1 The specification

We shall give a specification for fixed point arithmetic in two steps. First, we shall specify unbounded exact arithmetic, and then we shall specify a framework for converting fixed point numbers to "fixed point frames". The specification is executable code in the programming language Haskell [ref].

One specification covers both binary and decimal fixed point arithmetic. The base is part of the type of a fixed point number. We need a popular extension to Haskell 2010, and we're going to need a helper function for converting to text.

```
{-# LANGUAGE ScopedTypeVariables #-}
import Numeric (showSigned)
```

Binary and Decimal are "empty types" with no values other than $\perp$ used as compile-time marks. The type class Has_Radix exists to convert these marks to run-time values.

```
data Binary
data Decimal

class Has_Radix t
  where radix :: t → Integer

instance Has_Radix Binary
  where radix _ = 2

instance Has_Radix Decimal
  where radix _ = 10
```

Fixed_Point t is a "phantom type", so called because it contains no run-time value of type t. The values it does contain are an integer m and a scale s so that it represents the value $m \times r^s$ where r is the radix determined by t. These integers are unbounded; we are dealing with exact arithmetic at this point.

```
data Fixed_Point t = FP !Integer !Integer
```

From an FP value we can extract its scale and base, even though the base is not stored.

```
scale :: forall t ∘ Has_Radix t ⇒ FixedPoint t → Integer
scale (FP _ s) = s

base :: forall t ∘ Has_Radix t ⇒ FixedPoint t → Integer
base _ = radix (undefined :: t)
```

The :: line for the scale function says "for any type t, provided that t belongs to the Has_Radix class (that is, 'radix' is defined on it), scale is a function with a Fixed_Point argument that delivers an Integer result. The ScopedTypeVariables feature allows us to mention t in the body of the base function.

1.1 Comparison

What does it mean for (FP m_1 s_1) to equal (FP m_2 s_2)? Consider (FP 0 0) and (FP 0 1). They both represent 0. But they are behaviourally different. For example, (FP 0 0) should print as `"0"` while (FP 0 1) should print as `"0.0"`. If we want equality to satisfy Leibniz' law, we have to regard these as distinct. On the other hand, it would be difficult to do numeric calculations if zero wasn't equal to zero.

The standard mathematical way to deal with this is to distinguish two spaces: a space X of (m, s) pairs which are equal if and only if identity and a quotient space $F = X/\equiv$ where $(m_1, s_1) \equiv (m_2, s_2)$ if and only if $m_1 \times r^{s_2} = m_2 \times r^{s_1}$. Haskell's `==`, which we are about to define, should be thought of as $\equiv$ rather than $=$.

In order to compare two numbers, it helps to align them so they have their radix points in the same place. This is also useful for addition and subtraction.

```
align :: forall t a ∘ Has_Radix t ⇒
    Fixed_Point t → Fixed_Point t → (Integer → Integer → a) → a

align (FP m1 s1) (FP m2 s2) f
  | s1 > s2 = f m1 (m2 * r ^ (s1-s2))
  | s1 < s2 = f (m1 * r ^ (s2-s1)) m2
  | True    = f m1 m2
  where r = radix (undefined :: t)
```

This takes two fixed-point numbers in the same base, aligns them, and passes the aligned values to a function of integers.

By plugging Fixed_Point into Haskell's Eq type class and defining `==`, we get `/=` for free.

```
instance (Has_Radix t) ⇒ Eq (Fixed_Point t)
  where x1 == x2 = align x1 x2 (==)
```

By plugging Fixed_Point into Haskell's Ord type-class and defining three-way comparison, we get $<$, $\leq$, $>$, $>=$, max, and min for free.

```
instance (Has_Radix t) ⇒ Ord (Fixed_Point t)
  where compare x1 x2 = align x1 x2 compare
```

1.2 Converting numbers to text

For general use, we would need a function to convert a number to a specified number of decimal digits according to a given rounding mode. Here we simply specify conversion to decimal. Numbers with scale $s \leq 0$ have no decimal point; if $s > 0$ the number of decimal digits required is s whether the base is Binary or Decimal. If we supported bases that were not a multiple of 2, fixed-point numbers in such a base might not have a finite decimal representation.

Plugging into Haskell's Show type-class takes care of inserting a negative sign (and if the context demands it, enclosing parentheses), and avoiding excess concatenation.

```
instance (Has_Radix t) ⇒ Show (Fixed_Point t)
  where
    showsPrec p x = showSigned showPos p x
      where showPos x@(FP m s) rest =
              if s ≤ 0 then shows (m ∗ base x^negate s) rest
              else shows i ('.' : show_fract s r)
              where p = base x ^ s
                    (i,r) = quotRem m p
                    show_fract 0 _ = rest
                    show_fract d f = shows h (show_fract (d-1) l)
                      where (h,l) = quotRem (10 ∗ f) p
```

Converting from text to numbers runs into the problem that 0.1 has no finite representation as a Fixed_Point Binary. We'll deal with that after considering rounding.

