Type-Safety for Inverse Imaging Problems

Maryam Moghadas maryamm@mcmaster.ca
Christopher Kumar Anand anandc@mcmaster.ca
Yuriy Toporovskyy toporoy@mcmaster.ca

Department of Computing and Software, McMaster University
Hamilton, Ontario, Canada L8S 4K1

This paper gives a partial answer to the question: "Can type systems detect modelling errors in scientific computing, particularly for inverse problems derived from physical models?" by considering, in detail, the major aspects of inverse problems in Magnetic Resonance Imaging (MRI). We define a type-system that can capture all correctness properties for basic MRI inverse problems, including many properties that are not captured with current type-systems, e.g., frames of reference. We implemented a type-system in the Haskell language that can capture the errors arising in translating a mathematical model into a linear or nonlinear system, or alternatively into an objective function. Most models are (or can be approximated by) linear transformations, and we demonstrate the feasibility of capturing their correctness at the type level using a difficult case, the (discrete) Fourier transformation (DFT). By this, we mean that we are able to catch, at compile time, all errors known to us in applying the DFT. We describe the Haskell implementation of vector size, physical units, frame of reference, and so on required in the mathematical modelling of inverse problems without regularization.

Computing and Software Report CAS-14-04-CA

The authors thank NSERC, MITACS, Optimal Computational Algorithms, Inc., and IBM Canada for financial support.

Contents

Imagine yourself waiting for a potentially life-saving operation. By using keyhole surgery, your expected recovery time is a small fraction of what it would have been just a few years ago. To get the operation right, your surgeon is relying on plans based on Magnetic Resonance Imaging (MRI). How safe is it to rely on such images and the software used to produce them? Today, the answer is that it depends on the quality of the test plan and the adherence to that test plan. Unfortunately, for real problems created using common software development environments, the combinatorial explosion in possible test cases means that it is very easy to miss cases and fail to detect flaws, e.g., the reversal of left and right in an image. Such are the types of errors detected by one of the authors in commercial software after multiple rounds of government-mandated testing.

How can we do better? How can we leverage the most effective tools we have today? What is the maximum level of certainty we can provide?

In the following paper, we will present evidence that the correctness tool used by the majority of practitioners—static type checking by a compiler—is capable of flagging all of the modelling errors common in image and signal processing. We strongly believe that the same approach could yield static error detection for most errors in scientific computing and suspect that the absence of typing yielding static error detection indicates shortcuts in modelling which may themselves be considered latent errors.

The power of static typing depends on the language used and varies greatly, from simple checking of storage-type compatibility in C to elaborate proofs of correctness embedded into types in Agda Bove et al. [2009]. We chose Haskell Hudak et al. [2007] as our implementation platform because Haskell is a mature language well supported by libraries that allow the encoding of properties such as list length which very few languages can reason about at compile time. We started this work before type-level integers.

Strictly speaking, the only difference with dynamic type checking should be that type errors will arise at run-time rather than compile time. In practice, static type checking is used by many developers in many fields and, at least for Haskell, basic algorithms have long-established soundness arguments, even experimental type features have received public scrutiny and peer review. Dynamic type checking is likely to be developed in an ad hoc manner by domain experts unaccustomed to the types of arguments needed to prove the soundness of the type checker. While it is certainly faster to implement dependent types in a dynamic type checker written for a specific domain, we believe we can achieve our most important goals with stable type extensions to Haskell 2010 today. By providing interesting examples we hope to motivate the continuing development of these type extensions, and extensions for enhancing the utility of embedded Domain Specific Languages (DSLs).

Coming back to our example: static type checking means that a sick patient need never come back for reimaging because of a type error at run-time.

Of course, type checking is not new and manual type-inference for physical units?-which today we call Dimensional Analysis–was identified as a best practice in physics as far back as Rayleigh Rayleigh [1915]. What is new is our attempt to encode sufficient domain knowledge into types so that the process of implementing a software system to solve a mathematical model can also be checked. To the best of our knowledge, the considerable work on type safety falls on one side or the other of this process; either implementing physical units to automate simple unit conversions and to some extent dimensional analysis, or implementing shape constraints on programs so that, for example, array sizes match (or are coerced to do so) and tree branchings correspond.

In the following, we make our case for substantial protection against common errors in scientific programs by way of example. In section 1, we review existing uses of type-level programming to enhance correctness, including cardinality checks and the use of physical units in §1.1. The following section (2) starts by exploring a motivating example, the Fourier Transform (FT), as it is used in inverse imaging problems. While matching vector sizes at compile time may avoid run-time segmentation faults, this example shows that it does not prevent common errors in applying the FT. The rest of the section introduces the concepts we need to avoid these errors: *frames of reference* §2.2 and *discretization via sampling* §2.3. This section shows that such concepts can be encoded so that errors are caught by the compiler or avoided through type inference. We close the section with details of the encoding of floating point numbers and arithmetic, see §2.4.

We chose to start with the FT example because detecting common errors is more valuable than detecting infrequent ones. To really support the development of correct software, error reports have to be clear to the intended domain experts. We found this is not always achieved by the concisest implementations and we discuss the choices we made in §3. Our earlier implementations, see Moghadas [2012], preceded the availability of type-level naturals and implementing types with intelligible errors in that case has influenced our choices in the current implementation. We include two previous implementations for comparison and discuss differences in appendix B.

The Haskell compiler ghc supports several powerful extensions including advanced type-level programming, starting with multi-parameter type classes together with functional dependencies Jones [2000], which have been recast but not completely superseded by type families Kiselyov et al. [2010], data kinds Yorgey et al. [2012], etc. We have included a short appendix, A, so that even domain experts with only a basic knowledge of Haskell can learn to recognize these new features enough to read this paper, use the defined types, and figure out the type errors.

Finally, because type-level programming cannot rely on testing and verification mechanisms available to data-level (normal) programs, we include a third appendix, C, with a summary of our testing strategy.

This is a proof of concept rather than a usable library for two reasons: (1) it only encodes a small part of the explicit and implicit assumptions used to model the physical world using mathematics and (2) it is a type wrapper waiting for types to wrap. The initial implementations wrap **Double**s or [**Double**]s, but the real value of this work will come from wrapping other types, including symbolic expression types used for mathematical modelling and code generation. A prototype library capable of symbolically generating numerical solvers for some inverse problems arising in Magnetic Resonance Imaging, see Pavlin [2012], exists and will be described in a future paper, but we hope our implementation could be used to add a type safety to other libraries, for example DPH Keller et al. [2012], repa Lippmeier et al. [2012], etc. Svensson and Sheeran [2012].

1. Existing Static and Dynamic Type Checking

One basic aspect of type safety in scientific computation is size-matching in linear algebraic computation. For example, if we have two vectors with different sizes, our type-system should be able to capture the error when we attempt to add those two vectors. This has been demonstrated for lists (see Kiselyov [2005] for a complete implementation based on ideas introduced in Okasaki [1999] and McBride [2002]), but is not yet used by the most downloaded packages, e.g., hmatrix, to do linear algebra and vector computation. In fact, some packages produce results in cases that we would expect at least run-time errors, e.g., when adding two vectors with different lengths. In Haskell, we can augment vector and array types with run-time size information, so we will be able to verify such computational errors at run time (dynamic type checking). For example, we can define a vector by:

```haskell
data Vector a = Vector Int [a] deriving (Show,Eq)
```

Then, we can make this vector type an instance of the Num class to make it possible to add two vectors.

```haskell
instance (Num a) ⇒ Num (Vector a) where
  (Vector nx x) + (Vector ny y) =
               if nx ≡ ny then Vector nx $ zipWith (+) x y
               else error $ "adding␣mismatched␣vectors␣of␣sizes␣"
               ++ show nx ++ "␣and␣" ++ show ny

v1 = Vector 3 [1,2,3]
v2 = Vector 4 [1,2,3,4]
```

This implementation will cause a run time error when we add those two vectors:

```
*** Exception: adding mismatched vectors of sizes 3 and 4
```

As we mentioned, we are interested in catching all errors involving vector computation at compile time (static type checking) which will be discussed later, following the approach of Kiselyov [2005].

1.1. Physical Units.

The use of physical units for type checking and type inference has a long history, they were used by Galileo and Newton and certainly existed prior to that, long before the notion of physical units could be formalized. Although this is well known, it has not been well used. Lord Rayleigh attributed this to the disinterest of mathematicians and the inadequacy of the notation used by engineers Rayleigh [1915]; the same is true today, especially if we read "notation" as syntax of programming languages. As implemented by the Dimensional package, it is possible to check the physical units in arithmetic computation at compile time. Physical (dimensional) information is encoded in types which wrap numerical quantities, allowing the type checker to verify the correctness of operations on those physical quantities at compile time. Using this library can prevent us from performing meaningless computation like adding 2m to 20s. We will give an example using this package after explaining its physical unit encoding.

There are seven basic physical dimensions: length, mass, time, electric current, thermodynamic temperature, amount of substance, and luminous intensity. They can be combined to produce compound dimensions. In the Dimensional package, physical dimensions are represented by the powers of the seven basic dimensions. It implements this using type-level numbers to represent the powers of the basic dimensions. This package uses a type-level encoding called NumTypes which is defined in the Numeric.NumType module of the Dimensional package. Data type Dim collects the seven basic dimensions,

```
data Dim l m t i th n j
```

where each type variable represents the power of its corresponding basic unit, e.g., length squared to represent an area. Type variables l, m, t, i, th, n, and j represent the powers of length, mass, time, electric current, temperature, luminous intensity, and amount of substance respectively. Type synonyms for the types of common quantities are defined, e.g.,

```
type DLength      = Dim Pos1 Zero Zero Zero Zero Zero Zero
```

means m^1, metre to the power one and other basic units to the power zero, using type-level numbers $Zero$, $Pos1$ for 1, etc. Similarly, other built-in type synonyms are

```
type DVolume      = Dim Pos3 Zero Zero Zero Zero Zero Zero
type DOne         = Dim Zero Zero Zero Zero Zero Zero Zero
type DMass        = Dim Zero Pos1 Zero Zero Zero Zero Zero
type DTime        = Dim Zero Zero Pos1 Zero Zero Zero Zero
```

Then, type synonyms for quantities of particular physical dimensions are defined

```
type Dimensionless          = Quantity DOne
type Length                 = Quantity DLength
type Mass                   = Quantity DMass
type Time                   = Quantity DTime
```

and so on.

We cannot present all of the implementation details of this package but it is instructive to look at the implementation of one function, addition:

```
(+) :: Num a ⇒ Quantity d a → Quantity d a → Quantity d a
```

The data type Dimensional encodes both units and quantities in one data type

```
newtype Dimensional v d a = Dimensional a deriving (Eq, Ord, Enum)
```

where v and d are phantom type variables. The phantom type variable d represents the physical dimension of the Dimensional and v distinguishes between units and quantities using one of the two following phantom types:

```
data DUnit
data DQuantity
```

There are type synonyms for units and quantities:

```
type Unit     = Dimensional DUnit
type Quantity = Dimensional DQuantity
```

A Quantity is a number value which has a physical unit and it is represented by the product of a number and a Unit. The '$(* \sim)$' operator is a convenient way to declare quantities as such a product.

```
(*~) :: Num a ⇒ a → Unit d a → Quantity d a
```

We can define two physical quantities representing length and mass with the following definition:

```
h = 2 *~ meter
m = 3 *~ gram
```

To show the power of static type checking for such a computation, we can try to add those two variables, h and m, resulting in a compiler error:

```
Couldn't match expected type 'Numeric.NumType.Pos
                                Numeric.NumType.Zero'
            with actual type 'Numeric.NumType.Zero'
Expected type: Quantity DLength Prelude.Integer
  Actual type: Quantity DMass Prelude.Double
In the second argument of '(+)', namely 'm'
In the expression: h + m
```

As you can see, interpretation of the error message requires knowledge of the Haskell type system and the implementation of the Dimensional package. It is not intuitive to a domain expert (e.g., an applied mathematician), even though this is a relatively simple error. One of our goals is to make the error messages easier for non-programmers to understand. This will guide the design our type-system.

