Department of Computer Science,
University of Otago

Technical Report OUCS-2011-04

Specifying Exact Scaled Decimal Arithmetic

Author:

Richard A. O'Keefe

Department of Computer Science, University of Otago, New Zealand

---

Department of Computer Science,
University of Otago, PO Box 56, Dunedin, Otago, New Zealand

http://www.cs.otago.ac.nz/research/techreports.php

Specifying Exact Scaled Decimal Arithmetic

Richard A. O'Keefe
Department of Computer Science
The University of Otago

28 November 2011

Abstract

The ANSI Smalltalk standard includes a ScaledDecimal class for decimal fixed point arithmetic, but the specification is so vague that implementations vary greatly. The Language Independent Arithmetic standard has nothing to say about this data type. This article presents one reasonable specification, treating these numbers as exact.

1 Introduction

The ANSI Smalltalk standard [ANSI INCITS 319-1998] requires Smalltalk systems to offer several kinds of numbers: exact unbounded integers, exact unbounded fractions (rational numbers other than integers), up to three different sizes of floating point numbers, and also scaled decimal numbers.

Unfortunately, it fails to specify the semantics of scaled decimal arithmetic precisely. Existing Smalltalk implementations

- omit this type entirely, or
- treat it as 31 digit IBM/360-style packed decimal, rescaling to avoid overflow (so the decimal point is not as fixed as you might expect), or
- treat it as exact rational numbers with any denominator whatever, with the scale factor controlling only how a number is printed, not how it behaves in arithmetic, or
- treat it as exact rational numbers with a power of ten denominator.

Some programming languages, notably Ada, have fixed point numeric types thought of as approximations with a stated absolute error bound. That is a perfectly reasonable form of arithmetic, and it is not clear that the ANSI Smalltalk standard was intended to rule it out.

The ANSI Smalltalk standard repeatedly defers to the Language Independent Arithmetic standard with the phrase "as specified by ... ISO/IEC 10967".

Unfortunately, LIA still has nothing whatever to say about fixed point arithmetic of any kind.

A specification could be

- natural language text, as in the ANSI Smalltalk standard;
- formal mathematics with English text, as in ISO/IEC 10967;
- a reference implementation in a conventional programming language; or
- an executable specification.

We need a high degree of confidence in at least the completeness and consistency of the specification. It should not use any undefined operations. It should at least type check. It should be comparatively easy to test, to make sure that at least the most obvious errors are not present. Above all, once we have our specification, we would like to be able to use it to generate test cases for a system that is meant to conform to the specification.

All of these desiderata point to an executable specification in some high level or specification language. The existence of the QuickCheck library for automatically testing properties of Haskell functions makes Haskell an excellent choice.

This article was written as a Literate Haskell source file which can be processed by LaTeX for printing, or loaded directly by a Haskell system. The code that you see is the code that was checked.

2 The code

We want an abstract data type, so we use a module. Almost all the operations we need are defined in Haskell type-classes, so do not need to be exported explicitly. Smalltalk has an operation to read the scale of a number, so we have to export the type class defining that. A function for converting real numbers to scaled decimal with a given scale does not fit into any existing Haskell class, so is exported by itself, as are some generalisations of rounding.

```
module ScaledDecimal (
    ScaledDecimal,
    Scaled,
    ceilingTo,
    floorTo,
    roundTo,
    truncateTo,
    toScaledDecimal
) where
```

We can construct a scaled decimal number in two different ways other than arithmetic operations. Given a String at run time, we can read a number from

it. We need two functions from the Char module for that. The Haskell language says that a numeric literal with a decimal point in the source code is processed as if first read as an arbitrarily precise rational number and then converted using the `fromRational` function, so we need the numerator and denominator functions to unpack that rational number and the division operator that makes ratios from integers.

```
import Char (isSpace, isDigit)
import Ratio (numerator, denominator, (%))
```

