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. The LANGUAGE pragmas enable lanuage features added since the Haskell 2010 report. The import declarations make the (named) functions available, these will be used in text↔number conversion.

```
{-# LANGUAGE ScopedTypeVariables #-}
{-# LANGUAGE GeneralizedNewtypeDeriving #-}
import Numeric (showSigned)
import Data.Char (isDigit)
```

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
```

A fixed point number representation contains two unbounded integers. To avoid confusing them, we give scales a different type. The Scale t type has exactly the same representation as the Integer type; it has the same integer literals; and it has (some of) the same operations. But it is a different type, which helps us avoid errors. It is a "phantom type", so called because it contains no run-time value of type t. The ScopeTypeVariables feature means that the type variable t is available in the body of the 'shift' function, so that we can extract the radix without storing it anywhere. The 'shift' function shifts a number left or right by some number of digits; it is exactly what a shift instruction would do on a binary or decimal sign-and-magnitude machine.

```
newtype Scale t = Scale Integer
                  deriving (Eq, Ord, Show, Num)

shift :: forall t ∘ Has_Radix t ⇒ Integer → Scale t → Integer
shift m (Scale p) =
  if p ≥ 0 then m * radix (undefined :: t) ^ p
  else m `quot` radix (undefined :: t) ^ negate p
```

A Fixed t contains 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 t = FP !Integer !(Scale t)
```

For testing we want to be able to create an integral Fixed t value with a given scale.

```
toFixed :: Has_Radix t ⇒ Integer → Scale t → Fixed t
toFixed n s = FP (shift n s) s
```

The :: line for the toFixed function says "for any type t, provided that t belongs to the Has_Radix class (that is, 'radix' is defined on it), toFixed a function with an Integer argument and a Scale t argument that delivers a Fixed t result.

From an FP value we can extract its scale and base, even though the base is not stored. We can also provide the LIA function *ulp* (unit in last place).

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

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

ulp :: Has_Radix t ⇒ Fixed t → Fixed t
ulp (FP _ s) = FP 1 s
```

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 t → Fixed t → (Integer → Integer → a) → a

align (FP m1 s1) (FP m2 s2) f
  | s1 > s2 = f m1 (shift m2 (s1-s2))
  | s1 < s2 = f (shift m1 (s2-s1)) m2
  | True    = f m1 m2
```

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 into Haskell's Eq type class and defining `==`, we get `/=` for free, correct by construction (if `==` is). This provides the LIA operations *eq* and *neq*.

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

By plugging Fixed into Haskell's Ord type-class and defining three-way comparison, we get $<$, $\leq$, $>$, $>=$, max, and min for free, correct by construction. This provides the LIA-1 operations *lss*, *leq*, *gtr*, and *geq* and the LIA-2 operations *max* and *min*, and since 'maximum' and 'minimum' are predefined folds derived from 'max' and 'min', the LIA-2 operations *max_seq* and *min_seq*.

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

This is all we need to extend LIA-2's divisibility test to fixed-point numbers:

```
lia_divides :: Has_Radix t ⇒ Fixed t → Fixed t → Bool
lia_divides x y = align x y (λm n → m /= 0 && n`rem`m == 0)
```

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 t)
  where
    showsPrec p x = showSigned showPos p x
      where showPos x@(FP m s) rest =
              if s ≤ 0 then shows (shift m (negate s)) rest
              else shows i ('.' : show_fract s r)
              where p = shift 1 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 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, see subsection Division. We can convert a fixed-point number to an exact Rational number and those *are* a field. None of the ring operations can fail. This provides the LIA operations *add*, *sub*, *mul*, *neg*, *abs*, and *sign*. The 2012 revision of LIA-1 defines *signum* instead of *sign*. One possible confusion is that LIA's *sign* is Haskell's *signum*; testing is advisible to catch mistakes there. The LIA-2 *dim* function can also be defined for any ordered numeric type.

```
instance (Has_Radix t) ⇒ Num (Fixed 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)

lia_signum :: (Ord t, Num t) ⇒ t → t
lia_signum x = if x < 0 then -1 else 1

lia_dim :: (Ord t, Num t) ⇒ t → t → t
lia_dim x y = if x < y then fromInteger 0 else x - y

instance (Has_Radix t) ⇒ Real (Fixed t)
  where
    toRational (FP m s) =
      if s ≥ 0 then m % (shift 1 s)
      else (shift m (negate s)) % 1
```

The % operator divides two integers to make a rational number.

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 $n < 0$, aligning it with the LIA-2 $power_I$ function, but without the breakage of making 0^0 an error. For example,

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

1.4 Conversion to integer

The fromInteger function converts integers to fixed-point; going the other way loses information so we need to say how.

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.

```
data Rounding_Mode
   = Down
   | Up
   | In
   | Out
   | Exact
   | Even
   | 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
                if d /= 0 then i + d else
                if (i + 1 `mod` 4) == 0 then i+1 else i-1
    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 plug Fixed into RealFrac 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. However, we can compose convert with toRational, making it possible to convert fixed-point numbers to different scales and different radices in the spirit of LIA, being like the LIA functions *round* and *trunc*.