We will not use the Dimensional package in the reminder of this paper.

2. Enhanced Typing for Scientific Computation

We have summarized two general aspects of type safety–container size and physical units–implementated in Haskell. Some aspects of previous implementations are not compatible with our goals, e.g., separation of numbers and units in the Dimensional package which allows unsafe computations with no physical interpretation to be wrapped in types after the computation, thereby bypassing the type checker. Because of this, we implemented our own type wrappers, including two implementations of type-level numbers, integrating previous ideas. To have a satisfactory type-system for verifying scientific/medical computation, we needed additional features in our type-system. One of those features is combining both sizes and units to produce types to capture properties involving the interaction of size and units to produce a new class, called a Frame, which captures sufficiently the properties of a discretization to guarantee the correct use of such discretizations in mathematical models. The combination of these features is much greater than the sum of the parts. In our system, all measurements need to take place in a frame of reference with an origin so that only comparable quantities are combined.

Unlike the Dimensional package, there is no way of separating units from quantities and performing unsafe computation. To be clear, we are proposing methods for making mathematical modelling type safe, which may not be practical or useful for all stages of program generation. For example, a linear equation solver will be consistent for one set of units if it is consistent for other units. It is well-understood how to test it and correctness is relative to machine precision and other properties of the system, rather than absolute. Furthermore, it would be undesirable to recompile it every time it is used for a new application. Conversely, type safety is so valuable for mathematical modelling because other methods–in particular testing–are, at best, difficult to apply.

2.1. **Motivating Example: The Fourier Transform.**

Our first example illustrates the power of our new types to capture errors which are easy to make, partly because it seems cumbersome to fully specify them in natural language or the informal mathematics used by applied mathematicians and scientists. The DFT is widely used in scientific computing and is the basis of Magnetic Resonance Imaging. With type-level sizes, we can capture incorrect applications producing 256-sized arrays from 128-sized inputs. With physical units, we can capture errors in which tissue density and velocity data are erroneously added together before or after the DFT, but neither is capable of detecting the most common and hard to detect errors. This includes scaling problems related to the Nyquist Theorem and erroneously combining data in time- and frequency-space. We can capture all of these errors and even use type inferencing to infer missing type information (e.g., sample resolution). To appreciate the value of this correctness checking, we first review the properties of the DFT, beginning with sampling theory.

Recall that there are different FTs for functions of the real line, period functions of the real line, sequences, and periodic sequences. Physics usually demands real numbers as the domain but, computationally, the discrete transform is much more efficient. Rather than causing a general degradation in fidelity, the approximation errors due to replacing the continuous transform with the discrete transform can be catastrophic in the form of aliasing but they are understood and avoidable, as explained below.

Sampling is the process of converting a continuous signal into a numeric sequence so we can process the information by computer. The FT of a continuous signal $x(t)$, $x : \mathbb{R} \to \mathbb{R}$ is defined by:

$$X(f) = \int_{-\infty}^{\infty} x(t) \cdot e^{-2\cdot\pi\cdot i\cdot f\cdot t} dt \tag{1}$$

where $t, f \in \mathbb{R}$. The analogous transform for sampled signals $\{x_n : n \in \{0, ..., N-1\}\}$ is

$$X_k = \sum_{0}^{N-1} x_n \cdot e^{-i\cdot 2\cdot\pi\cdot k\cdot n/N} \tag{2}$$

In the following, we show how to use the DFT to approximate the FT of a continuous signal $x(t)$ and how aliasing errors arise.

In Fig. 1, part (a) represents a band-limited signal $x(t)$, part (b) represents the FT of that signal $X(f)$ which is zero outside the interval $(-f_m < f < f_m)$ because it is band-limited, part (c) represents the periodic unit impulse train, part (d) illustrates the FT of the impulse train illustrated in (c), part (e) represents the sampled version of $x(t)$ and part (f) represents the FT of the sampled version of $x(t)$. The samples $x_s(t)$ can be viewed as the product of the function $x(t)$ with a periodic impulse train (Fig. 1.(c)). The train is represented as Sklar [1988]:

$$x_\delta(t) = \sum_{-\infty}^{\infty} \delta(t - nt_s) \ , \tag{3}$$

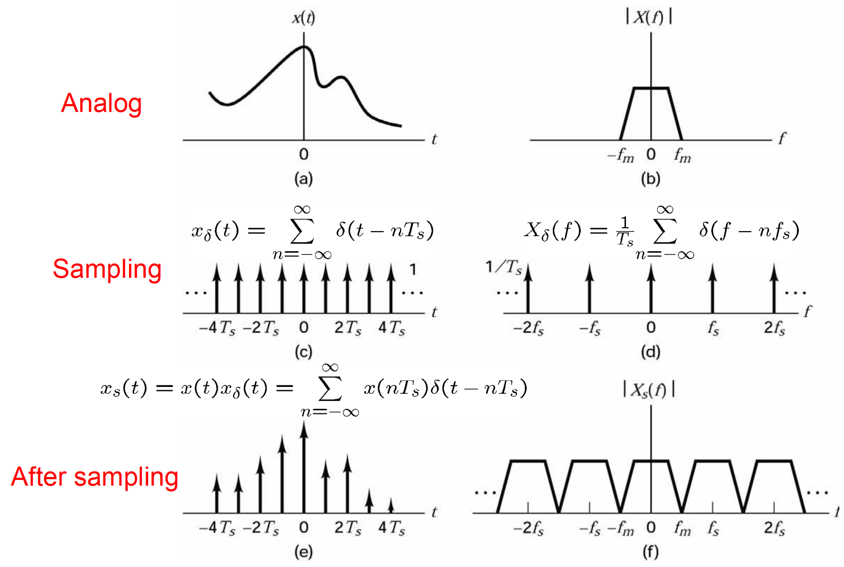
where t_s is the sampling period. We abuse notation by writing $\delta(t) : \mathbb{R} \to \mathbb{R}$ as the unit impulse "function" although is is formally a distribution, which means

$$x(t) \cdot_{\text{(pointwise product)}} \delta(t - t_0) = x(t_0) \cdot_{\text{scaling}} \delta(t - t_0). \tag{4}$$

The sampled version of $x(t)$ can be represented by:

$$x_s(t) = x(t)x_\delta(t) = \sum_{n=-\infty}^{\infty} x(t)\delta(t - nt_s) = \sum_{n=-\infty}^{\infty} x(nt_s)\delta(t - nt_s) \tag{5}$$

Using the Convolution Theorem, the pointwise multiplication in the time domain (5) is equivalent to convolution in frequency domain $X(x) * X(x_\delta) = X(x) * f_\delta$,



Analog
Sampling
After sampling
$x(t)$
t
0
(a)
$|X(f)|$
f
$-f_m$ 0 f_m
(b)
$x_\delta(t) = \sum_{n=-\infty}^{\infty} \delta(t - nT_s)$
1
$-4T_s$ $-2T_s$ 0 $2T_s$ $4T_s$
(c)
$X_\delta(f) = \frac{1}{T_s} \sum_{n=-\infty}^{\infty} \delta(f - nf_s)$
$1/T_s$
$-2f_s$ $-f_s$ 0 f_s $2f_s$
(d)
$x_s(t) = x(t)x_\delta(t) = \sum_{n=-\infty}^{\infty} x(nT_s)\delta(t - nT_s)$
(e)
$|X_s(f)|$
$-2f_s$ $-f_s$ $-f_m$ 0 f_m f_s $2f_s$
(f)

Figure 1. Fourier Transform

where f_δ, is the impulse train with step $f_s = 1/t_s$, which is the FT of x_δ,

$$f_\delta = \frac{1}{t_s} \sum_{n=-\infty}^{\infty} \delta(f - nf_s) \tag{6}$$

Another property of convolution is that convolution with an impulse function shifts the original function:

$$X(f) * \delta(f - nf_s) = X(f - nf_s) \tag{7}$$

Hence, the FT of the sampled version of $x(t)$, $X_s(f)$, is represented by:

$$X_s(f) = X(f) * X_\delta(f) = X(f) * \left[\frac{1}{t_s} \sum_{n=-\infty}^{\infty} \delta(f - nf_s)\right] = \frac{1}{t_s} \sum_{n=-\infty}^{\infty} X(f - nf_s) \tag{8}$$

In conclusion, the FT of the sampled signal is the sum of multiple shifted spectra–of the original signal–centred at the sampling frequency and its harmonics. As illustrated in Fig. 1.(f), if $f_s = 2f_m$ then there is no overlap (aliasing) between the multiple spectra. The Nyquist theorem states that any band limited signal $x(t)$ that has no frequency component for $f > f_s$ can be uniquely reconstructed from its samples $X(nT)$ if the sampling frequency satisfies $f_s > 2f_m$ which is called the Nyquist criterion. This criterion is one of the important FT properties which should be encoded in our the type-system because when the sampling resolutions exactly satisfy the Nyquist criterion the sampled spaces connected by the *Fast* FT will not exhibit aliasing.

In experimental MRI, tissue density information is not directly measurable. Instead, MRI experiments collect samples of the continuous FT of the tissue density and other tissue properties of interest. The dual-space to physical space is called k-space, which is the domain of the FT. Note that, unlike applications of the FT

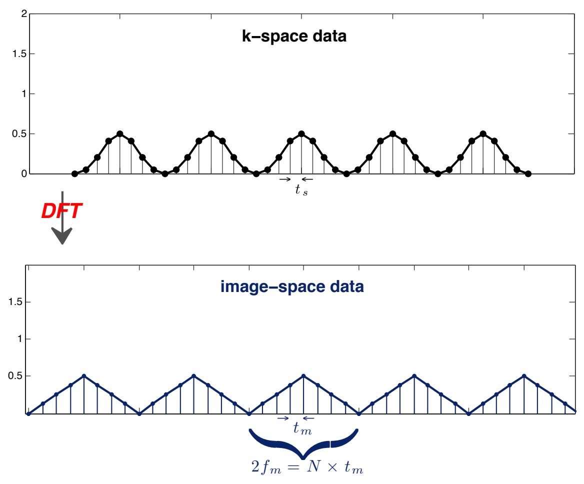


Figure 2. K-space and Image-space Data

to sound production, MRI requires multi-dimensional transforms (3D for normal space, 2D for planar imaging, and potentially higher dimensions for velocity, diffusion and spectrographic imaging). Correctness in MRI reconstructions depends on precisely encoding the meaning of data at the type level to prevent errors in processing, from the obvious error of mixing k-space and image-space data to subtle errors involving incompatible resolutions in k- and image-space, which would create aliasing or images with the wrong scale.

First, the number of samples (resolution) should be equal in both k- and image-space which is represented by N. Second, encoding the Nyquist criterion into our type-system results in the appropriate relation between sampling rates in k- and image-space. According to Fig. 1, the FT of a non-periodic discrete function $x_s(t)$ is a periodic continuous function. The $X_s(f)$ is a continuous signal in the frequency domain which should be discretized to be processed numerically. Likewise, the sampling in the time domain, the sampled version of $X_S(f)$, can be obtained by multiplying by another impulse train. Such multiplication in the frequency domain affects the original signal in the time domain by making it periodic. Hence we can represent the sampled data in k- and image-space by Fig. 2.