1.3 Ring arithmetic

Numbers of the form $m \times r^s$ form a ring under the usual arithmetic operations, but they are *not* a field. Consider 1.0/3.0; $\frac{1}{3}$ has no finite decimal representation. We can convert a fixed-point number to an exact Rational number and those *are* a field. None of the ring operations can fail.

```
instance (Has_Radix t) ⇒ Num (Fixed_Point t)
  where
    x + y = FP (align x y (+)) (scale x `max` scale y)
    x - y = FP (align x y (-)) (scale x `max` scale y)
    negate (FP m s) = FP (negate m) s
    abs    (FP m s) = FP (abs    m) s
    signum (FP m s) = FP (signum m) 0
    fromInteger n   = FP n 0
    (FP m1 s1) ∗ (FP m2 s2)  = FP (m1∗m2) (s1+s2)

instance (Has_Radix t) ⇒ Real (Fixed_Point t)
  where
    toRational x@(FP m s) = toRational m / toRational (base x) ^ s
```

Since raising to a non-negative power requires only the unit (fromInteger 1) and multiplication, we get the power operation $x\hat{}n$ free. This operation fails if $n < 0$. For example,

```
> (1234 2 :: Fixed_Point Decimal) ^ 4
23187.85835536
```

1.4 Conversion to integer

In [section Rounding] we identified 14 rounding modes. We have three options for naming them:

- They could be *values*, as done in this section.

- They could be *functions*, *e.g.*,

  ```
  down :: Real t ⇒ Integer
  down x = floor (toRational x)
  ```

  We can construct these functions if we want them:

  ```
  down = convert Down
  ```

- They could be *types* like Binary and Decimal, which would mean that the Fixed_Frame type defined in a later section would have less run-time content.

Very little hangs on this.

```
data Rounding_Mode
   = Down
   | Up
   | In
   | Out
   | Exact
   | Even
   | Odd
   | Nearest Rounding_Mode
```

This is very nearly an enumeration type, except that it allows constructions like Nearest Out. (It also allows Nearest (Nearest Odd), which is useless but harmless.)

We're going to define a single "convert" function that has a rounding mode and a rational number and returns an integer. This depends on the standard function 'properFraction', which splits a rational number into an integer part and a fractional part of the same type as its argument, such that both parts are non-negative or both parts are non-positive. RealFrac is the class of numeric types on which it is defined.

```
convert :: RealFrac t ⇒ Rounding_Mode → t → Integer

convert round x =
  case round of
    Down      → if d < 0 then i-1 else i
    Up        → if d > 0 then i+1 else i
    In        → i
    Out       → i + d
    Exact     → if d == 0 then i else error "inexact"
    Even      → if even i then i else i + d
    Odd       → if odd  i then i else i + d
    Nearest h → if d == 0 then i else
                case compare (2*abs f) 1 of
                  LT → i
                  GT → i + d
                  EQ → i + convert h f
```

```
  where (i,f) = properFraction x
        d     = if f < 0 then -1 else if f > 0 then 1 else 0
```

We can't just plug Fixed_Point into that because numeric types in Haskell can only belong to RealFrac if they also belong to Fractional, which requires them to have division, reciprocal, and conversion from arbitrary rationals.

This is an occasion where Haskell proves to be inconvenient, because the properFraction, truncate, round, ceiling, and floor functions of the RealFrac class make perfect sense for fixed-point numbers.

```
convertFixed :: Has_Radix t ⇒
    Rounding_Mode → Fixed_Point t → (Integer, Fixed_Point t)

convertFixed round x = (q, x - fromInteger q)
  where q = convert round (toRational x)
```

Given this operation, we can define division with an integer quotient and fixed-point remainder.

```
quotient :: Has_Radix t ⇒
    Rounding_Mode ⇒ Fixed_Point t → Fixed_Point t →
    (Integer, Fixed_Point t)
quotient round x y = (q, x - fromInteger q * y)
  where q = convertFixed round x
```

The Haskell type-class Integral provides integer quotient and remainder in the type-class Integral. This is arguably the wrong place, which explains why we have

```
quotient Floor -- instead of divMod
quotient In    -- instead of quotRem
```

1.5 Frames

1.6 Division