We represent a scaled decimal number as a pair (SD n s) standing for the exact rational number $n \times 10^{-s}$. Haskell is a non-strict functional language, meaning that just because an expression has been evaluated to the point where we know it is (SD _ _) that doesn't necessarily mean the components are known. We use strict data annotations to say that either nothing is known about a number yet or all of it is known. There are only two places where this module takes advantage of Hasell's laziness, and they aren't needed to specify Smalltalk.

```
data ScaledDecimal = SD !Integer !Int
```

The standard says

> The maximum allowed precision for a scaled decimal numeric object is implementation defined and may be unbounded. (3.4.6.1)
>
> The maximum precision must be at least 30. (3.6)
>
> If the result [of an operation] is outside of the range of the common numeric representation, the effect of underflow or overflow is implementation defined. (5.6.2.1 and elsewhere)
>
> The effect of underflow and overflow is implementation defined. (5.6.2.17)
>
> It is an error if the [source of a conversion] cannot be represented within the maximum precision of the [ScaledDecimal] implementation. (5.6.5.3)
>
> The representation must be able to accurately represent decimal fractions. The standard recommends that the implementation of this protocol support unbounded precision, with no limit to the number of digits before and after the decimal point. If a bounded implementation is provided, then any operation which exceeds the bounds has an implementation-specified result. (5.6.6)

It is not obvious why a language that is required to support integer arithmetic with no fixed bounds should allow arbitrary bounds to be imposed on scaled decimal arithmetic. The answer appears to be VisualAge Smalltalk, which implements decimal numbers using 17 bytes; one byte for the scale and 16 to hold 31 decimal digits and a sign in IBM/360 packed decimal format. In the case of precision or scale overflow, VisualAge Smalltalk shifts the decimal point to the right if possible. Such a representation makes excellent sense for interfacing to

an SQL implementation with a similar representation, but it does not make for arithmetic that is easy to reason about.

In order to accommodate some such implementation-specific fudging, this specification uses a function

```
approx :: Integer -> Int -> ScaledDecimal

approx n s = SD n s -- may be redefined by an implementation
```

We use the constructor `SD` where it is obvious that no overflow is possible, `approx` otherwise.

Many of the operations that work on two numbers need to have those numbers represented in the same scale. The function `common_scale` does this. It is not exported. It works by multiplying the number with the smaller scale by a suitable power of 10.

```
common_scale :: ScaledDecimal -> ScaledDecimal -> (Integer, Integer, Int)

common_scale (SD nx sx) (SD ny sy)
  | sx > sy = (nx, ny * 10^(sx-sy), sx)
  | sx < sy = (nx * 10^(sy-sx), ny, sy)
  | True    = (nx,              ny, sx)
```

The scale of a scaled decimal number can be recovered using the `scale` function. Since integers behave like scaled decimals with a scale of zero, we may as well extend `scale` to cover them. We shall see later that defining a version of division that returns a scaled decimal result is problematic, but since they are abstractly rational numbers, it should be easy to divide two scaled decimals and get an exact rational answer. This too we extend to integers.

```
class Integral a => Scaled a
  where
    scale :: a -> Int
    scale _ = 0
    ratio :: a -> a -> Rational
    ratio x y = fromIntegral x % (fromIntegral y :: Integer)

instance Scaled Int

instance Scaled Integer

instance Scaled ScaledDecimal where
  scale (SD _ s) = s
  ratio x y = ratio nx ny where (nx,ny,_) = common_scale x y
```

This gives us a way to convert a scaled decimal to rational: `ratio` x 1. The Smalltalk standard requires a function that takes a number and a scale and

returns a scaled decimal with that scale approximating the given number as well as possible. Basically, we are going to convert x to `round`$(x \times 10^s)$. Plugging this into the Haskell numeric framework is a little tricky, because while `round` is defined on rational numbers and floats, it is not defined on integers. Fortunately, we are producing an executable specification, not a high performance library, so we can start by converting x to a rational number.

```
toScaledDecimal :: Real a => a -> Int -> ScaledDecimal

toScaledDecimal x s =
  approx (round (toRational x * 10^s)) s
\end{Code}

If a type belongs to the Haskell type class called Eq,
you can use equality (\verb|==|) and inequality (\verb|/=|)
on that type.  If we define either function, the other is
automatically defined.

\begin{code}
instance Eq ScaledDecimal where
  x == y = nx == ny
    where (nx,ny,_) = common_scale x y
```