In the image-space:

$$N : \text{number of samples}$$
$$t_m : \text{the sampling step size in the image-space}$$
$$2f_m : \text{the bandwidth collected in k-space},$$

where image-width $= N \times t_m$.

In k-space:

$$N : \text{number of samples}$$
$$t_s : \text{the sampling step size in the k-space}$$
$$t_s = 1/f_s$$

According to the Nyquist criterion $2f_m = f_s$, therefore:

$$N \times t_m \times t_k = 1 \tag{9}$$

This enforces constraints on k- and image-space step sizes. To encode this constraint in our type-system, we need to first introduce the concept of a Discretization and use it to define an FT class.

As an example, we consider a discretized function (meas1) and infer the type of its FT, then we observe that their step-sizes satisfy (9). The discretized function meas1 has type:

```
meas1 :: Discretization1D  (F "LabFrame")
                           (NAT 4)
                           [float|0.002|]
                           Meter
                           [Complex Double]
```

We have not yet introduced our type-level numbers but, for example, NAT 3 is a type-level natural which represents the number 3. [float|0.002|] is a Template Haskell expression which will be expanded to FLOAT Pos 2 (E_ 3), which represents the number $2 * 10^{-3}$.

We can get the type of its FT from the ghci interpreter:

```
>:t ft meas1
ft meas1
  :: Discretization1D
       (F "LabFrameT")
       (NAT 4)
       (FLOAT 'Pos 125 ('E 0))
       (SIUnit ('M 1) ('S 0) ('Kg 0) ('A 0) ('Mol 0) ('K 0))
       [Complex Double]
```

The number of samples in both cases is 4. Step-size is represented as an infinite precision, floating point number. Therefore, the step-size of meas1 is 0.002m and the step-size of ft meas1 is 125(1/m) which satisfies (9) ($4 \times 0.002\text{m} \times 125(1/\text{m}) = 1$).

Having motivated their usefulness, we now introduce our types in logical order.

2.2. **Frames.** A frame of reference is a coordinate system or a vector space basis together with physical units and an origin from which to measure some characteristics of objects in it. In the existing type systems, there is no way of recording the frame of reference. To formalize it in our type-system, we define a class whose instances are frames:

```
data F (s :: Symbol) = F

class (KnownUnit (BaseUnit frame)) ⇒ Frame frame where
  type BaseUnit frame :: * → *
  name :: frame → String
  default name :: (KnownSymbol x, frame ~ F x) ⇒ frame → String
  name = symbolVal
```

Note that the units are explicitly encoded on the BaseUnit but the basis and origin are implicitly encoded. The constraint in the class declaration requires that BaseUnit be an instance of KnownUnit which is an assertion that the type is a valid unit.

As a reminder, a vector space is a set of vectors on which two operations are defined, vector addition and scalar multiplication and it should satisfy several axioms with respect to these two operations. Moreover, given any vector space V over a field F, the dual space V^* is defined as the set of all linear maps $\phi : V \to F$.

Because a frame is mathematically similar to a vector space, we can define the dual frame concept analogous to the dual vector space. Hence, for any given `Frame a`, the set of linear maps from `Base a` to `Unitless` is the dual frame of `Frame a`. To formalize the definition of dual frame, first we need to formalize the duality between units:

```
type AssertDualUnits a b = Assert (a :*: b ≡? Unitless) (DualUnits a b)
```

where `:*:` is unit multiplication. The `Assert` (defined in §3) type allows for better control over the way error messages are printed.

Using this definition, we can assert the duality between frames by:

```
class (Frame a, Frame b, AssertDualUnits (BaseUnit a) (BaseUnit b)
      ) ⇒ AssertDualFrames a b | a → b, b → a
```

where, in our type-system, the user has to specify the frame of reference and assert duality between frames such that if one frame changes alone it will cause a compile-time error.

The functional dependency, `a → b, b → a`, asserts that taking the dual is a bijection on the set of frames, see appendix A for a longer explanation.

The user must then declare the proper frames and make the assertion that they are dual:

```
instance Frame (F "LabFrame") where
  type BaseUnit (F "LabFrame") = Meter

instance Frame (F "LabFrameT") where
  type BaseUnit (F "LabFrameT") = Recip Meter
  name _ = "LabFrameDual"

instance AssertDualFrames (F "LabFrame") (F "LabFrameT") where
```

If the user defines a frame improperly and attempts to assert the duality of the improperly defined frames

```
instance Frame (F "BadFrame0") where
  type BaseUnit (F "BadFrame0") = Unitless

instance Frame (F "BadFrame1") where
  type BaseUnit (F "BadFrame1") = Meter

instance AssertDualFrames (F "BadFrame1") (F "BadFrame0") where
```

they will meet this compile time error:

```
    No instance for
        (DualUnits
           (SIUnit ('M 1) ('S 0) ('Kg 0) ('A 0) ('Mol 0) ('K 0))
           (SIUnit ('M 0) ('S 0) ('Kg 0) ('A 0) ('Mol 0) ('K 0)))
      arising from the superclasses of an instance declaration
    In the instance declaration for
      'AssertDualFrames (F "BadFrame1") (F "BadFrame0")'
```

The source of the problem is clear from the error.

To justify the necessity of formalizing the frame into an enhanced type-system, consider the orientation in medical imaging. Image orientation identifies the spatial orientation of the imaging plane with respect to the patient. The orientation is important to the correct diagnosis. Because, when a physician examines a medical image related to a patient's leg, they need to know whether it is the left or right leg; otherwise an operation may be done on the wrong leg. Currently, this level of correctness cannot be guaranteed by the existing type systems. This property of imaging is formalized using the `Frame` in our type-system.

Consider a discretized measurement meas1 made in the LabFrame.

```
meas1 :: Discretization1D  (F "LabFrame")
                           (NAT 4)
                           [float|0.002|]
                           Meter
                           [Complex Double]
meas1 = Discretization1D [0,1,2,3]
```

The FT of meas1 is relative to the dual frame of LabFrame, meaning that adding meas1 with its FT is not a valid operation because they are not co-measurable. So, whereas a conventional language which does size checking would allow this operation, we flag a type error:

```
>add (ft meas1) meas1
  No instance for
   (Add
      (Discretization1D
         (F "LabFrameT")
         (NAT 4)
         (FLOAT 'Pos 125 ('E 0))
         (SIUnit ('M 1) ('S 0) ('Kg 0) ('A 0) ('Mol 0) ('K 0))
         [Complex Double])
      (Discretization1D
         (F "LabFrame")
         (NAT 4)
         (FLOAT 'Pos 2 ('E_ 3))
         (SIUnit ('M 1) ('S 0) ('Kg 0) ('A 0) ('Mol 0) ('K 0))
         [Complex Double])
```

2.3. **Discretizations and FTs.** The Discretization data type is defined as:

```
data Discretization1D frame numSamples stepSize rangeU val where
     Discretization1D ::
        (Frame frame, IsNat numSamples, IsFloat stepSize,
         rangeU ~ SIUnit m1 s1 kg1 a1 mol1 k1
         ) ⇒ val → Discretization1D frame numSamples stepSize rangeU val
```

where frame is an abstract vector space basis; numSamples refers to the number of samples; stepSize specifies the step-size in the units attached to the frame; rangeU is the physical unit of the discrete values; and val represents the discrete values.

We start with an application of this enhanced data type.

To increase the signal to noise ratio, one may take the average of multiple samplings of the signal of interest. There are some constraints on the sampling to have a meaningful average. For example, if two samplings have the same number of samples but different sampling steps, then adding those two samplings does not make sense. Suppose we have two samplings of the height of water in a canal illustrated in Fig. 3. In the blue/solid sampling, there are 12 samples and the sampling step is 0.01. In the red/dashed sampling, there are 12 samples, but the sampling step is 0.005. Those two samplings can be defined in our type system using the Discretization data type:

```
canalSample1 :: Discretization1D (F "CanalFrame")
                                 (NAT 12)
                                 [float|0.01|]
                                 Meter
                                 [Double]

canalSample1 =
```

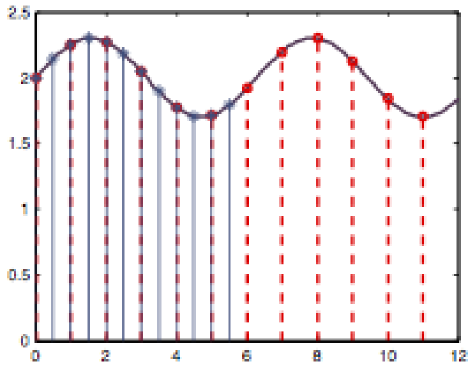


Figure 3. Sampling of height of water in a canal

```
  Discretization1D [1,1.2,1.3,1.12,1.23,1.12,1.15,1.25,1.18,1.20,1.24,1.28]


canalSample2 :: Discretization1D (F "CanalFrame")
                                 (NAT 12)
                                 [float|0.02|]
                                 Meter
                                 [Double]

canalSample2 =
  Discretization1D [1,1.21,1.2,1.42,1.3,1.32,1.12,1.25,1.23,1.20,1.12,1.28]
```

Trying to add those two samplings to each other

```
> canalSample1 U.+ canalSample2
```

will cause a compile time error:

```
  No instance for
    (Add
       (Discretization1D
          (F "CanalFrame")
          (NAT 12)
          (FLOAT 'Pos 1 ('E_ 2))
          (SIUnit ('M 1) ('S 0) ('Kg 0) ('A 0) ('Mol 0) ('K 0))
          [Double])
       (Discretization1D
          (F "CanalFrame")
          (NAT 12)
          (FLOAT 'Pos 2 ('E_ 2))
          (SIUnit ('M 1) ('S 0) ('Kg 0) ('A 0) ('Mol 0) ('K 0))
          [Double])
```

The error message indicates that the discretizations have different types and adding them is not a valid operation. Ideally, we would prefer the error message to say the sample spacing needs to be identical but it would not be difficult for a domain expert to understand this error.

As we mentioned, another powerful feature in our type system is the FT class which is defined by:

```
class FT a b | a → b, b → a where
  ft :: a → b
  invFt :: b → a
```

To implement an instance, we assert the preconditions required by the Discretization1D constructor:

```
instance (
    AssertDualFrames frame1 frame2, Frame frame1, Frame frame2,
    IsFloat stepSize2, IsFloat stepSize1, ToFloat numSamp ~ numSampF,
```

and encode the Nyquist criterion:

```
    MultNZ stepSize1 numSampF t0,
    MultNZ t0 stepSize2 t1,
    t1 ~ FLOAT Pos 1 (E 0)

    ) ⇒
    FT (Discretization1D frame1 numSamp stepSize1 rangeU [Complex Double])
       (Discretization1D frame2 numSamp stepSize2 rangeU [Complex Double])
        where
            ft (Discretization1D x) = (Discretization1D $ FFT.fft x)
            invFt (Discretization1D x) = (Discretization1D $ FFT.ifft x)
```

where fft and ifft come from the pure−fft package. Two parameters of that type class represent a discretized function and its FT. Using the multi-parameter type classes together with functional dependency, it is possible to infer the type of the FT of a discretized function if it knows the type of the discretized function itself and vice versa. In the instance definition, there are some important constraints on the two parameters of the FT class which are related to FT properties. First, the number of samples in those two parameters should be the same, which is encoded by clarifying the same numSamp for those discretized parameters. Second, the discretized functions (class parameters) are related to two dual frames, and this duality must be asserted by the user. The most subtle constraint is encoded via two instances of the MultNZ class. This constraint specifies that the product of the first two parameters should be equal to the third parameter. To allow for type inferencing, we must also assert that the multiplication is nonzero. Since equation (9) involves a triple product, we need two multiplications to assert it.

