      (x, y) = TwoSum f h;
      (a', es') = grow-mpf-it (rev es) x [y];
      (x', y') = TwoSum a' a
    in (x', y' # es'))
  unfolding grow-append-rev[of a es [h] f]
  apply simp
  by (simp add: split-def)
```

The right part of the equation can also be written using *grow-mpf-tr*:

lemma *gm-snoc1*: $(grow\text{-}mpf\text{-}tr\ (a,\ es\ @\ [h])\ f) = (let$
    $(x,\ y) = TwoSum\ f\ h;$
    $(a',\ es') = grow\text{-}mpf\text{-}tr\ (a,\ es)\ x$
   $in\ (a',\ es'\ @\ [y]))$
  by (*induction es arbitrary*: *a*) (*simp-all add*: *case-prod-beta grow-it-append-expansion*)

2.6.3 Generality

Only the two defining properties (lemmas TwoSum-correct1 and TwoSum-correct2) of the TwoSum-method are needed for the mpfs properties of error-free computation. Any software or hardware format that enables such a possibility to add two values and "record" the precise error is in principle suited for error-free computations using these recursive algorithms. As four our *Main-Thesis.float*s, additional lemmas about the format like exact rounding would be needed to make use of Shewchuk's "nonoverlapping" property that he proves to be preserverved by them in [10]. Since this property is needed to make assertions about the first components approximation quality or the maximum expansion length, these cannot be provided as HOL-facts yet.

2.7 Further Operations and Constants

We provide a test if the mpf represents 0:

definition $IsZero\text{-}mpf\ mpf \longleftrightarrow Iszero\ (approx\ mpf) \wedge errors\ mpf = []$

From CodeFloat[12] we use the definition:

definition *One* :: *float* **where**
$One = Abs\text{-}float\ (0,\ bias\ float\text{-}format,\ 0)$
declare *One-def*[*code del*]

Together with *Plus-zero* and *Minus-zero*, we can define:

definition *Plus-zero-mpf* :: *mpf* **where**
 $Plus\text{-}zero\text{-}mpf = (Plus\text{-}zero,\ [])$

definition *Minus-zero-mpf* :: *mpf* **where**
 $Minus\text{-}zero\text{-}mpf = (Minus\text{-}zero,\ [])$

definition *One-mpf* :: *mpf* **where**
 $One\text{-}mpf = (One,\ [])$

A negation will be useful to get a subtraction operator from mpf-add:

fun *mpf-neg* :: $mpf \Rightarrow mpf$ **where**
 $mpf\text{-}neg\ (a,\ es) = (float\text{-}neg\ a,\ map\ float\text{-}neg\ es)$

lemma *valid-zero-mpf*:
  shows *valid Plus-zero-mpf*
  and *valid Minus-zero-mpf*
by (*simp-all add*: *Plus-zero-mpf-def Minus-zero-mpf-def float-zero1 float-zero2*)

One way to inspect which computations will be performed is to define a test mpf with dummy values and then use Isabelle's simplifier to apply the methods simplifications to the desired point:

definition a_4 = *undefined*
definition a_3 = *undefined*
definition a_2 = *undefined*
definition a_1 = *undefined*
definition a_0 = *undefined*
definition *test-mpf* = (a_4, [a_3, a_2, a_1])
definition *output* = *grow-by-fold test-mpf* a_0

To make the simplifier apply the definition, we need to state a lemma:

lemma *P output* **unfolding** *output-def test-mpf-def grow-by-fold.simps*
— We can now use various proof methods to get a neatly arranged output:
  apply (*clarsimp split*: *prod.splits*) **oops**

where P is an undefined dummy predicate. At the last step, the output is as follows:

$\bigwedge$*x1 x2 x1a x2a x1b x2b x1c x2c.*
    TwoSum x1b a_4 = (*x1c*, *x2c*) $\Longrightarrow$
    TwoSum x1a a_3 = (*x1b*, *x2b*) $\Longrightarrow$
    TwoSum x1 a_2 = (*x1a*, *x2a*) $\Longrightarrow$
    TwoSum a_0 a_1 = (*x1*, *x2*) $\Longrightarrow$
   P (*x1c*, [*x2c*, *x2b*, *x2a*, *x2*])

value *approx output* delivers

Plus-zero ⊕ *One* ⊕ *undefined* ⊕ *undefined* ⊕ *undefined* ⊕ *undefined* ⊕ *undefined*
:: *IEEE.float*

3 Code generation and Output

To profit from the accelerated execution of hardwired float operations (and also to enable an output for them in Isabelle, see below), the HOL-code needs to be translated into a compilable language like SML. To this end, Isabelle provides the **export-code** command. It uses the **code-printing** statements of the current context. However, due to such conversions being prone to introducing errors, only safe translations are being used by default, i.e. those that preserve the HOL-statements with high certainty. Thus, to enable the generation for hardware float using code, some additional translations need to be added. These should be tested thoroughly to ensure the resulting ML code's correctness.

3.1 Problems

In addition to the computation being possibly incorrect (see section "Testing PolyML"), two more problems hinder the testing possibilities:

- **value** [*code*] runs into an error because the translation from the computed `real` back into a HOL term is not implemented.
- Evaluating it via the ML command only gives the inexact representation as a rounded sequence in base 10.

  To make matters worse, the simplifier can't simulate the float operations in HOL due a lack of lemmas for the very abstract definitions of the via an all-quantifier over HOL's *real* type. Thus, we have no way to compute the correct result in our verified setting for a comparison with the SML output.

3.2 Use of SML floats

To enable computation for hardware floats, **theory** *Code-Float*[12] provides the built-in operators of the target language, e.g.:

code-printing constant *op* / :: *float* ⇒ *float* ⇒ *float* ⇀
  (*SML*) *Real.*'/ ((-), (-)) **and** (*OCaml*) *Pervasives*.('/.)
declare *divide-float-def* [*code del*]

The other operations are defined analogously.

Even ML's comparisons can be used (SML's `bool` is already defined as translation for HOL's *bool* in the *Main* theory *HOL*):

code-printing constant *Orderings.less* :: *float* ⇒ *float* ⇒ *bool* ⇀