The type class Ord deals with the ordered comparison predicates (`<`), (`>=`), (`>`), (`<=`), `max`, `min`, and the three-way comparison `compare`. If you define three-way comparison, the other functions are automatically defined, as are things like sorting.

```
instance Ord ScaledDecimal where
  compare x y = compare nx ny
    where (nx,ny,_) = common_scale x y
```

Now we can define arithmetic. The arithmetic operations are spread across several classes, and the assignment of operations to classes is not always convenient. Addition, and subtraction seem trivial, but express our intention that these numbers be regarded as *exact*. For approximate numbers, we would convert the operands to the *smaller* scale, not the larger.

```
instance Num ScaledDecimal where
  negate (SD n s) = SD (negate n) s
  abs    (SD n s) = SD (abs n)    s
  signum (SD n _) = SD (signum n) 0
  fromInteger i   = approx i 0

  x + y = approx (nx+ny) s where (nx,ny,s) = common_scale x y
  x - y = approx (nx-ny) s where (nx,ny,s) = common_scale x y
  (SD nx sx) * (SD ny sy) = approx (nx*ny) (sx+sy)
```

Since we view scaled decimal numbers as exact rationals, we should be able to convert them to the Rational type. C programmers beware: (%) is not a remainder operator, but a division operator that produces Rational results.

```
instance Real ScaledDecimal where
  toRational (SD n s) = n % 10^s
```

Now we run into one of the glitches in the Haskell numeric classes. We want to define quotient and remainder for our numbers, which means they have to belong to the Integral class (glitch one), and that requires them to be in the Enum class (glitch two), which not only provides enumeration over ranges of numbers, but also provides two-way conversion between these numbers and hardware integersfromEnum (glitch three). It happens that we can define enumeration reasonably enough, so glitch two is not much of a problem. Converting machine integers to scaled decimal is always possible, but the `fromEnum` function is a problem. We don't want it; it is there just to satisfy Haskell. The definition of the Enum class for floating point types is somewhat contentious in the Haskell community. The following definitions of `fromEnum` and `toEnum` for are consistent with those for Double.

Of the four enumeration functions, we only need the two finite ones for specifying Smalltalk, and could have managed without those.

- (enumFrom x) enumerates all the rational numbers with the same scale as x, starting from x. This is an infinite list. It's one of the two functions that depends on laziness.
- (enumFromThen x y) reports x, $x + (y - x)$, $x + 2(y - x)$, ... This is an infinite list. It is the other function that depends on laziness.
- (enumFromTo x z) enumerates all the rational numbers with the same scale as x, lying between x and z inclusive.
- (enumFromThenTo x y z) reports x, $x + (y - x)$, $x + 2(y - x)$, ..., up to and including z. The first and second elements may have different scales; this is deliberate.

```
round_to_integer (SD n s) = (n * 10 + signum n * 5) `quot` 10 ^ (s + 1)

instance Enum ScaledDecimal where
  succ (SD n s) = approx (succ n) s
  pred (SD n s) = approx (pred n) s
  toEnum i      = approx (fromIntegral i) 0
  fromEnum x    = fromIntegral (round_to_integer x) -- bogus

  enumFrom       x     = x : enumFrom (succ x)
  enumFromThen   x y   = loop x (y-x)
                         where loop x d = x : loop (x+d) d
  enumFromTo     x   z = if x > z then [] else x : enumFromTo (succ x) z
```

```
  enumFromThenTo x y z = loop x (y-x)
                         where loop x d = if x > z then []
                                          else x : loop (x+d) d
```

What we really want is quotient and remainder operations. The traditional quotient operator truncates towards zero; this was used in Fortran and Algol 60, and is still used in C. That definition has been criticised for a long time, with truncation towards negative infinity (floor division) preferred. LIA-1 offers both definitions. So does Smalltalk, and so does Haskell. The `quotRem` function provides truncating quotient and remainder. Defining it gives us `quot` and `rem` as well. The `divMod` function provides flooring quotient and remainder. Defining it gives us `div` and `mod` as well.