While this definition is limited to one dimension, we can encode the same relationship in multiple dimensions. Similarly to the way multi-dimensional FTs are usually implemented, we construct multi-dimensional frames by composing single-dimensional, orthogonal frames. To each dimension is associated the value's step size, number of samples, and unit:

```
infixr 5 :>
data CompositeFrame xs where
  F0    :: CompositeFrame '[]
  (:>)  :: (F sym, NAT x, FLOAT sn y ex, Proxy (SIUnit m s kg amp mol k))
        → CompositeFrame xs
        → CompositeFrame
            ( '(sym, NAT x, FLOAT sg y ex, SIUnit m s kg amp mol k) ': xs )
```

In the one dimensional example, the parameters were phantom type variables; however, due to a feature of the current type checker, we must have a concrete representation of the structure of the composite frame.

The dimension of the data within the a discretization must match the dimension of the frame. We can compute the exact type of the data from the CompositeFrame:

```
type family Dimension (n :: k) (f :: ∗ → ∗) (v :: ∗) :: ∗ where
  Dimension '[]        f a = a
  Dimension (x ': xs)  f a = f (Dimension xs f a)
```

For example, Dimension '[x,y] [] (**Complex Double**) is [[**Complex Double**]]. Then the discretization is defined simply as:

```
data Discretization xs vec val where
  Discretization :: CompositeFrame xs
```

```
                  → Dimension xs vec val
                  → Discretization xs vec val

frameOf :: Discretization frame vec val → CompositeFrame frame
frameOf (Discretization f _) = f

dataOf :: Discretization frame vec val → Dimension frame vec val
dataOf (Discretization _ d) = d
```

where `xs` is the composite frame, `vec` is the datatype in which the data is contained–here it is `[]`, the list type–and `val` is the underlying value of the data, here it is **Complex Double**.

We must declare frames in the same way as before:

```
instance Frame "LabFrameX" where
  type BaseUnit "LabFrameX" = Meter

instance Frame "LabFrameY" where
  type BaseUnit "LabFrameY" = Meter

instance Frame "LabFrameXT" where
  type BaseUnit "LabFrameXT" = Recip Meter

instance Frame "LabFrameYT" where
  type BaseUnit "LabFrameYT" = Recip Meter

instance AssertDualFrames "LabFrameX" "LabFrameXT"
instance AssertDualFrames "LabFrameY" "LabFrameYT"
```

In order to make it easier to declare discretizations, we include helper functions and type synonyms for discretizations of dimension up to six. Higher dimensions are of course permitted, but the user must either declare their own type synonyms or use the `Discretization` constructor directly, which will require more typing and produce more confusing type errors. For example, for two dimensions:

```
type Discretization2D dim0 dim1 =
     Discretization [dim0, dim1]

discretization2D = Discretization (someFrame :> someFrame :> F0)
```

where `someFrame` is a polymorphic value representing any frame, whose type will be specialized when discretizations are declared.

We can now declare some example data:

```
labFrame ::   Discretization2D
                '("LabFrameX", NAT 8, [float|5e-2|], Meter)
                '("LabFrameY", NAT 4, [float|1e-2|], Meter)
                []
                (Complex Double)

labFrame = Discretization2D
                [[-0.2  ,-0.26  ,-0.32  ,-0.38  ,-0.44  ,-0.5  ,-0.56  ,-0.63]
                ,[0.2   ,0.26   ,0.32   ,0.38   ,0.44   ,0.5   ,0.56   ,0.76  ]
                ,[0.6   ,0.78   ,0.96   ,1.14   ,1.32   ,1.5   ,1.68   ,1.98  ]
                ,[1.0   ,1.3    ,1.6    ,1.90   ,2.2    ,2.5   ,2.8    ,2.94  ]]
```

The FT can be parametrized over the direction:

```
data Dir = Forward | Inverse

data Direction (d :: Dir) where
  ForwardDir :: Direction Forward
  InverseDir :: Direction Inverse
```

which allows a class to compute the dual frame of a given frame based on the direction and a function to pick the appropriate FFT function from the numeric−FFT package:

```
class DualFrameOf dir a b | dir a → b
instance AssertDualFrames a b ⇒ DualFrameOf Forward a b
instance AssertDualFrames b a ⇒ DualFrameOf Inverse a b

ftFunc :: Direction dir → [Complex Double] → [Complex Double]
ftFunc ForwardDir = fft
ftFunc InverseDir = ifft
```

In order to perform an FFT on an arbitrary dimension of an n-dimensional matrix, we have to be able to index the dimensions. In order to do this counting, we store a CompositeFrame which will later be used to determine the correct 'depth'.

Two classes are required to implement the recursive search of the required frame:

```
class DirectionFT1' dir db frame dim    frame' dim'
                  | dir db frame dim → frame' dim' where

  directionFT1' :: Direction dir → F db → CompositeFrame frame
                → CompositeFrame dim →
                 (CompositeFrame frame', CompositeFrame dim')
```

and

```
class DirectionFT1'' bool dir db frame dim    frame' dim'
                   | bool dir db frame dim → frame' dim' where

  directionFT1'' :: Proxy bool → Direction dir → F db
                 → CompositeFrame frame → CompositeFrame dim →
                  (CompositeFrame frame', CompositeFrame dim')
```

The class DirectionFT1'' will decide if the target frame is the same as the frame at the front of the composite frame and pass the appropriate value to DirectionFT1'.

```
instance (bool ~ (fr ≡? fr'),
          DirectionFT1'' bool dir fr' ('(fr, n, s, ru) ': xs)
                         dim (y ': ys) dim'
         ) ⇒ DirectionFT1' dir fr' ('(fr, n, s, ru) ': xs)
                           dim (y ': ys) dim' where
  directionFT1' = directionFT1'' (Proxy :: Proxy bool)
```

In the case that the target frame and head of the composite frame of the Discretization are the same, return the current counter and compute the new type of the frame.

```
instance (IsNat numSamples, DualFrameOf dir frame frameD,
          stepSizeD ~ FLOAT s0 n0 e0,

          ToFloat numSamples ~ numSamplesF,
          stepSizeD ~ RecipFloat (numSamplesF * stepSize)

         ) ⇒ DirectionFT1'' True dir fr
                 ( '(frame , numSamples, stepSize , rangeU ) ': xs) dim
                 ( '(frameD, numSamples, stepSizeD , rangeU ) ': xs) dim
    where
  directionFT1'' _ _ _ (_ :> xs) n = ((F, NAT, FLOAT, Proxy) :> xs, n)
```

In the case that the target frame and head of the composite frame of the Discretization are different, recurse on the rest of the composite frame and increment the counter by one.

```
instance (DirectionFT1' dir fr xs dim ys dim',
          d ~ '(frame, numSamples, stepSize, rangeU )
         ) ⇒ DirectionFT1'' False dir fr
                  ( '(frame, numSamples, stepSize, rangeU ) ': xs) dim
                  ( '(frame, numSamples, stepSize, rangeU ) ': ys) (d ':
                  dim')
   where
  directionFT1'' _ dir f (x :> xs) dim =
     let (ys, dim') = directionFT1' dir f xs dim in (x :> ys, x :> dim')
```

In order to actually perform the FFT, we need to know the depth of the target dimension and the overall dimension of the matrix. Then there is a pattern to compute the FFT from these values:

D1	x	**map**^0 **transpose** . fft^1 . **map**^0 **transpose**
D2	x	**map**^0 **transpose** . fft^2 . **map**^0 **transpose**
	y	**map**^1 **transpose** . fft^2 . **map**^1 **transpose**
D3	x	**map**^0 **transpose** . fft^3 . **map**^0 **transpose**
	y	**map**^1 **transpose** . fft^3 . **map**^1 **transpose**
	z	**map**^2 **transpose** . fft^3 . **map**^2 **transpose**

where `f^0` is **id**, `f^1` is `f`, `f^2` is `f . f`, etc. This function exponentiation is provided by dimension, which is `f^n`, and dimensionPred, which is `f^n−1`.

```
dimension :: (Dimension x f v ~ fv, Functor f) ⇒
          Proxy f → CompositeFrame x → (v → v) → fv → fv
dimension _ F0         f a = f a
dimension p (_ :> n)   f a = fmap (dimension p n f) a

dimensionPred :: (DimensionPred x f v ~ fv, Functor f) ⇒
              Proxy f → CompositeFrame x → (v → v) → fv → fv
dimensionPred _ F0        _ a = a
dimensionPred p (_ :> x) f a = dimension p x f a
```

In order to actually compute the FFT relative to a frame, that frame must be in the composite frame. We check this precondition.

```
type AssertAtomicSubframe x xs = Assert (IsAtomicSubframe x xs)
(AtomicSubframe x xs)

type family IsAtomicSubframe (x :: Symbol) (xs :: [ (Symbol, k0, k1, k2)
]) :: Bool where
  IsAtomicSubframe x '[]                          = False
  IsAtomicSubframe x ( '(x, k0, k1, k2) ': xs) = True
  IsAtomicSubframe x ( '(y, k0, k1, k2) ': xs) = IsAtomicSubframe x xs
```

The top level FFT class encodes the base case of a one dimensional discretization and the general case.

```
class (AssertAtomicSubframe db frame
      ) ⇒ DirectionFT1 dir db frame    frame'
                     | dir db frame → frame' where
  directionFT1 :: Direction dir → F db
               → Discretization frame  [] (Complex Double)
               → Discretization frame' [] (Complex Double)
```

The general case computes the depth of the target frame, then creates the transposition and FFT which apply to the proper dimensions.

At this point we compute the actual function to be applied −fft or ifft from the direction parameter.

```
instance
   (frame ~ (x ': xs), frame' ~ (y ': ys),
    AssertAtomicSubframe db (x ': xs),
    DirectionFT1' dir db frame '[] frame' dim',
```

Some extra constraints are required in order to convince the compiler that our types really do match up. The user is not required to satisfy these, rather, they are tautologies.

```
     DimensionPred dim' [] [[a]] ~ [Dimension xs [] (Complex Double)],
     Dimension ys [] (Complex Double) ~ Dimension xs [] (Complex Double),
     Dimension xs [] [Complex Double] ~ [Dimension xs [] (Complex Double)]
  ) ⇒ DirectionFT1 dir db (x ': xs)
                          (y ': ys) where

  directionFT1 dir db (Discretization frame val) =
   Discretization frame' (tp . ft . tp $ val)
    where (frame', (dim :: CompositeFrame dim')) =
             directionFT1' dir db frame F0

          tp = dimensionPred (Proxy :: Proxy [])
                 dim (transpose :: [[a]] → [[a]])
          ft = dimensionPred (Proxy :: Proxy [])
                 frame (ftFunc dir)
```

Once the base case exists, we can write the more general case of taking the FFT with respect to multiple dimensions. This is comprised of recursing over the type-level list representing the frames and taking the 1D FT with respect to each one.

```
class DirectionFT dir xs frame frame' | dir xs frame → frame' where
  directionFT :: Direction dir
              → List xs
              → Discretization frame  [] (Complex Double)
              → Discretization frame' [] (Complex Double)

instance DirectionFT dir '[] frame frame where ...

instance (DirectionFT1 dir x frame frame',
          DirectionFT dir xs frame' frame''
         ) ⇒ DirectionFT dir (F x ': xs) frame frame'' where ...
```

Once there is a general form of the FT, getting the inverse or forward FT just requires passing the correct argument to the general function. We define some functions for convenience:

```
forwardFT1 = directionFT1 ForwardDir
inverseFT1 = directionFT1 InverseDir
forwardFT  = directionFT  ForwardDir
inverseFT  = directionFT  InverseDir
```

Some examples of this approach are included in the appendix.