```
  (SML) Real.< ((-), (-)) and (OCaml) Pervasives.(<)
declare less-eq-float-def [code del]
```

3.3 Printing Floats

If we decide that an unchecked code module is safe enough for us, we can use the format *Float.float*[4] from Isabelle's HOL-library to get the ability to print IEEE floats.

To enable the conversion from *IEEE.float* to *Float.float* in the generated code, we first insert the possibility to produce them from integers:

```
definition float-of-int i = Float (real-of-int i)
context includes integer.lifting begin
lift-definition float-of-integer::integer ⇒ float is float-of-int .
end

lemma float-of-int[code]:
  float-of-int i = float-of-integer (integer-of-int i)
  by (simp add: float-of-integer-def)

code-printing
  constant float-of-integer :: integer ⇒ float ⇀ (SML) Real.fromInt
declare [[code drop: float-of-integer]]
```

Then, the conversion is possible:

```
— convert hardware floats to Float.float for an exact representation
code-printing
code-module ToManExp ⇀ (SML)
  ‹fun tomanexp x =
  let
    val {man = m, exp = e} = Real.toManExp x;
    val p = Math.pow (2.0, 53.0);
    val ms = m * p;
    val mi = Real.floor ms;
    val ei = op Int.− (e, 53);
  in (mi, ei)
  end›

consts tomanexp::float ⇒ integer ∗ integer
code-printing constant tomanexp :: float ⇒ integer ∗ integer ⇀
  (SML) tomanexp

definition toFloat::float ⇒ Float.float where
  toFloat x = (let (m, e) = tomanexp x in Float.Float (int-of-integer m) (int-of-integer
e))
```

We can now define a test list:

```
definition list :: float list where
```

— Note that floats with magnitude < *1* can only be defined via *op div*:

```
list = [
  float-of-int 43,
  float-of-int 34538,
  float-of-int 3 / float-of-int 44,
  float-of-int 0,
  float-of-int 0,
  float-of-int (−348976754389282980)]
```

To use the ML operators, we have to insert the transformation to a term and back:

```
instantiation float::term-of
begin
definition term-of::float ⇒ term where term-of x = undefined
instance ..
end

code-printing
code-module FromManExp ⇀ (SML)
  ‹fun frommanexp m e = Real.fromManExp {man = Real.fromLargeInt m, exp = e}›
consts frommanexp::integer ⇒ integer ⇒ float
code-printing constant frommanexp :: integer ⇒ integer ⇒ float ⇀
  (SML) frommanexp

definition of-Float::Float.float ⇒ float where
  of-Float x = frommanexp (integer-of-int (Float.mantissa x)) (integer-of-int (Float.exponent
x))

lemma [code]: term-of-class.term-of (x::float) ≡
  Code-Evaluation.App
    (Code-Evaluation.termify of-Float)
    (term-of-class.term-of (normfloat (toFloat x)))
  by (rule term-of-anything)
```

We can now print the list without an error:

```
value list
```

produces

```
[of-Float 43, of-Float (Float.Float 17269 1),
  of-Float (Float.Float 1228254443828317 (− 54)),
  of-Float (Float.Float 0 0), of-Float (Float.Float 0 0),
  of-Float (Float.Float (− 5452761787332547) 6)]
  :: float list
```

which is an error-free representation.

4 Testing

4.1 PolyML in Isabelle2015

In SML the IEEE-floats are called `real` and use hardware operations by default. Thus, the translation

code-printing type-constructor *float* (*SML*) *real*

from *Code-Float* immediately suggests itself. When executing the presented methods however, it turned out that the output was obviously wrong. Tracing this problem back to individual code blocks lead to the conclusion that already the *TwoSum* method delivered wrong results: The sum of the two input values did not match the sum of the two output values. Luckily, the error was so large that it was not overcast by the inexact representation as rounded sequence in base 10.

This problem with the translated code turned out to stem from an unexpected computation of intermediate results in the "double extended" precision ([1] section 3.7). The ML-code used it on some systems and thus computed the error value e as the error how it would be for an addition in extended precision. As this result needed to be translated into the 64-bit format for further usage, it needed to be rounded again at the end of the method. The error of this conversion is not accommodated for in the other output value designed to deliver the correctly rounded result of the 64-addition, thus nullifying the property of error-free transformation. The 80-bit registers are used until storage in a 64-bit value is enforced. Our results are thus not determined by the sequence of operations specified in the code, but by hardly controllable circumstances e.g. the way PolyML handles function calls (see http://lists.inf.ed.ac.uk/pipermail/polyml/2015-October/001661.html). Worse still, due to different instruction sets being used, the correct behavior was depending on the operating system used.

One way to circumvent these arbitrary changes in the results is to enforce the storage in a 64-bit value after every floating point operation. From this, the approach of the "STORE"-method: was derived: The result of every addition or subtraction is written to a ML variable (which uses double precision). This value is then used for next operation.

definition *STORE* $x = x$
code-printing constant *STORE* :: $'a \Rightarrow 'a$ $\rightharpoonup$
  (*SML*) (*Unsynchronized*.! (*Unsynchronized*.*ref* ((-))))
declare [[*code drop*: *STORE*]]

fun *twoSumSTORE* :: *float* $*$ *float* $\Rightarrow$ *float* $*$*float*

