Experience Report: Type-checking Polymorphic Units for Astrophysics Research in Haskell

Takayuki Muranushi
The Hakubi Center for Advanced Research
Kyoto University
muranushi.takayuki.3r@kyoto-u.ac.jp

Richard A. Eisenberg
University of Pennsylvania
eir@cis.upenn.edu

Abstract

Many of the bugs in scientific programs have their roots in mistreatment of physical dimensions, via erroneous expressions in the quantity calculus. Now that the type system in the Glasgow Haskell Compiler is rich enough to support type-level integers and other promoted datatypes, we can type-check the quantity calculus in Haskell. In addition to basic dimension-aware arithmetic and unit conversions, our units library features an extensible system of dimensions and units, a notion of dimensions apart from that of units, and unit polymorphism designed to describe the laws of physics. We demonstrate the utility of units by writing an astrophysics research paper. This work is free of unit concerns because every quantity expression in the paper is rigorously type-checked.

1. Introduction

In 1997, when Canada was moving from the Imperial to the metric system, a Boeing 767 ran out of fuel in the midst of a flight and was on the verge of crashing, all because the airplane was given 22,300 pounds of fuel where 22,300 kg was actually needed [15]. In 1999, a NASA spacecraft Mars Climate Orbiter was lost after it entered too low an orbit, because the software involved mistakenly interpreted the thruster force in pounds of force when a measurement in Newtons was assumed [17]. These are just samples of the many incidents that have been caused by a mistake in units and dimensions.* Because of the possibility of such mistakes, scientists routinely check for both dimensions and units as they perform calculations. Kennedy sums it up well in saying that "units-of-measure are to science what types are to programming" [9].

Type systems that enforce the correct handling of dimensions and units have been around for some time [6, 8, 16]. Haskell, a language renowned for its rich type system, naturally has multiple packages that enable type-checking units-of-measure.† However, as we attempt to describe real astrophysical reasoning in Haskell, we became aware of an important concept that no existing library supports — unit polymorphism.

Any meaningful law of physics holds in any system of units. For example, the *mass* of fuel consumed in a flight equals the *length* of the air route divided by the economy of the aircraft (*length* travelled per *volume* of fuel) times *density* (*mass* per *volume*) of the fuel. Slanted words all refer to *dimensions*, and the equation holds no matter whether specific *units* are kilograms or pounds. Such unit-independent laws are common in the quantity calculus, and unit polymorphism naturally arises when we type-check our work. We demonstrate three distinct merits of unit polymorphism (§ 5):

1. We can faithfully express unit-independence of laws of physics.
2. We can write libraries that deal with quantities without forcing users to adopt specific choice of units.
3. We can perform calculations that would have resulted in overflows/underflows, had we used the default choice of units.

This report addresses the implementation of a unit-polymorphic quantity calculus system not by extending Haskell's type system directly, but by defining a library built with the features that Haskell already supports. We describe the package units, and our experiences in using this package to support astrophysics research.

Our contributions are as follows:

- In order to type-check the quantity calculus in Haskell's type system, we summarize the design principles and specifics of the units package, which is available for `cabal install` (§ 3).
- We compare units to unittyped, a similar library that does not support unit polymorphism (§ 4).
- With units, we can now write functions that deal with abstract dimensions in Haskell. We demonstrate the use of units library by writing an astrophysics paper in Haskell (§ 6).

2. Background: terminology

We describe here some terminology taken from the quantity calculus, which we will use throughout this experience report. These definitions are somewhat informal; please see "International Vocabulary of Metrology" ([1], hereafter VIM) for the full details.

Quantity [VIM 1.1] A quantity can be thought of something that can be measured. A quantity has an associated dimension and is expressed in terms of a magnitude and a unit. For example, *diameter* and *wavelength* are both quantities of the dimension *length*. Examples of quantity values are 3.2 kg and $9.8\ \mathrm{m/s^2}$.

Dimension [VIM 1.7] A dimension is a name given to a group of quantities that differ only by numerical factors. Examples

---

* See, `http://lamar.colostate.edu/~hillger/unit-mixups.html`, from where the anecdotes above are adapted.

† dimensional, dimensional-tf and unittyped among others. For a list of these, see `http://www.haskell.org/haskellwiki/Physical_units`

include *length*, *mass*, or *velocity*. Dimensions can be composed of other dimensions; a *velocity* is a *length* over a *time*.