2.4. **Detailed Type-Level Implementation.**

2.4.1. *Numbers Types.* Now, we explain some interesting implementation details of our type-system. First of all, we present the type-level numbers encoded into our type-system. There are several encodings of the type-level number including Peano numbers, binary encoding, and so on. Originally, we used a phantom type representation of a sequence of decimal digits. With the introduction of type-level naturals and arithmetic in the latest Haskell compiler (GHC 7.8), much of the code has been greatly simplified.

We provide three different type-level number types: NAT, INT, and FLOAT. The type NAT is a trivial wrapper of the built-in type-level naturals–we implement it so that naturals can have values, see appendix A.

First, there is a common interface to the different types of numbers, implemented using classes and open type families.

```
class Compare a b (c :: Ordering) | a b → c where
  compare' :: a → b → Proxy c
  compare' _ _ = Proxy

class Add a b c | a b → c, b c → a, a c → b where
  add :: a → b → c

class Mult a b c | a b → c where
  mult :: a → b → c

class (NonZero a, NonZero b, NonZero c) ⇒
  MultNZ a b c | a b → c, b c → a, a c → b where
    multNZ :: a → b → c

⋮
```

In addition to Mult, we define an extra multiplying class, MultNZ, with extra constraints such that functional dependency works in all directions, meaning that knowing the type of any two elements of the triple $(a; b; c)$ gives the type of third one.

Type inference would not work if some sizes were allowed to be zero, since any x satisfies $x \cdot 0 = 0$.

The numeric types are defined as single constructors with phantom type parameters:

```
data NAT (n :: Nat) = NAT

data Sign = Neg | Pos
data INT (s :: Sign) (n :: Nat) = INT

data E_K = E Nat | E_ Nat
data FLOAT (s :: Sign) (n :: Nat) (exp :: E_K) = FLOAT
```

Note that the type parameters have specific kind signatures, which makes writing a nonsensical type like INT :: INT **Char** **Float** is a type error, see appendix A.

Also note that the type E_K is the same as INT. While this causes some code duplication, one goal when writing the definitions of number datatypes was to produce easily readable types when the compiler prints them.

The type FLOAT represents arbitrary precision floating point decimal numbers. The numbers are represented as $sign * base * 10^{exp}$; E_K is the kind of exponents and can be a negative exponent, E_ n, or a positive exponent, E n.

If the user declares their numbers using Template Haskell (see appendix A) or uses the 'smart constructor' NFLOAT:

```
type family NFLOAT (s :: Sign) (n :: Nat) (exp :: E_K) where
  NFLOAT s n exp = NormalizeFloat (FLOAT s n exp)
```

the resulting number is guaranteed to be in normal form. Otherwise, the user must assure that their declared numbers are indeed in normal form.

Normalizing floats consists of four steps: the base is converted to a list of digits, trailing zeroes are dropped, the list is combined back into a single number, and the exponent is adjusted by the appropriate number of digits.

The most important part of this process is converting the base, x, to a list of digits. We count the number of digits in the base, d:

```
type NumDigits num = NumDigits' 1 0 False (num + 1)
type family NumDigits' pow powCount bool num where
  NumDigits' pow powCount True num = powCount
  NumDigits' pow powCount b    num =
     NumDigits' (pow*10) (powCount+1) (num ≤? (pow * 10)) num
```

and then successively add 10^{n-1}, counting how many exponents were added:

```
type family DigitSplit''' num dig pow where
  DigitSplit''' num dig pow =
    DigitSplit'''' num dig pow ((((dig + 1) * (10 ^ (pow - 1))) >? num))

type family DigitSplit'''' num dig pow b where
  DigitSplit'''' num dig pow False  = DigitSplit'''  num (dig + 1) pow
```

until we have exceeded x. At this point, the count, c, is the first digit of x while $x - c * 10^{n-1}$ is the rest of the digits. We recurse on the remaining value.

```
  DigitSplit'''' num dig pow True  =
   dig ': DigitSplit''  (num - (dig * (10 ^ (pow - 1)))) (pow - 1)
```

We perform a bounds check for numbers with only 1 digit, and define the special case `Digits 0 = []`:

```
type DigitSplit num = DigitSplit' num (num ≤? 9)

type family DigitSplit' num bool where
  DigitSplit' num True  = '[num]
  DigitSplit' num False = DigitSplit'' num (NumDigits num)

type family DigitSplit'' num pow where
  DigitSplit'' num  0    =   '[]
  DigitSplit'' num  pow  =  DigitSplit''' num 0 pow
```

2.4.2. *Arithmetic.* We will omit details regarding the implementation of arithmetic for NAT and INT since they rely on the built-in type functions for arithmetic, with type-level naturals, in a similar way to the implementation of FLOAT.

Due to the function dependency on the `Add` class, we are required by the type-checker to supply a proof for $c = a + b$, $a = c - b$, and $b = c - a$, which allows the typechecker to determine any one type from the other two:

```
instance
  (a ∼ (FLOAT s0 n0 e0), b ∼ (FLOAT s1 n1 e1), c ∼ (FLOAT s2 n2 e2),
   NormalizeFloat (AddFloat a                b)  ∼ c ,
   NormalizeFloat (AddFloat c  (NegateFloat  b)) ∼ a ,
   NormalizeFloat (AddFloat c  (NegateFloat  a)) ∼ b
  ) ⇒ Add (FLOAT s0 n0 e0) (FLOAT s1 n1 e1) (FLOAT s2 n2 e2)
         where add _ _ = FLOAT
```

Addition of two summands is computed in the usual way: we compute the difference between the exponents, multiply the smaller base by $10^{|e_0 - e_1|}$, and add the bases:

```
type family AddFloat a b where
  AddFloat (FLOAT s0 n0 e0  )(FLOAT s1 n1 e1  ) =
        AddFloat' (CompareInt (ExpToInt e0) (ExpToInt e1))
            (SubtractInt (ExpToInt e0) (ExpToInt e1))
            (INT s0 n0) (ExpToInt e0)
            (INT s1 n1)  (ExpToInt e1)

type family AddFloat' cmp diff am ae bm be where
  AddFloat' GT diff am ae bm be =
      PowFloat (AddInt (MultInt am (ExpInt (INT Pos 10) diff)) bm)
      (IntToExp be)
  AddFloat' LT diff am ae bm be =
      PowFloat (AddInt am (MultInt bm (ExpInt (INT Pos 10) (NegateInt
      diff)))) (IntToExp ae)
```

where ExpToInt and IntToExp are type functions which convert between E_n/E n and INT. Some patterns for the above functions are omitted; these patterns handle special cases such as zero, one, summands with the same exponent, summands with the same base, and so on. Having these extra patterns optimizes some common cases, speeding computation slightly.

Multiplication is also done the usual way:

```
type family MultFloat a b where
  MultFloat (FLOAT s0 n0 e0) (FLOAT s1 n1 e1) =
    NormalizeFloat (FLOAT (MultSign' s0 s1)
                          (n0*n1)
                          (IntToExp (ExpToInt e0 `AddInt` ExpToInt e1)))
```

Multiplication of non-zero multiplicands is more interesting, as it requires floating point division to prove to the compiler that the functional dependencies are satisfied - namely, that $a * b = c$, $a = \frac{c}{b}$, and $b = \frac{c}{a}$.

Due to technical limitations, floating point division would take a prohibitively long time since it requires integer division, which is not present for built-in type-level naturals. Integer division with remainder requires repeatedly subtracting the divisor, checking after each subtraction that the result isn't less than the divisor, and increasing an accumulator for the quotient. This is at least three type family applications and in practice more likely five. Each type family application has an associated cost. Even for small numbers divisions, like 1000/2, this requires at least 2500 type family applications, which is quite expensive, and will overflow without increasing the −ftype−function−depth compiler option, which is the maximum recursion depth for type families.

We can work around this, employing a divisions of the form $1/a$. However, it would be preferrable to have division with remainder built into the compiler; this would make floating point division feasible.

```
instance
  (a ~ (FLOAT s0 n0 e0), b ~ (FLOAT s1 n1 e1), c ~ (FLOAT s2 n2 e2),
   NonZero a, NonZero b, NonZero c,
   MultFloat a             b  ~ c,
   MultFloat c (RecipFloat b) ~ a,
   MultFloat c (RecipFloat a) ~ b
  ) ⇒ MultNZ (FLOAT s0 n0 e0) (FLOAT s1 n1 e1) (FLOAT s2 n2 e2)
        where multNZ _ _ = FLOAT
```

The consequence of this is that some multiplications will fail because an intermediate number may not be expressible as a decimal–for example, $2 * 3$ requires computing $6 * (1/3)$, which is expressible as a decimal but the intermediate value $1/3$ is not.

3. Error handling

One of our motivations was to write code which, when used incorrectly, would produce descriptive errors from which a person with little knowledge of the implementation could deduce what they did wrong. Unfortunately, in many cases the error message produced would not be very helpful.

For example, the following

```
class (Frame a, Frame b, BaseUnit a :*: BaseUnit b ≡ Unitless
      ) ⇒ AssertDualFrames a b | a → b, b → a

instance Frame (F "BadFrame0") where
  type BaseUnit (F "BadFrame0") = Unitless

instance Frame (F "BadFrame1") where
  type BaseUnit (F "BadFrame1") = Meter

instance AssertDualFrames (F "BadFrame1") (F "BadFrame0") where
```

would produce

```
... Could not match type 1 with 0 ...
```

It is not clear where these types appear or what the problem is conceptually.

We addressed this by creating a system which forces the compiler to print better error messages. First we define some primitive types and classes:

```
type family If (a :: Bool) (b :: k) (c :: k) :: k where
  If True  a b = a
  If False a b = b
```

`Always` is the constraint which is always satisfied:

```
class Always
instance Always
```

`Never` is the constraint which is never satisfied:

```
data T

type family Never' t where
  Never' T = Always

class Never
instance Never' () ⇒ Never
```

No other instance can ever be declared for never and the constraint `Never' ()` will not be satisfied.

However, seeing `Never' ()` will likely not be very helpful either, so we declare similar type families for each individual case. The name of the type family reflects the error made:

```
type family AtomicSubframe (x :: k) (xs :: k0) where
  AtomicSubframe T T = Never

type family DualUnits (a :: k0) (b :: k1) where
  DualUnits T T = Never

type family Less_Than (a :: k0) (b :: k1) where
   Less_Than T T = Never

type family Greater_Than (a :: k0) (b :: k1) where
   Greater_Than T T = Never
```

The data type `T` is not exported, so a user can never pass it to any of the above type functions and the constraint can never be satisfied. This is necessary because closed type families must have at least one clause.

The type `Assert` is a useful synonym: if the condition is true, `Assert` is satisfied. Otherwise it is equal to `err`, a constraint which is never satisfied.

```
type Assert bool err = If bool Always err
```

Now we redefine our original example:

```
type AssertDualUnits a b = Assert (a :*: b ≡? Unitless) (DualUnits a b)

class (Frame a, Frame b, AssertDualUnits (BaseUnit a) (BaseUnit b)
      ) ⇒ AssertDualFrames a b | a → b, b → a

instance AssertDualFrames (F "BadFrame1") (F "BadFrame0") where
```

which now produces the error:

```
 No instance for (DualUnits
    (SIUnit ('M 1) ('S 0) ('Kg 0) ('A 0) ('Mol 0) ('K 0))
    (SIUnit ('M 0) ('S 0) ('Kg 0) ('A 0) ('Mol 0) ('K 0)))
```

The type error now indicates what types have caused the error. It also tells the user that conceptually they have declared dual frames which physically should not be dual, since their units are not dual.

4. Conclusion

To be useful, a type system for mathematical models should:

(1) use syntax compatible with the usual notation used by domain experts,
(2) detect significant errors which do occur in practice, and
(3) produce error messages which domain experts will understand.