```
where twoSumSTORE (a, b) =
  (let
    s =  STORE(a + b);
    an = STORE(s − b);
    bn = STORE(s − an);
    da = STORE(a − an);
    db = STORE(b − bn);
    e =  STORE(da + db)
  in (s, e))
```

This leads to the correct results, but is very slow in execution.

As another way of avoiding the unpredictable additional precision, the processors could be set to a mode that does not use this unwanted feature. This could be confirmed through compiling the exported TwoSum code with another ML compiler: running mlton with the flag "-codegen amd64" apparently produces a correct executable for TwoSum. It is desirable to make a similar modification to PolyML. This will also make the **value** command use the correct code. Kindly, polyml maintainer David Mathews provided a changed version (https://github.com/polyml/polyml/commit/218dfbd9ceb0f7ade51aece3de4f3e3800f697fb) that should work on most systems to run Isabelle with and agreed to consider the behaviour of floats in future polyml-releases. It works by setting the precision to 64-bit in the control word of every float instruction. The change made it into polyML 5.6 (http://lists.inf.ed.ac.uk/pipermail/polyml/2015-October/001695.html) and is thus available in Isabelle2016.

4.2 Ideas for Tests in Isabelle2016

We reuse the constant *list* from the chapter "Code generation and Output".

floats are much more readable when they are in their normal form. A *float* with values *a* and *b* represents the value *a* ∗ *2* ˆ *b*. It is normal if *a* is uneven.

abbreviation *toNF* :: *float* ⇒ *Float.float* **where**
  toNF ≡ *normfloat o toFloat*

Expressions like

value [*code*] *toNF* (*fold op+* (*tl list*) (*hd list*))

value [*code*] *listsum* (*map toNF list*)

value [*code*] *map toNF* (*let mpf* = (*hd list*, *tl list*); (*a*, *es*) = *grow-by-fold mpf* (*float-of 4*) *in a # es*)

value [*code*] *let mpf* = (*hd list*, *tl list*); (*a*, *es*) = *grow-mpf-tr mpf* (*float-of 4*) *in map toNF* (*a # es*)

produce now output and can be examined for correctness.

5 Results and conclusion

The IEEE 754 floats were already modelled in Isabelle. Using this formalization, this work provides an easy way to use them for fast and error-free addition and subtraction. For these operations, we translated algorithms from the literature and adapted them for our purposes.

5.1 Impact

We give a more practice-oriented analysis of Shewchuk's algorithms and offer explanations for challenges that can arise when implementing them. We also give ideas and solutions for verifying them in a functional setting. Based on the existing formalization of IEEE-floats, we then specified a data format to provide an easy access to these algorithms. This means that users have a new option for a number format to perform verified computations using fast and error-free addition and subtraction. As results of our thorough testing of generated code, an error in polyML's float handling has been detected and removed. This means the code generated using the AFP-theory *IEEE-Floating-Point/Code-Float* has now a clearer semantics.

5.2 Future Work

A correctness proof for the *TwoSum* method needs to be converted to Isabelle's IEEE754 formalization. This will then also enable proofs for Shewchuk's "nonoverlapping" property, which, when implemented, allows more assertions about multiple precision float arithmetic to be formally verified, e.g. about the maximum length of a valid *mpf*, or the quality of the approximation stored in the first component.

Another improvement could be made by adapting code generation for IEEE-floats to support more of Isabelle's target languages. This will make our arithmetic library more flexible for use in languages than SML. However, the correct behaviour of floats in the language should be ensured beforehand, to avoid getting wrong results when using the generated code.

Bibliography

[1] IEEE Standard for Floating-Point Arithmetic. *IEEE Std 754-2008*, pages 1–70, Aug 2008. http://ieeexplore.ieee.org/xpl/articleDetails.jsp?arnumber=4610935.

[2] Y. Bertot and P. Castéran. *Interactive Theorem Proving and Program Development. Coq'Art: The Calculus of Inductive Constructions*. Springer, 2004. http://www.labri.fr/perso/casteran/CoqArt/index.html.

[3] T. J. Dekker. A floating-point technique for extending the available precision. *Numerische Mathematik*, 18(3):224–242.

[4] J. Hölzl and F. Immler. Floating-point numbers. 2012. https://isabelle.in.tum.de/library/HOL/HOL-Library/Float.html, TU München.

[5] M. Joldes, V. Popescu, and W. Tucker. Searching for sinks of Henon map using a multiple-precision GPU arithmetic library. Technical report, Nov. 2013. https://hal.archives-ouvertes.fr/hal-00957438, 7 pages.

[6] O. Møller. Quasi double-precision in floating point addition. *BIT Numerical Mathematics*, 5(1):37–50.

[7] J.-M. Muller, N. Brisebarre, F. de Dinechin, C.-P. Jeannerod, V. Lefèvre, G. Melquiond, N. Revol, D. Stehlé, and S. Torres. *Handbook of Floating-Point Arithmetic*. Birkhäuser Boston, 2010. ACM G.1.0; G.1.2; G.4; B.2.0; B.2.4; F.2.1., ISBN 978-0-8176-4704-9.

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

[9] D. M. Priest. Algorithms for arbitrary precision floating point arithmetic. In *Computer Arithmetic, 1991. Proceedings., 10th IEEE Symposium on*, pages 132–143, Jun 1991.

[10] J. R. Shewchuk. Adaptive Precision Floating-Point Arithmetic and Fast Robust Geometric Predicates. *Discrete & Computational Geometry*, 18(3):305–363, Oct. 1997.

[11] L. Théry, L. Rideau, L. Fousse, G. Melquiond, and S. Boldo. Twosum. *A Coq Library on Floating-Point Arithmetic*. http://lipforge.ens-lyon.fr/www/pff/TwoSum.html.

[12] L. Yu. A Formal Model of IEEE Floating Point Arithmetic. *Archive of Formal Proofs*, July 2013. http://afp.sf.net/entries/IEEE_Floating_Point.shtml, Formal proof development.

Appendix

The Isabelle code for this thesis is available at Github: https://github.com/Helli/IsabelleBasicNumericalProofs.

An Isabelle session typesetting this document (using "isabelle build") is at https://github.com/Helli/IsabelleBasicNumericalProofs/tree/thesis.