Unit [VIM 1.9] A unit is a particular amount of a quantity, with which all other quantities of the same dimension can be measured. Examples of units include meters, feet, and kilograms. Note that a unit always measures a unique dimension, but that a dimension may have several different units available.

Quantities of the same units can be added. Any two quantities can be multiplied, with their units and dimensions changing accordingly. For example, $50\,\mathrm{mph} = 100\,\mathrm{mile}/2\,\mathrm{hr}$, at dimension $\mathit{velocity} = \mathit{length}^1 \times \mathit{time}^{-1}$.

Coherent System of Units (CSU) [VIM 1.14] A coherent system of units consists of a chosen set of base units and base dimensions. Other units are derived by taking the product of powers of base units. *Coherent* units are direct products of powers of base units, while *off-system* units require conversion factors other than 1.

For example, the International System of Units (SI) is a CSU that consists of seven base units and dimensions. A Joule (=$1\,\mathrm{kg} \cdot \mathrm{m}^2/\mathrm{s}^2$) is the coherent derived unit of *energy* in SI, while a liter (=$10^{-3}\,\mathrm{m}^3$) is an off-system unit of *volume* with respect to SI.

Numerical Value [VIM 1.20] The numerical value of a quantity is the number in the quantity represented in a certain unit. For example, the numerical value of $9.8\ \mathrm{m}/\mathrm{s}^2$ is 9.8.

3. The units package

3.1 Design goals

What would distinguish units from other quantity calculus packages? What design principles were we aiming for? These ideals are presented roughly in order of importance.

Strong type-checking: Under no circumstances should a quantity representing a length be allowed to be added to a quantity representing a time, for example. Multiplication of these quantities is allowed, of course, yielding a multiplicative dimension of *length* × *time*.

Extensibility: A key attribute of units is that it is fully extensible. The core system defines only one dimension, *Dimensionless*, with its unit *Number*. All other units are defined purely using the interface exported by units. This makes units easily applicable to domains beyond physics, such as finance and management (Imagine *ManMonth*, a unit of dimension *Labor*.)

Simple user-visible types: After a small amount of work to set up the dimensions and the units, users should be able to write simple type signatures on their code. Additionally, declaring new dimensions and units should be formulaic and not require expert knowledge of Haskell's type system.

Unit-polymorphism: Definitions should not be tied to a user's particular choice of units. For example, take the following definition of kinetic energy E_k:

```
kineticEnergy :: Mass → Velocity → Energy
kineticEnergy m v = 0.5 *| m |*| v |*| v
```

The property that $E_\mathrm{k} = \frac{1}{2}mv^2$ remains true whether we're measuring mass in kilograms, velocity in meters per second, and energy in Joules, or not. Thus, we want enough flexibility in our types to be able to give *kineticEnergy* a type more like

```
kineticEnergy :: Mass u → Velocity u → Energy u
```

where the type variable *u* encodes details of the particular units at hand. Here, the *dimensions*, but not the *units* are fixed – *kineticEnergy* is a dimension-monomorphic, unit-polymorphic definition.

Unit conversions: A user should be able to specify known quantities in any convenient choice of units and request a result in any convenient choice of units. How many years does the Sun take to complete its galactic orbit of 27,200 light-years radius, given its speed $220\ \mathrm{km}/\mathrm{s}$? Such a calculation should not require the user to think about unit conversions — the type system should handle it for him.

3.2 Interface

How would one keep track of dimensions and units in a type? Let's focus on dimensions, first. A dimension can be thought of as a set of base dimensions paired with integer exponents. For example, a *velocity* is a $\mathit{length}^1 \times \mathit{time}^{-1}$, and *energy* is $\mathit{mass}^1 \times \mathit{length}^2 \times \mathit{time}^{-2}$. To make these pairs easier to manage, we introduce a *Factor* kind*

```
data kind Factor = F ⋆ Z
```

where *Z* is our kind for type-level integers. Here, and throughout our implementation, we make liberal use of algebraic datakinds, promoted from datatypes, as introduced by Yorgey et al. [20].

Is the kind of a dimension *Dimension*? No — we choose to use ⋆ because of its extensibility and convenience. Users can, in any module, freely declare new datatypes with **data**, and these datatypes serve as dimensions. For example, we might say

```
data Length = Length
```

and use the type *Length* as a dimension in a factor.

For these reasons, the first argument to our quantity type, of kind [*Factor*], represents the set of dimension factors.

But, what about units? Units control exactly what number is internally stored. If we are storing the length of a day in seconds, we would store 86,400. If we were storing in hours, we would store 24. However, we still want to keep the dimensions primary, not the units, in order to allow for unit-polymorphic programming.