We think all domain experts would agree that implementing computation involving Fourier Transforms is error prone and some errors can be difficult to detect, but dangerous, producing downstream errors such as to flipped images. Using our encoding of physical properties, all known erroneous applications of the Discrete Fourier Transform are detected at compile time and, moreover, many can be completely eliminated by letting the compiler use type inferencing to derive properties.

It would be impossible to provide a system entirely consistent with the usual notation of experts in this area because applied mathematics does not have consistent notation. Using an embedded language–even embedded in Haskell, which is among the fittest to the purpose–imposes limitations on the syntax, but we think we have made intelligible choices. Also producing intelligible error messages is more challenging; we admit that some error messages will require additional explanation for domain experts but we feel that the strengths of the system outweigh this shortcoming.

If we could ask for further support to increase the utility of embedded DSL offering strong typing, we would ask for additional arithmetical support; division being helpful in our case. Also, an easier way to intercept types and type errors to provide hints at their interpretation, or even suggestions for fixing errors, would make it practical to provide increasing levels of type safety in embedded DSLs.

Given the tools we do have, there remains a lot of work to complete our vision of a system of types capable of capturing modelling errors in inverse imaging problems, even just MRI problems. Some work is routine, but much of it requires the construction of routine concepts from first principles in order to formalize them properly. When we started this work, in addition to the obvious problems with FTs, we tried to derive common regularizers from first principles in order to formalize them in our type system. While we could interpret some regularizers in terms of probability Moghadas [2012], something we need to understand better before encoding in our type system, most are still a mystery and whether errors in modelling and implementation can be avoided by type checking is an open problem. Given the importance of regularization in solving challenging inverse problems, it is an important one.

In conclusion, we are encouraged by our initial success to continue the process of formalizing the concepts and methods mathematical modelling. In a next step, we will demonstrate the advantages of using the types we have constructed to produce sound specifications for a symbolic code generation layer, which will lead to medical imaging software with provable correctness properties.

References

Ana Bove, Peter Dybjer, and Ulf Norell. A Brief Overview of Agda — A Functional Language with Dependent Types. In *Proceedings of the 22nd International Conference on Theorem Proving in Higher Order Logics*, TPHOLs '09, pages 73–78, Berlin, Heidelberg, 2009. Springer-Verlag. ISBN 978-3-642-03358-2. doi: 10.1007/978-3-642-03359-9_6. URL http://dx.doi.org/10.1007/978-3-642-03359-9_6.

Paul Hudak, John Hughes, Simon Peyton Jones, and Philip Wadler. A history of Haskell: being lazy with class. In *Proceedings of the third ACM SIGPLAN conference on History of programming languages*, HOPL III, pages 12–1–12–55, New York, NY, USA, 2007. ACM. ISBN 978-1-59593-766-7. doi: 10.1145/1238844.1238856. URL http://doi.acm.org/10.1145/1238844.1238856.

Mark P. Jones. Type classes with functional dependencies. In *Proceedings of the 9th European Symposium on Programming Languages and Systems*, ESOP '00, pages 230–244, London, UK, UK, 2000. Springer-Verlag. ISBN 3-540-67262-1. URL http://dl.acm.org.libaccess.lib.mcmaster.ca/citation.cfm?id=645394.651909.

Gabriele Keller, Manuel M.T. Chakravarty, Roman Leshchinskiy, Ben Lippmeier, and Simon Peyton Jones. Vectorisation avoidance. *SIGPLAN Not.*, 47(12):37–48, September 2012. ISSN 0362-1340. doi: 10.1145/2430532.2364512. URL http://doi.acm.org/10.1145/2430532.2364512.

Oleg Kiselyov. Number-parameterized Types. *The Monad. Reader*, 5, 2005.

Oleg Kiselyov, Simon Peyton Jones, and Chung chieh Shan. Fun with type functions, 2010.

Ben Lippmeier, Manuel Chakravarty, Gabriele Keller, and Simon Peyton Jones. Guiding parallel array fusion with indexed types. *SIGPLAN Not.*, 47(12):25–36, September 2012. ISSN 0362-1340. doi: 10.1145/2430532.2364511. URL http://doi.acm.org/10.1145/2430532.2364511.

Conor McBride. Faking it simulating dependent types in haskell. *J. Funct. Program.*, 12(5):375–392, July 2002. ISSN 0956-7968. doi: 10.1017/S0956796802004355. URL http://dx.doi.org/10.1017/S0956796802004355.

Maryam Moghadas. Type-safety for inverse imaging problems. MSc thesis, McMaster University, 2012.

Chris Okasaki. From fast exponentiation to square matrices: an adventure in types. *SIGPLAN Notices*, 34(9):28–35, September 1999. ISSN 0362-1340. doi: 10.1145/317765.317781. URL http://doi.acm.org/10.1145/317765.317781.

Jessica L. M. Pavlin. Symbolic generation of parallel solvers for unconstrained optimization. MSc thesis, McMaster University, Department of Computing and Software, 2012.

Rayleigh. The principle of similitude. *Nature*, 95:66–68, 1915.

Bernard Sklar. *Digital communications: fundamentals and applications*. Prentice-Hall, Inc., Upper Saddle River, NJ, USA, 1988. ISBN 0-13-211939-0.

Bo Joel Svensson and Mary Sheeran. Parallel programming in Haskell almost for free: an embedding of Intel's array building blocks. In *Proceedings of the 1st ACM SIGPLAN workshop on Functional high-performance computing*, FHPC '12, pages 3–14, New York, NY, USA, 2012. ACM. ISBN 978-1-4503-1577-7. doi: 10.1145/2364474.2364477. URL http://doi.acm.org/10.1145/2364474.2364477.

Brent A. Yorgey, Stephanie Weirich, Julien Cretin, Simon Peyton Jones, Dimitrios Vytiniotis, and José Pedro Magalhães. Giving haskell a promotion. In *Proceedings of the 8th ACM SIGPLAN Workshop on Types in Language Design and Implementation*, TLDI '12, pages 53–66, New York, NY, USA, 2012. ACM. ISBN 978-1-4503-1120-5. doi: 10.1145/2103786.2103795. URL http://doi.acm.org/10.1145/2103786.2103795.

Appendix A. Haskell Extensions

We have tried to minimize the use of Haskell extensions, partly for portability, but mostly to reduce the learning requirement for a domain expert. Initially, we only used functional dependencies; later found it impractical to do testing of type-level programming without TemplateHaskell, which we also use to improve the input of type-level numbers. In reimplementing our system to take advantage of built-in type-level naturals, we found it much easier to use type families rather than functional dependencies.

In this appendix we describe these features so someone with a basic knowledge of Haskell will be able to follow our code, especially in cases where the use of templates makes numerical type input look different from numerical output from the compiler.

A.1. **Template Haskell.**

In order to make entering type-level numbers more natural, we use two extensions, called TemplateHaskell and QuasiQuotes. The first allows the execution of Haskell code at compile time in order to generate Haskell code included in the module. For example, in order to create a number:

```
> :t $(mkFloatExp "3.14")
$(mkFloatExp "3.14") :: FLOAT Pos 314 (E_ 2)
```

QuasiQuotes enables a new type of syntax called quasi-quotation. With this syntax, one can write:

```
> :t [float| 3.14 |]
[float| 3.14 |] :: FLOAT Pos 314 (E_ 2)
```

We also define the Template Haskell functions randomNat, randomInt, and randomFloat, which generate random numbers at compile time. We use these functions later for generating test cases.

A.2. **Kinds.**

Just as types provide some assurance of correctness for data-level computations, kinds provide some assurance of correctness for type-level computations. There are several Haskell extensions which give the ability to work with kinds. There are several built in kinds:

$*$	All types which are inhabited (i.e., have values) have this kind, read OpenKind.
$* \to *$	Functions have this kind.
Nat	Built in type-level naturals have kind Nat.
Symbol	Type-level symbols, used as labels, have kind Symbol
Constraint	Class constraints have kind Constraint.
[k]	The kind of lists of types is [k].

The extension DataKinds gives the programmer the ability to define their own kinds. It does not introduce new syntax, but rather reuses the syntax of datatype declarations. When DataKinds is enabled, a declaration of the form:

```
data Bool = True | False
```

implicitly becomes a declaration of the form:

```
kind Bool = 'True | 'False
```

The types **'True** and **'False** have kind **Bool**.

The leading quote is used to distinguish the *kind* A from the *type* A in situations where it would be ambiguous. If there is no ambiguity, it can be omitted.

Kinds can be polymorphic in the same way as types. PolyKinds and KindSignatures enable polymorphic kinds. Kind signatures can then be specified using the same syntax as type signatures:

```
data INT (s :: Sign) (n :: Nat) = INT
```

Type-level symbols provide a convenient way of declaring new types for use as labels without having to declare data types and list kinds are parametrized over kind, in analogy with data level lists.

A.3. **Functional Dependencies.** The Haskell 2010 standard allows for type classes with exactly one type parameter. The MultiParamTypeClasses extension relaxes this requirement, allowing for any number of parameters. When combined with FunctionalDependencies, classes of the form:

```
class AssertDualFrames a b | a → b, b → a
```

can now be written. The functional dependency a → b indicates that the type b can be uniquely determined from the type a. In other words, when attempting to satisfy the constraint AssertDualFrames X a, where X is a type and a a type variable, the compiler will know that there is only one valid type for a and it will know what that type is.

If the compiler determines that an instance declaration does not satisfy a functional dependency it will reject the instance.

A.4. **Type Families.** Type families are the main workhorse of most of our code. They implement almost all of the computation done with types. Type families come in two flavours: open and closed. Open type families behave much like classes in that different type instances cannot be overlapping. While open type families are sometimes useful, multi-parameter type classes with functional dependencies are often more powerful.

Closed type families, on the other hand, provide a very powerful feature: pattern matching on types. This means that different patterns in closed families *may* be overlapping and they will be matched in sequence, much like pattern matching on data:

```
type family Equals a b where
  Equals a a = True
  Equals a b = False
```

Unlike pattern matching on data, two variables can have the same name and such a pattern will only match when both variables are known to be the same type.

Appendix B. Implementation of the FT without type-level naturals

We include, in the web-accessible material, implementations of the typed FT developed before built-in type-level naturals were available. While the advantages of built-in naturals are overwhelming, several aspects of the original implementation will be of interest to some readers. To facilitate comparisons, concepts are presented in the same order as in the body of the paper, although many elements are omitted in this appendix and only available in the web-accessible material; more details of the fixed-point implementations can be found in Moghadas [2012]. There are implementations with fixed-point and rational numbers, and arbitrary-precision floating-point numbers.

B.1. **Discretizations and FTs.** Discretizations change only in the way in which numbers are recorded. Numbers are necessarily encoded as compositions of primitive types. For the implementation with fixed-point numbers, fractional numbers are represented via fractions rather than decimal fractions–binary fractions were not considered for readability reasons.

```
meas1 :: Discretization1D     LabFrame
                              (NumSamples (SIZE3 D0 D0 D4))
                              (StepSizeNum(SIZE3 D0 D0 D3))
                              (StepSizeDenom(SIZE3 D5 D0 D0))
                              (U.Meter ())
                              [Complex Double]
```

In some cases, fractions such as 1/3 may be more convenient, and lead to more readable code, and type inferencing across the FT will find representable sampling patterns in more cases, such as:

```
> :type (ft meas1)
(ft meas1)
  :: Discretization1D
       LabFrameT
       (NumSamples (SIZE3 D0 D0 D4))
       (StepSizeNum (SIZE3 D5 D0 D0))
       (StepSizeDenom (SIZE3 D0 D1 D2))
       (U.Meter ())
       [Complex Double]
```

As a side effect of the overall implementation, some type errors are more specific. For example, in the following case the type error flags the incomparability of the frames rather than flagging the higher-level discretizations:

```
> :type (ft meas1) U.+ meas1
> Couldn't match type ‘LabFrameT’ with ‘LabFrame’
    When using functional dependencies to combine
      AssertDualFrames LabFrame LabFrameT,
        arising from the dependency ‘a → b’
        in the instance declaration at Practice.lhs:96:10
      AssertDualFrames LabFrame LabFrame,
        arising from a use of ‘ft’ at <interactive>:1:1−2
    In the first argument of ‘(U.+)’, namely ‘ft meas1’
    In the expression: ft meas1 U.+ meas1
```

Note that ghc also flags errors associated with sampling sizes, but the important error above appears first.