The Haskell library designers took the view that these operations only made sense for integral numbers. Not so. They make perfect sense for any rational numbers. The definition of `toInteger` is not just a type conversion, it is rounding. The reporting of division by zero is delegated to the integer division functions.

```
instance Integral ScaledDecimal where
  toInteger x = round_to_integer x
  quotRem x y = (SD q 0, SD r s)
                where (nx, ny, s) = common_scale x y
                      (q, r)      = quotRem nx ny
  divMod x y  = (SD q 0, SD r s)
                where (nx, ny, s) = common_scale x y
                      (q, r)      = divMod nx ny
```

You would expect that the step from quotient and remainder to conversion to integer would be a small one. After all,

```
floor x = q where (q,r) = x `divMod` 1
```

You would be wrong. In order to define these Haskell functions on scaled decimal integers, these numbers have to count as Fractional, which means supporting division that returns a number of the same kind as the operands.

Now we run into a big difficulty. If $x = m \times 10^{-s}$ and $y = n \times 10^{-t}$, it does not follow that the ratio x/y has this form. Consider 0.3/0.7.

Possible responses to this problem are

- do not define this operation. Since the Smalltalk standard requires it, we can't do that.
- observe that the answer *is* always a rational number, so have the quotient of any mix of integers, rationals, and scaled decimals produce a rational number answer. Since the other operations on scaled decimals are exact, it would be pleasant and consistent for this one to be exact also. The Haskell type system does not permit that: x/y must have the same type as its operands. That need not be a problem because we could define

Smalltalk's (/) in terms of some other Haskell function like our `ratio`. However, the Smalltalk standard does not allow it.

- define it to return a scaled decimal approximation to the exact rational result.

The Smalltalk standard says only that "the scale of the result is at least the scale of the receiver", where the receiver in x/y is x. It is not enough to make the scale of the result be the greater of scale x and scale y; that would give a very poor result for 1.0/3.0, as PL/I programmers knew to their cost. Here we have to make an essentially arbitrary choice about the scale of the result, and raise the scale to at least 18.

The `fromRational` function is used, amongst other things, to allow overloaded floating point literals. Haskell defines the value of a literal with a decimal point to be the result of applying `fromRational` to an exact rational version of the literal. The function here is not total: it works only when the conversion can be done exactly. It is enough to make literals like 1.5 work.

```
instance Fractional ScaledDecimal where
  recip y = 1 / y

  x / y = toScaledDecimal (ratio x y) (scale x `max` scale y `max` 18)

  fromRational x = find_scale (numerator x) (denominator x) 0
    where find_scale n 1 s = approx n s
          find_scale n d s =
              if d`mod`10 == 0 then find_scale n     (d`div`10) (s+1) else
              if d`mod` 2 == 0 then find_scale (n*5) (d`div` 2) (s+1) else
              if d`mod` 5 == 0 then find_scale (n*2) (d`div` 5) (s+1) else
              error "ScaledDecimal.fromRational: not a decimal number"
```

We complete the arithmetic operations with floor and friends. The calls to `fromIntegral` are required to allow the caller to determine the integral result type.

```
instance RealFrac ScaledDecimal where
  properFraction (SD n s) = (fromIntegral q, SD r s)
                            where (q,r) = quotRem n (10^s)
  truncate (SD n s) = fromIntegral (n `quot` 10^s)
  floor    (SD n s) = fromIntegral (n `div`  10^s)
  ceiling  (SD n s) = fromIntegral (negate (negate n `div` 10^s))
  round    (SD n s) = fromIntegral ((n*10+5) `quot` 10^(s+1))
```

Smalltalk adds extends the idea of converting to a whole multiple of 1 to a whole multiple of some given y, but it only provides two of the expected four. We can define these operations on any numeric type that can be converted to a rational number.

```
truncateTo, floorTo, ceilingTo, roundTo :: Real a => a -> a -> a
helper :: Real a => (Rational -> Integer) -> a -> a -> a

helper f x y = fromIntegral (f (toRational x / toRational y)) * y
truncateTo = helper truncate
floorTo    = helper floor
ceilingTo  = helper ceiling
roundTo    = helper round
```