The solution: store a finite map from dimensions to units in the type. We call this mapping a local CSU, or LCSU. It is local because, as we will see, it can vary in different places in our program.

Finally, we wish to allow users to choose their own underlying numerical type, so we index by this choice of representation.

We thus have the *Qu* type, defined as follows:

```
newtype Qu (dimFactors :: [Factor])
           (lcsu :: LCSU) (n :: ⋆) = Qu n
```

The *dimFactors* parameter is our type-level list of *Factor*s. The parameter *n* is the underlying numerical representation; functions creating and eliminating *Qu*s require *n* to be in the *Fractional* class. Note that *Qu* is a **newtype** — there is no runtime cost to type-checking your quantities.

An *LCSU* is stored as a type-level association list, mapping dimensions to units:

```
data kind LCSU = MkLCSU [(⋆, ⋆)]
```

Here, we see that units, like dimensions, are denoted with elements of kind ⋆. The situation for units is analogous for that for dimen-

---

* Throughout this report, we imagine that Haskell has a richer syntax for dealing with kinds. Here, we declare a datakind, which is a datatype that can be used only at the kind level. Alas, this is not yet possible — one must declare a data*type* to get a data*kind*. Because ⋆ is only a kind, not a type, using ⋆ as we have done here is impossible. The true definition of *Factor* is **data** *Factor star* = *F star Z*, and whenever we use *Factor*, we must write *Factor* ⋆. These details are distracting, so we ignore them from here on out.

sions — we want the set of units to be modular and extensible; hooking into Haskell's ability to do this with datatype definitions is the easiest way of achieving these goals.

Dimension *and* Unit *classes* When a user wishes to declare a new dimension or unit, she must also make instances of the *Dimension* or *Unit* class, as appropriate. The details of how these work internally are beyond the scope of this report, but they are critically necessary for type-checking and automatic unit conversion. It is worth noting, however, that a *Unit* instance must declare the related dimension and define the relationship to a convertible unit, if any — the rest of the unit conversions are inferred from this one known link.

Unit and dimension combinators The units package defines a small set of combinators used to build compound units and dimensions from simpler ones. These are :∗ for multiplication, :/ for division, :ˆ for exponentiation, and :@ for prefixing. Thus, the following definitions are all quite logical:

```
type Velocity  = Length :/ Time
type Kilogram = Kilo :@ Gram
type Area      = Length :^ Two
    -- Two is our type-level integer 2
```

Setting up the* Qu *type Getting all the details correct for the *Qu* type is challenging. The units package exports a number of type synonyms to make this easier. Here is a more complete example of how it all fits together:

```
data LengthDim = LengthDim
instance Dimension LengthDim
data TimeDim = TimeDim
instance Dimension TimeDim
data Meter = Meter
instance Unit Meter where
    -- there is no known convertible unit
  type BaseUnit   Meter = Canonical
  type DimOfUnit Meter = LengthDim
data Second = Second
instance Unit Second where
  type BaseUnit   Second = Canonical
  type DimOfUnit Second = TimeDim
type MyLCSU = MkLCSU '[ '(LengthDim, Meter),
                        '(TimeDim, Second)]
type Length   = MkQu_DLN LengthDim MyLCSU Double
type Time     = MkQu_DLN TimeDim MyLCSU Double
type Velocity = MkQu_DLN (LengthDim :/ TimeDim)
                         MyLCSU Double
distance :: Velocity → Time → Length
distance v t = v |*| t
```

The *MkQu_DLN* type synonym creates an appropriately instantiated *Qu* type, suitable for use in the type signature for *distance*. The *DLN* suffix means that *MkQu_DLN* expects a dimension, an LCSU, and a numerical type. Other combinations are also exported.

Introducing and eliminating quantities Quantities are made using the (%) operator and eliminated using the (#) operator:

```
(%) :: (ValidDLU dim lcsu unit, Fractional n)
    ⇒ n → unit → Qu dim lcsu n
(#) :: (ValidDLU dim lcsu unit, Fractional n)
    ⇒ Qu dim lcsu n → unit → n
```

Both of these functions take a unit parameter. This is why the unit (and dimension) types are not empty; it is convenient to use the one data constructor to represent the type. Continuing the example above, we could write

```
speedOfLight :: Velocity
speedOfLight = 299792458 % (Meter :/ Second)
```

Quantity combinators All the standard mathematical operations are available on quantities: addition, subtraction, multiplication and division (both among quantities and with scalars), exponentiation, comparisons, and roots. Operator names are decorated by vertical bar(s) on the side(s) quantities are applied.