On the other hand, when sizes do not match the error is flagged at too fine a scale, identifying the individual digit which doesn't match, as below where two discretizations:

```
canalSample1 :: Discretization1D  CanalFrame
                                  (NumSamples (SIZE3 D0 D1 D2))
                                  (StepSizeNum (SIZE3 D0 D0 D1))
                                  (StepSizeDenom (SIZE3 D1 D0 D0))
                                  (U.Meter ())
                                  [Double]

canalSample1 =
  Discretization1D [1,1.2,1.3,1.12,1.23,1.12,1.15,1.25,1.18,1.20,1.24,1.28]


canalSample2 :: Discretization1D  CanalFrame
                                  (NumSamples (SIZE3 D0 D1 D2))
                                  (StepSizeNum (SIZE3 D0 D0 D1))
                                  (StepSizeDenom (SIZE3 D2 D0 D0))
```

```
                                        (U.Meter ())
                                        [Double]

canalSample2 =
  Discretization1D [1,1.21,1.2,1.42,1.3,1.32,1.12,1.25,1.23,1.20,1.12,1.28]
```

are summed:

```
> canalSample1 U.+ canalSample2
```

causing the error:

```
    Couldn't match expected type ‘D1’ with actual type ‘D2’
    Expected type: Discretization1D
                     CanalFrame
                     (NumSamples (SIZE3 D0 D1 D2))
                     (StepSizeNum (SIZE3 D0 D0 D1))
                     (StepSizeDenom (SIZE3 D1 D0 D0))
                     (U.Meter ())
                     [Double]
      Actual type: Discretization1D
                     CanalFrame
                     (NumSamples (SIZE3 D0 D1 D2))
                     (StepSizeNum (SIZE3 D0 D0 D1))
                     (StepSizeDenom (SIZE3 D2 D0 D0))
                     (U.Meter ())
                     [Double]
    In the second argument of ‘(U.+)’, namely ‘canalSample2’
    In the expression: canalSample1 U.+ canalSample2
```

The full error message is still quite readable and is unlikely to cause confusion.

The FT is defined in this system as

```
class FT a b | a → b, b → a where
  ft :: a → b
  invFt :: b → a

instance ( Size numSamp, U.Unit rangeU, AssertDualFrames frame1 frame2
         , MultDNonZero (stepSizeNum1, stepSizeDenom1)
                        (numSamp,SIZE3 D0 D0 D1)
                        (stepSizeDenom2, stepSizeNum2))
    ⇒ FT (Discretization1D frame1
                             (NumSamples numSamp)
                             (StepSizeNum stepSizeNum1)
                             (StepSizeDenom stepSizeDenom1)
                             rangeU [Complex Double])
          (Discretization1D frame2
                            (NumSamples numSamp)
                            (StepSizeNum stepSizeNum2)
                            (StepSizeDenom stepSizeDenom2)
                            rangeU [Complex Double]) where
  ft (Discretization1D x) = (Discretization1D $ fft x)
  invFt (Discretization1D x) = (Discretization1D $ ifft x)
```

$$\frac{\text{stepSizeNum1}}{\text{stepSizeDenom1}} \times \frac{\text{numSamp}}{\text{SIZE3 D0 D0 D1}} = \frac{\text{stepSizeDenom2}}{\text{stepSizeNum2}} \tag{10}$$

Which is the same as:

$$\text{stepSize1} \times \text{stepSize2} \times \text{numSamp} = 1 \tag{11}$$

This corresponds to equation (9).

B.2. **Detailed Type-Level Implementation without Naturals.** In this section we explain some interesting implementation details of our type-system. First of all, we present the type-level numbers encoded into our type-system. There are several encodings of the type-level number including Peano numbers, binary encoding, and so on. We used a phantom type representation of a sequence of decimal digits because decimal encoding makes error messages more comprehensible. Since we are using the decimal notation, we need the types for all ten digits:

```
data D0
data D1
data D2
...
data D9
```

There is a class Digit of which all 10 type-level digits are instances. It has a method to convert a type-level single digit to its corresponding data level number.

```
class Digit a where
  digit :: a → Int
```

We defined a phantom data type, SIZE, to implement a 10-digit (fixed precision) type-level number. We also implemented a smaller version, SIZE3, which is a 3-digit type level number to make the examples shorter. We made our 10-digit and 3-digits numbers an instance of class Size, which has a method toInt to convert a type-level number into the corresponding data level (i.e. run-time) number.

```
data SIZE d9 d8 d7 d6 d5 d4 d3 d2 d1 d0

data SIZE3 d2 d1 d0

class Size a where
  toInt :: a → Int
```

For 10-digit numbers we used the same method to make it an instance of the class Size.

```
instance forall d2 d1 d0 . (Digit d2, Digit d1, Digit d0)
         ⇒ Size (SIZE3 d2 d1 d0 ) where
  toInt _ = digit d0 + 10 *(digit d1) + 100 *(digit d2)
    where
      d0 = undefined :: d0
      d1 = undefined :: d1
      d2 = undefined :: d2
```

In the instance definition, d0, d1, and d3 are both types and values. The values are needed by the toInt function, although the value is not inspected, which is why it is acceptable to define them as **undefined**.

Then we implemented the required arithmetic operations on our type-level number. There is a class called Times for multiplying two single digits with each other.

```
class Times da db high low | da db → high , da db → low where
```

Then we specified all of its instances w.r.t different combinations of any two digits. For example:

```
instance Times D0 D0 D0 D0 where
⋮
instance Times D1 D0 D0 D0 where
instance Times D1 D1 D0 D1 where
⋮
instance Times D9 D1 D0 D9 where
⋮
```

```
instance Times D9 D9 D8 D1 where
```

For multiplying two type-level numbers, e.g., two numbers of type `Size`, we needed to implement the type-level addition of 3 to 20 digits. Here, we present such additional classes for adding 3 digit numbers. Other classes have the similar implementations:

```
class Add3 a1 a2 a3 sh sl | a1 a2 a3 → sh, a1 a2 a3 → sl where

instance (Add2 a1 a2 a1a2h a1a2l, Add2 a1a2l a3 a1a2la3h a1a2a3l,
          Add2 a1a2h a1a2la3h D0 a1a2a3h) ⇒ Add3 a1 a2 a3 a1a2a3h a1a2a3l
          where
```

We used multi-parameter type classes together with functional dependencies so that by knowing all input digits (`a1 a2 a3 a4`) we can infer the type of both the low and the high digits of the result.

To present the implementation of a class for multiplying two type-level numbers (of type SIZE), we present the smaller version that is responsible for multiplying two SIZE3 numbers, which captures all of the important ideas. We start by sketching the multiplication of 3 digits by 3 digits to show the required constraints for their multiplication class.

			f_2	f_1	f_0
			e_2	e_1	e_0
–	–	–	–	$[e_0 * f_0]_h$	$[e_0 * f_0]_l$
–	–	–	$[e_0 * f_1]_h$	$[e_0 * f_1]_l$	–
–	–	$[e_0 * f_2]_h$	$[e_0 * f_2]_l$		–
–	–	–	$[e_1 * f_0]_h$	$[e_1 * f_0]_l$	–
–	–	$[e_1 * f_1]_h$	$[e_1 * f_1]_l$	–	–
–	$[e_1 * f_2]_h$	$[e_1 * f_2]_l$	–	–	–
–	–	$[e_2 * f_0]_h$	$[e_2 * f_0]_l$	–	–
–	$[e_2 * f_1]_h$	$[e_2 * f_1]_l$	–	–	–
$[e_2 * f_2]_h$	$[e_2 * f_2]_l$	–	–	–	–
			g_2	g_1	g_0

For the first digit of result 'g_0', there is just one term to be counted, which is the low digit of '$e_0 \times f_0$', and this can be implemented using the `Times` class. The second digit of result 'g_1' is obtained by adding the high digit of '$e_0 \times f_0$', the low digit of '$e_0 \times f_1$', and the low digit of '$e_1 \times f_0$' which can be implemented using the 'Add3' class. The third digit is the result of adding five different known terms which is implemented using `Add5` class. Continuing in this way, we arrive at the definition:

```
class MultD3  f2 f1 f0  e2 e1 e0  g2 g1 g0 |
              f2 f1 f0 e2 e1 e0 → g2,
              f2 f1 f0 e2 e1 e0 → g1,
              f2 f1 f0 e2 e1 e0 → g0 where
instance (  Times f0 e0 p00h p00l, Times f1 e0 p10h p10l,
            Times f2 e0 D0 p20l, Times f0 e1 p01h p01l,
            Times f1 e1 D0 p11l, Times f2 e1 D0 D0,
            Times f0 e2 D0 p02l, Times f1 e2 D0 D0,
            Times f2 e2 D0 D0,
            Add3 p00h p10l p01l c1h c1l,
            Add5 p20l p01h p11l p02l c1h D0 c2l)
        ⇒ MultD3 f2 f1 f0 e2 e1 e0 c2l c1l p00l where
```

We implemented arithmetic for three- and ten-digit numbers in this way and defined a `MultD` class such that each size-specific multiply would be an instance of this class.

In addition, we defined an special case:

```
class MultDNonZero f e g | f e → g, f g → e, e g → f where
instance  (NonZero f,  NonZero e, NonZero g, MultD e f g)  ⇒
                       MultDNonZero e f g where
```

to allow type inferencing to work, for example, across FTs.

Type-level numbers are needed for both sizes and dimensions, but dimensions are never big numbers so it makes sense to create small numbers using phantom data types to represent different powers for each basic unit. We created a data type for each required basic unit w.r.t the different powers (we do not need all basic units for medical imaging purpose). For example:

```
data M0   -- means the dimension length to the power of 0 (dimensionless)
data M1   -- means the dimension length to the power of 1
data M2
...
data M_1  -- means the dimension length to the power of −1
data M_2
```

For every basic unit, we needed a class with a method to convert type-level dimensional numbers–from -5 to 5 is enough for our application–into its corresponding data level number. For example, for length we have a class called `UnitM`:

```
class UnitM a where
  toIntM :: a → Int
instance UnitM M0 where
  toIntM _ = 0
instance UnitM M1 where
  toIntM _ = 1
...
instance UnitM M_5 where
  toIntM _ = −5
```

There is also another class for multiplying two powers of the same basic dimensions, which is responsible for simplifying the resulting dimensions by adding their powers.

```
class AddDim a b c | a b → c, a c → b, b c → a where
```

We added all of its reasonable instances to our type-system. For example:

```
...
instance AddDim M_4 M3 M_1 where
...
```

The above instance means that $m^{-4} \times m^3$ is equal to m^{-1}.