All that's left is conversion between internal representations and textual representations. The Smalltalk standard defines output of scaled decimals, weakly, and scaled decimal literals in source code, but offers nothing that can convert from text to numeric form at run time. Actual Smalltalk systems do have a way to do input conversions, but it was too powerful (and so vulnerable) to be included in the standard. The conversions here are not exactly what Smalltalk wants, not that existing Smalltalks agree perfectly, but are building blocks from which suitable definitions can be made.

```
instance Show ScaledDecimal where
  showsPrec p (SD n s) suffix
    | n < 0 && p > 6 = "(-" ++ showAbs n s (")" ++ suffix)
    | True           =         showAbs n s suffix
    where showAbs n s suffix
            | s <= 0 = shows n suffix
            | True = take (n1-s) d1 ++ "." ++ drop (n1-s) d1 ++ suffix
            where d0  = show n
                  n0 = length d0
                  d1 = replicate (s+1-n0) '0' ++ d0
                  n1 = length d1

instance Read ScaledDecimal where
  readsPrec _ cs = parse cs
    where parse ( c:cs) | isSpace c = parse cs
          parse ('+':cs) = before cs 0 False
          parse ('-':cs) = before cs 0 True
          parse      cs  = before cs 0 False

          before ( c :cs) n neg | isDigit c = before cs (n*10 + val c) neg
          before ('.':cs) n neg = after cs n neg 0
          before      cs  n neg =
            [(approx (if neg then negate n else n) 0, cs)]

          after (c:cs) n neg s | isDigit c = after cs (n*10 + val c) neg (s+1)
          after cs     n neg s =
            [(approx (if neg then negate n else n) s, cs)]
```

```
val '0' = 0
val '1' = 1
val '2' = 2
val '3' = 3
val '4' = 4
val '5' = 5
val '6' = 6
val '7' = 7
val '8' = 8
val '9' = 9
```

Smalltalk defines `sqrt` for all numbers. The usual definition applies to floats; integers return the floor of the square root as an integer; but for rational numbers and scaled decimals the standard defers to ISO 10967, which has explicitly nothing to say about rational numbers or scaled decimals. Historic systems just convert to floating point. It would be possible to implement a floor square root like the one for integers, but it was easiest to follow historic practice.

Smalltalk also defines a general exponentiation operator for all numbers. However, section 5.6.2.27 says of (x raisedTo: y) that

> If [y is an integer]. answer the result of [raising x to an integral power]. Otherwise, answer
>
> `(x asFloat ln * operand) exp`

This has the unfortunate consequence that $-27^{1/3}$ is an error instead of -3, but it does mean that we do not need to specify this operation for scaled decimals.

Every value in Smalltalk can be hashed. If $x = y$ it is required that $xhash = yhash$. This means, for example, that 1, 1.0, 1.00, and 1.0e0 must all have the same hash value. We can express that requirement by definining

```
hash :: Real a => a -> Int
hash x = rationalHash (toRational x)
```

but there is nothing specific to say about `rationalHash`, so neither it nor `hash` is defined here.