```
rescale :: forall t r ∘ (Has_Radix t, Real r) ⇒
          Rounding_Mode → Integer → r → Fixed t

rescale round digits x = FP (convert round y) digits
  where y = toRational x * toRational (radix (undefined :: t) ^ digits)
```

The (convertFixed In) function provides the LIA (floating-point) operations *intpart*, and *fractpart*.

```
convertFixed :: Has_Radix t ⇒
    Rounding_Mode → Fixed t → (Integer, Fixed 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, supporting the LIA-2 *quot*, *mod*, *ratio*, *residue*, *group*, and *pad* functions.

```
quotient :: Has_Radix t ⇒
    Rounding_Mode → Fixed t → Fixed t →
    (Integer, Fixed t)
quotient round x y = (q, x - fromInteger q * y)
  where q = convert round (toRational x / toRational y)
```

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

In a language like COBOL or PL/I, the result of a calculation must fit into the declared type of a variable, while the way that the exact result is rounded to fit may be separately specified. In other languages, like C♯, the scale is part of the value, but there is a maximum scale. We shall call the combination of radix (carried in the type), a desired or maximal scale, a rounding mode, and optional bounds a Fixed_Frame, and the operation of taking an exact result and

coercing it into a frame "fitting". A programming language standard could for example say that "BINARY FIXED (p, s) corresponds to Fixed_Frame Binary (Exact_Scale s) (Nearest Out) (Just (-(2^p), $2^p - 1$))".

```
data Result_Scale
   = Exact_Scale   !Integer
   | Maximum_Scale !Integer

data Fixed_Frame t
   = Fixed_Frame !Result_Scale !Rounding_Mode
                 !Maybe (Integer, Integer)

fit :: Has_Radix t ⇒
       Fixed_Frame t → Fixed t → Fixed t

fit (Fixed_Frame rs round bounds) x@(FP m s)
  = check_bounds bounds m' y
  where
    y@(FP m' _) =
      case rs of
        Exact_Scale e   → if e == s then x else
                          rescale round e x
        Maximum_Scale e → if e ≥ s then x else
                          rescale round e x

check_bounds :: Ord t ⇒ Maybe (t,t) → t → r → r

check_bounds Nothing      _ y = y
check_bounds (Just (l,u)) x y =
  if l ≤ x && x ≤ u then y else error "overflow"
```

1.6 Division

COBOL, PL/I, SQL, Java, C♯, Swift, and most libraries inspected for this work provide division. The COBOL DIVIDE statement can be given a satisfactory definition because the destination and the rounding option provide a Fixed_Frame. PL/I's DIVIDE function also provides a Fixed_Frame for the answer (with only one rounding mode available).

However, no language or library has a satisfactory definition of "/" delivering an intermediate result without constraints provided by a destination. That is because no satisfactory definition exists.

The set of fixed-point numbers is not closed under division. Consider 0.1/0.3. This has an exact rational result, 1/3, but there is no s such that $1/3 \times 10^s$ is an integer. So unlike addition, subtraction, and multiplication, it is not possible for division to provide an answer that is both fixed point and exact.

The best that can be one is to specify division given a frame.

```
divide :: forall t u v ∘ (Has_Radix t, Real u, Real v) ⇒
          Fixed_Frame t → u → v → Fixed t
```

```
divide (Fixed_Frame rs round bounds) x y
  = check_bounds bounds m (FP m s)
  where
    s = case rs of
          (Exact_Scale   z) → z
          (Maximum_Scale z) → z
    p = radix (undefined :: t) ^ s
    q = toRational x / toRational y
    m = convert round (q * fromInteger p)
```

The same problem arises when trying to define a square root function on fixed-point numbers, as some libraries do. The same solution, of requiring an explicit frame. LIA-2 extends *sqrt* to integers, but only with the In rounding mode.

1.7 Converting from text to number

Specifying conversion for Fixed Decimal numbers with their natural scale is tedious rather than difficult. The code that follows is a finite state automaton that accumulates an integer and counts digits after the decimal point. This follows the convention of the Haskell Read class that a string is mapped to a list of (value,residue) pairs with a pair for each legal way to parse a prefix of the string. In this case there is either one way or none.

```
readDecimalFixed :: String → [(Fixed Decimal,String)]
readDecimalFixed ('-':cs) = after_sign negate cs
readDecimalFixed cs       = after_sign id     cs

after_sign f (c:cs)
  | isDigit c = after_digit f (add_digit 0 c) cs
after_sign _ _ = []

after_digit f n (c:cs)
  | isDigit c = after_digit f (add_digit n c) cs
after_digit f n ('.':c:cs)
  | isDigit c = after_dot f 1 (add_digit n c) cs
after_digit f n ('.':_) = []
after_digit f n cs = [(FP (f n) 0, cs)]

after_dot f s n (c:cs)
  | isDigit c = after_dot f (s+1) (add_digit n c) cs
after_dot f s n cs = [(FP (f n) s, cs)]

add_digit :: Integer → Char → Integer

add_digit n c =
   n * 10 + fromIntegral (fromEnum c - fromEnum '0')
```

However, `"0.1"` has no exact representation as a Fixed Binary, nor is the natural scale what a program necessarily needs. What is needed is precisely a

Fixed_Frame.

```
readFixed :: Has_Radix t ⇒
             Fixedλ_Frame t → String →
             [(Fixed t, String)]

readFixed frame cs =
   [ (fix frame num,rest) | (num,read) ← readDecimalFixed cs ]
```