Because exponentiation and root-taking change the type-level integers, the exponent must be known at compile time. This reflects the fact that only nondimensional quantities can be raised to variable powers. We use singletons (as generated by the singletons package [4]) to represent the exponents.

The types of these combinators ensure the correctness of the operations. For example, the type of |+|, the addition operator, checks to make sure the dimensions of both operands are equivalent.

3.3 Implementation

The gritty details of how the implementation tracks these types and makes all the pieces fit together are beyond the scope of this report. We pause to note a few points that may be of general interest:

Type-level integers We use our own type-level integer kind *Z* extensively. This kind uses a unary (Peano-style) representation with type families implementing standard operations. There is no enhanced reasoning/solving capability with our *Z* kind, but this is not a problem in practice. We chose not to use GHC's built-in type-level naturals for two reasons: 1) we require integers, not natural numbers; and 2) there is not yet enough support for runtime access to the build-in naturals. Specifically, if n_1 and n_2 are known at runtime (that is, we have *KnownNat* n_1 and *KnownNat* n_2), we cannot get $n_1 + n_2$ at runtime (that is, GHC cannot solve for *KnownNat* $(n_1 + n_2)$).

Closed type families Our implementation depends critically on closed type families [5] to be able to consider a type-level list of *Factor*s to be a set. This whole approach would be infeasible without closed type families.

4. The unittyped package

We evaluate the success of units by comparing it with unittyped, a different approach to type-checking units-of-measure in Haskell. Indeed, before discovering units, one of the authors (TM) had contributed to unittyped with physics applications in mind. We give a brief overview of unittyped in this section. The differences from a user's point of view are summarized in Table 1.

unittyped, like units, expresses compound dimensions and units as lists of pairs of key types and type-level integers. The type for quantity in unittyped is:

```
Value (dim :: [(⋆, Z)]) (uni :: [(⋆, Z)]) (n :: ⋆)
```

which resembles units's counterpart:*

```
Qu (dim :: [Factor]) (lcsu :: LCSU) (n :: ⋆)
```

However, the two libraries treat dimension-unit correspondence in different ways. The *Value* type constructor of unittyped takes both the dimension and unit as arguments. A type class *Convertible'* asserts that *uni* is of the dimension *dim*, and then provides the conversion factor for the unit. Every arithmetic operation requires a *Convertible'* constraint.

* Note that *Factor* is equivalent to (⋆, *Z*)

(1) quantity calculus package	unittyped	units
(2) means of type level computation	functional dependencies	closed type families
(3) current status	not buildable since GHC 7.8.1	buildable on GHC 7.8.2
(4) type signature of a quantity	$\mathit{Value}\ (\mathit{dim} :: [(\star, Z)])\ (\mathit{uni} :: [(\star, Z)])\ (n :: \star)$	$\mathit{Qu}\ (\mathit{dim} :: [\mathit{Factor}])\ (\mathit{lcsu} :: \mathit{LCSU})\ (n :: \star)$
(5) convert to quantity	$x = \mathit{mkVal}\ 5.2 :: \mathit{Gram}\ \mathit{Double}$	$x = 5.2\ \%\ \mathit{Gram}$
(6) convert to numerical value	$\mathit{val}\ x$	$x\ \#\ \mathit{Gram}$
(7) extract x (in g) in kg	$\mathit{val}\ (\mathit{autoc}\ x :: \mathit{Value}\ [(\mathit{Mass}, \mathit{POne})]$ $[(\mathit{Kilo}\ \mathit{Gram}, \mathit{POne})]\ \mathit{Double})$	$x\ \#\ \mathit{kilo}\ \mathit{Gram}$
(8) pretty-print x (in g) in kg	$\mathit{pp}\ (\mathit{fmap}\ (0.001*)\ x)$ ++ "kg"	$\mathit{ppln}\ (\mathit{kilo}\ \mathit{Gram})\ x$
(9) define a type synonym for a compound unit (spectral radiance)	**type** $\mathit{SR} = \mathit{Value}$ $[(\mathit{Mass}, \mathit{POne}), (\mathit{Time}, \mathit{NTwo})]$ $[(\mathit{Watt}, \mathit{POne}), (\mathit{Meter}, \mathit{NTwo}),$ $(\mathit{Hertz}, \mathit{NOne})]$	**type** $\mathit{SR} = \mathit{Watt} :/ (\mathit{Meter} :\hat{}\ \mathit{Two}) :/ \mathit{Hertz}$

Table 1: Comparison of features in unittyped and units. Code fragments are abbreviated for the sake of readability.