These powers of fundamental units are combined into an SI unit:

```
data SIUnit u1 u2 u3 u4 u5 val = SIUnit val
```

Type synonyms are defined for some composite units:

```
type Unitless val  = SIUnit M0 Kg0 S0 A0 Mol0 val
type Meter val     = SIUnit M1 Kg0 S0 A0 Mol0 val
type PerM val      = SIUnit M_1 Kg0 S0 A0 Mol0 val
type MPerS val     = SIUnit M1 Kg0 S_1 A0 Mol0 val
...
```

We implemented a class for all possible units, `Unit`, which accepts only composite units that have the basic units in a specific order.

```
class Unit a where

instance  (UnitM m, Show m, UnitKg kg, Show kg, UnitS s, Show s,
           UnitA a, Show a, UnitMol mol, Show mol, Show val)
           ⇒ Unit (SIUnit m kg s a mol val) where
```

We also needed to implement the required arithmetic operation with respect to the physical quantities. Such operations are more restricted than their default implementation in the **Prelude**. For example, adding or subtracting two physical quantities only makes sense when both have the same dimension. We implemented the Mult class with a '(∗)' method and the Add class with '(+)', **negate**, and '(−)' methods:

```
class Mult a b c | a b → c, a c → b, b c → a where
  (∗) :: a → b → c

class Add a where
  (+) :: a → a → a
  negate :: a → a
  (−) :: a → a → a
```

Then we made the SIUnit data an instance of those classes:

```
instance  (UnitM u1, UnitKg u2, UnitS u3, UnitA u4, UnitMol u5, Add val)
          ⇒ Add (SIUnit u1 u2 u3 u4 u5 val) where
  (SIUnit x) + (SIUnit y) = SIUnit (x + y)
  negate (SIUnit x) = SIUnit (negate x)
  (SIUnit x) − (SIUnit y) = SIUnit (x − y)
```

Which states that addition and subtraction are only valid when both quantities have the same dimension.

```
instance  (AddDim u1 v1 w1, AddDim u2 v2 w2, AddDim u3 v3 w3,
           AddDim u4 v4 w4, AddDim u5 v5 w5, Mult val val val)
    ⇒ Mult  (SIUnit u1 u2 u3 u4 u5 val)
            (SIUnit v1 v2 v3 v4 v5 val)
            (SIUnit w1 w2 w3 w4 w5 val) where
  (SIUnit a) ∗ (SIUnit b) = SIUnit (a ∗ b)
```

For multiplication, the power of each basic unit in the first argument is added to the power of the same basic unit in the second argument to determine the physical dimension of the result. The functional dependency in all directions makes it possible to infer the physical unit of any argument by knowing the two others. At this state, we can show an example using the physical dimensions in our type-system. Suppose we have three physical quantities representing distance, time and velocity defined as:

```
distance = SIUnit 2 ::Meter Double
time = SIUnit 10 :: Second Double
velocity = SIUnit 0.2 :: MPerS Double
```

If we multiply velocity and time, the resulting quantity correctly has the dimension distance:

```
>  velocity ∗ time
2.0 m
```

but if we try to add time with distance it will cause a compile time error:

```
> time + distance
  Couldn't match expected type 'M0' with actual type 'M1'
    Expected type: Second Double
      Actual type: Meter Double
    In the second argument of '(+)', namely 'distance'
    In the expression: time + distance
```

In the error message, it explicitly mentions that the expected type is Second **Double** and the actual type is Meter **Double** which is easy for the user to understand.

In addition to fixed-point and rational numbers, we also include an implementation of arbitrary-precision floating point numbers and systems of units built on top of them. As with the implementation in the body of this paper, the floating point implementation has many advantages over a fixed-point system, but it does have the disadvantage of not being closed under division.

Appendix C. Testing type-level numbers

The use of type-level numbers in order to ensure correctness in computation is only useful if the arithmetic of the type-level numbers themselves is correct. Here we describe tests conducted in order to assure the correctness of the type-level numbers.

C.1. **Fixed-width numbers.** Two standard lengths of fixed-width numbers are used: width-3 and width-10. Since there are a finite number of representable numbers in these systems, addition and multiplication can produce only a finite number of numbers. It is then possible to test *all* multiplications and additions. For width-3 this may be feasible; there are 1000 different possibilities for both the summands/multiplicands, so a mere million tests are required for each operation (further limited by overflow). However, due to the slow speed of the compiler when type checking arithmetic, even these million cases are too many to test individually. For width-10, this number is is 10^{20}; it is impossible to verify so many cases individually. Therefore, we decide to check the validity of multiplication on a series of random numbers.

The main function used is defined as follows:

```
testAsync nCases nProcs = do
    putStrLn "Writing source files..."
    writeRand nCases nProcs
    putStrLn "Compiling..."
    h ← mapM compileFile [0..(nProcs − 1)]
    mapM_ waitForProcess h
    mapM_
      (λx → readProcess ("test\\testcase" ++ alpha x ++ ".exe") [] []>>=
               putStrLn . (++) ("λ‘nRunning test case " ++ alpha x ++ "λ‘n")
             ) [0..(nProcs − 1)]
```

It will produce nProcs source files, each containing nCases random test cases. It will then call compileFile on each source file which will compile the source files concurrently. After compiling is finished, each test file will be called.
The function generates files which contain:

```
{-# LANGUAGE ScopedTypeVariables #-}
module Main where
import SizeTypes
f :: forall e f g . (MultDNonZero e f g, NonZero f, NonZero e, NonZero g,
    MultD e f g, Size e, Size f, Size g) ⇒ (e, f, Integer) → IO Bool
f (a, b, s)
    | s ≡ g = putStr "" >> return True
    | otherwise = putStrLn ("Mult of " ++ show (toInt a) ++ " and "
                            ++ show (toInt b) ++ " failed")
                    >> return False
        where g = toInt (u :: g)
u = undefined
```

f is a function which takes two type-level numbers and the data-level representation of the result of their multiplication. It then multiplies them on the type level and

compares this to the multiplication computed on the data level.
Each module contains a `main` function which contains the randomly generated numbers. The following was produced with `testAsync 10 2` :

```
main =
    putStrLn "Testing type-level multiplication of 10 pairs of numbers."
    >> ((λb → if and b then putStr "All mults passed" else putStr "")
    =<< sequence
        [
        f (u :: SIZE D0 D0 D0 D0 D0 D0 D6 D3 D7 D7 ,
           u :: SIZE D0 D0 D0 D0 D2 D0 D0 D6 D9 D3 ,
           1279819261),
        ...
        f (u :: SIZE D0 D0 D0 D0 D0 D1 D6 D7 D6 D1 ,
           u :: SIZE D0 D0 D0 D0 D0 D4 D0 D6 D0 D0 ,
           680496600)
        ])
```

The `main` function will call `f` on every set of numbers and verify that every check returns **True**.

As an example:

```
>testAsync 10 2
Writing source files...
Compiling...

...

Running test case A
Testing type-level multiplication of 10 pairs of numbers.g
All mults passed

Running test case B
Testing type-level multiplication of 10 pairs of numbers.
All mults passed
```

C.2. **Variable-width numbers.** While the number of total test cases for fixed width numbers is very large, it is finite. For variable width numbers, there are infinite test cases. Despite this, the implementation of arithmetic on variable width numbers facilitates testing in several ways. Since arithmetic is recursive, it is sufficient to test arithmetic on a set of number pairs of a certain width; if these tests pass, then arithmetic with longer numbers should pass as well. Also, arithmetic on floating point numbers is defined by arithmetic on naturals, so it is sufficient to verify the correctness of natural numbers, and, knowing the correctness of the simple mathematic rules which govern the arithmetic of floating point numbers, we can infer the correctness of floating point numbers as well.
The testing of variable width numbers involves the use of Template Haskell to generate test code. Testing of four operations is supported:

```
data Op = Sum | Mult | Sub | Comp deriving (Enum, Show)
```

Then we have the following function, which takes two numbers and an operation, and outputs `Q Exp` which represents the abstract syntax tree of the Haskell code being generated. This data type will later be converted to actual code and placed in the module where the function is called.

```
mkTestFunc :: Word32 → Word32 → Op → Q Exp
```

For example, `mkTestFunc 100 750 Mult` generates the following expression:

```
(750 * 100) ≡
  (numToIntegral (tMultD (D7 :$ D5 :$ D0 :$ End) (D1 :$ D0 :$ D0 :$ End)))
```

Another function will take a list of pairs of numbers and call `mkTestFunc` on each pair, returning a list of declarations–assigning arbitrary names to each declaration–as well as another function declaration which contains a list of the generated test functions.

```
mkFuncList2 :: String → [(Word32, Word32)] → Op → Q [Dec]
```

In another module, the test code is invoked by calling `mkFuncListM2`, which is identical to the above function except it allows for its second argument to be of type `IO [(Word32, Word32)]`, which allows it to be a list of randomly generated numbers. Note the `${...`, which is a special syntax of Template Haskell, allowing the `Q Dec` data type to be inserted as code.
For example, a possible invocation of the test code is:

```
$(mkFuncListM2 "test" (randList2' (10^3 + 1, 10^4 − 1)>>=return . take
100) Mult)
```

The output of this call is:

```
f0 = ... :: Bool
f1 = ... :: Bool
...
f99 = ... :: Bool
test = [f1, f2 ... f98, f99] :: [Bool]
```

Then to evaluate the test function:

```
>and test
True
```

Each function beginning with 'f' is a test function of the form described above. This allows calling all of the tests at once or calling any individual test.

C.3. **Variable-width numbers.** While the number of total test cases for fixed width numbers was very large, it was finite. For variable width numbers, there are infinite test cases. Since arithmetic is implemented using the built-in naturals, we can be reasonable confident of its correctness. We are only testing for errors that arise for certain edge cases.

Since arithmetic on naturals requires no extra code, it is guaranteed to be correct if the built in naturals are correct; despite this fact, we implement tests for naturals.

The testing of variable width numbers involves the use of Template Haskell to generate test code. Testing of four operations is supported: addition, multiplication, subtraction, and comparison.

Then we have the following function which abstracts over different ways of creating test cases and outputs `Q Exp`, which represents the abstract syntax tree of the Haskell code being generated. This data type will later be converted to actual code and placed in the module where the function is called.

```
mkTestCase mf mg f g mka mkb =
  fmap return mka>>=λa → fmap return mkb>>=λb →
    [| (show $a, show $b, $mf ($f $a $b), ($g ($mg $a) ($mg $b))) |]
```

The definition is perhaps not very indicative of how the function works, so as a concrete example

```
addTest = mkTestCase [|numToRational|] [|numToRational|] [|add|] [|(+)|]
```

will generate the expression:

```
λx y → (show x, show y,
         numToRational (add x y), numToRational x + numToRational y)
```

Another function takes a method of random number generation, a test function of the above form, and a number of tests to perform, and then returns a function which takes a minimum and maximum bound for the generated numbers and a list of numbers to exclude from the tests, and returns a `ExpQ` representing the generated test cases:

```
randomTests rnd test m a b ex =
  let xs = replicate (ceiling $ sqrt $ fromIntegral m) (rnd a b ex) in
    listE $ take m [ test q r | q ← xs, r ← xs]
```

We specialize this function to the different number types:

```
testRandomNats = randomTests randomNat
testRandomInts = randomTests randomInt
testRandomFloats = randomTests randomFloat
```

In another module, the tests are invoked as follows:

```
main :: IO ()
main = do
  sat "Nat/addition"       $(testRandomNats addTest     200 0 100000 [])
  sat "Nat/multiplication" $(testRandomNats multTest    200 0 100000 [])
  sat "Nat/mult-nz"        $(testRandomNats multNZTest  200 1 100000 [])
  sat "Nat/comparison"     $(testRandomNats compareTest 200 0 100000 [])

  sat "Int/addition" $(testRandomInts addTest     200 (-100000) 100000 [])
  ...

  sat "Float/addition" $(testRandomFloats addTest 50 (-1000) 1000 [])
  ...
```

where `sat` is a function which takes a list of test cases, checks to see if they are satisfied, and pretty prints the result.