3 Testing the specification

Using the QuickCheck library, we can write tests like these:

```
prop_scale =
    forAll siGen $ \i ->  -- i is a signed Integer
    forAll ssGen $ \s ->  -- s is a small non-negative Int
    scale (toScaledDecimal i s) == s

prop_ratio =
```

```
    forAll siGen $ \n ->
    forAll piGen $ \d ->  -- d is a positive Integer
    forAll ssGen $ \s ->
    ratio (toScaledDecimal n s) (toScaledDecimal d s) == ratio n d

prop_equality_ignores_scale =
    forAll siGen $ \n ->
    forAll ssGen $ \s ->
    forAll ssGen $ \t ->
    toScaledDecimal n s == toScaledDecimal n t

prop_equality_commutes =
    forAll sdGen $ \x ->  -- x is a ScaledDecimal
    forAll sdGen $ \y ->
    (x == y) == (y == x)

prop_equality_is_transitive =
    forAll sdGen $ \x ->
    forAll sdGen $ \y ->
    forAll sdGen $ \z ->
    x == y ==> (y == z) == (x == z)

prop_less_and_greater_are_opposites =
    forAll sdGen $ \x ->
    forAll sdGen $ \y ->
    (x < y) == (y > x)

prop_less_and_greater_equal_are_complements =
    forAll sdGen $ \x ->
    forAll sdGen $ \y ->
    (x < y) /= (x >= y)

prop_negate_negates_sign =
    forAll sdGen $ \x ->
    signum (negate x) == negate (signum x)

prop_negate_cancels =
    forAll sdGen $ \x ->
    negate (negate x) == x

prop_abs_times_sign_is_identity =
    forAll sdGen $ \x ->
    abs x * signum x == x

prop_to_integer_is_inverse_of_from_integer =
    forAll siGen $ \i ->
```

```
    toInteger (fromInteger i :: ScaledDecimal) == i

prop_zero_is_additive_identity =
    forAll sdGen $ \x ->
    forAll ssGen $ \s ->
    let z = toScaledDecimal 0 s in
    z + x == x && x + z == x && x - z == x

prop_addition_commutes =
    forAll sdGen $ \x ->
    forAll sdGen $ \y ->
    x + y == y + x

prop_addition_associates =
    forAll sdGen $ \x ->
    forAll sdGen $ \y ->
    forAll sdGen $ \z ->
    x + (y + z) == (x + y) + z

prop_negation_is_compatible_with_subtraction =
    forAll sdGen $ \x ->
    forAll sdGen $ \y ->
    x - y == x + negate y
```

Other test cases were also written and run. These tests were very effective in finding errors, all but one of which turned out to be mistakes in the formulation of the tests. The original definition of (/) was quite wrong.

4 Defining the Smalltalk operations

$x * y$	$x * y$
$x + y$	$x + y$
$x - y$	$x - y$
x/y	x/y
$x//y$	x ‘div‘ y
$x < y$	$x < y$
$x = y$	$x == y$
$x\backslash\backslash y$	x ‘mod‘ y
x abs	abs x
x asFloat	asFloatE or asFloatD
x asFloatE	fromRational (toRational x) :: Float
x asFloatD	fromRational (toRational x) :: Double
x asFloatQ	like asFloatD
x asFraction	toRational x
x asInteger	toInteger x
x asScaledDecimal: s	toScaledDecimal x s
x ceiling	ceiling x
x floor	floor x
x fractionPart	snd (properFraction x)
x integerPart	fst (properFraction x)
x negated	negate x
x negative	$x < 0$
x positive	$x >= 0$
x printString	shows x (’s’ : show (scale x))
x quo: y	x ‘quot‘ y
x raisedToInteger: y	x ^^ y
x reciprocal	recip x
x rem: y	x ‘rem‘ y
x rounded	round x
x roundTo: y	roundTo x
x scale	scale x
x sign	signum x
x sqrt	sqrt (fromRational (toRational x) :: Double)
x squared	x ^^ 2
x strictlyPositive	$x > 0$
x to: z	enumFromThenTo x $(x + 1)$ z
x to: z by: y	enumFromThenTo x $(x + y)$ z
x truncated	truncate x
x truncateTo: y	truncateTo x

5 Generality of this specification, and suitability of Haskell

There are other programming languages than Smalltalk with some kind of fixed point support. The present specification is equally suitable for languages with static typing (it's written in one) and dynamic typing (it was devised for one). Languages like PL/I, Ada, and SQL make the scale part of the static type of a literal, variable, or expression; this specification makes it part of the values. However, Haskell with multi-parameter type classes is powerful enough to make the scale part of the type also, at the price of not fitting into the Haskell numeric classes.

This specification appears to make essential use of unbounded integers. Thanks to the `approx` function, this is not so. There are no limitations on the specification due to the choice of Haskell.