On the other hand, *Qu* of units takes only the dimensions. The corresponding unit can be computed on demand by referring to the LCSU dictionary. A *Value* can equally represent coherent and off-system units, while *Qu* is designed to express only coherent unit quantities. Off-system units in units appear only at conversion to/from numerical values. Various type constraints are still needed for unit conversions, but none is needed for arithmetic. This is because coherent units can be operated on without conversion.

Another difference is in the method of type-level computation. unittyped uses functional dependencies while units uses closed type families (Table 1, (2)). Critically, the way unittyped uses functional dependencies appears to take advantage of a long-standing GHC bug,* which is fixed in GHC 7.8. (Table 1, (3)) Essentially, the functional dependencies in unittyped overlap in much the same way as the type family equations do in units, as needed to implement type-level sets. However, overlapping equations in a closed type family has a reasonable, well-defined semantics; such overlap over unordered functional dependencies does not. It is an open question whether the unittyped approach can be brought up to date with GHC 7.8. The authors conjecture that this is indeed not possible.

The discontinuity of the build is one reason why TM stopped using unittyped. The other is the polymorphism problem that was discovered while using unittyped, and that motivated units' design.

5. Unit Polymorphism

5.1 Writing Unit-Independent Quantity Expressions

So, how many pounds of gasoline did the pilot actually need? Given the travel distance *dist* (in miles), the fuel economy of the aircraft *eco* (miles per gallon) and gasoline density *gasden* (pounds per gallon), it is easy to write the function in unittyped:

```
gasMass dist eco gasden = dist |/| eco |*| gasden
```

Then, can we make the above functions accept arbitrary set of units, to prepare for the future metric system change? A natural starting point is to put type variables in places of unit types.

```
gasMass :: Value '[ '(Length, POne)] len n
    → Value '[ '(Length, NTwo)] fe n
    → Value '[ '(Mass, POne), '(Length, NThree)] den n
    → Value '[ '(Mass, POne)] mass n
gasMass dist eco gasden = dist |/| eco |*| gasden
```

Alas, it doesn't compile! The correct program is as follows.

* Fixing the bug (https://ghc.haskell.org/trac/ghc/ticket/2247) is by implementing the liberal coverage condition [18].

```
gasMass :: (Fractional n
  , Convertible' '[ '(Length, POne)] len
  , Convertible' '[ '(Length, NTwo)] fe
  , Convertible' '[ '(Length, PThree)] vol
  , Convertible' '[ '(Mass, POne), '(Length, NThree)] den
  , Convertible' '[ '(Mass, POne)] mass
  , MapNeg fe nfe            -- nfe = 1 / fe
  , MapMerge len nfe vol     -- vol = len * nfe
  , MapMerge vol den mass    -- mass = vol * den
  ) ⇒ Value '[ '(Length, POne)] len n
    → Value '[ '(Length, NTwo)] fe n
    → Value '[ '(Mass, POne), '(Length, NThree)] den n
    → Value '[ '(Mass, POne)] mass n
gasMass dist eco gasden = dist |/| eco |*| gasden
```

We need 9 lines of type constraints and 4 lines of types for just one line of value-level computation. This result does not argue in favor of strong typing.

In the unit-monomorphic case, unittyped is a pleasure to use, because it allows GHC to infer the dimensions and units via functional dependencies for even compound computations. In the unit-polymorphic case, however, the design causes trouble. The result unit type *mass* is not just universally quantified, but actually depends on *len*, *fe*, and *den* in a complicated way — hence all the constraints above. The number of constraints required are at least twice the number of arithmetic operations in the function body. This quickly renders unit polymorphism in unittyped impractical.

In contrast, computing gasoline weight polymorphically is much simpler in units:

```
gasMass :: Fractional n ⇒ Length ℓ n → PerArea ℓ n
                       → Density ℓ n → Mass ℓ n
gasMass dist eco gasden = dist |/| eco |*| gasden
```

The number of type constraints are reduced to one, *Fractional f*. This reflects the property of a coherent unit system that it effectively nondimensionalizes the computation. We need the ℓ to be the same, but it can be anything. This is our solution to the multi-variable multi-constraint unit polymorphism problem: transform it into one unconstrained variable of *LCSU* kind.

5.2 Avoiding Over/underflows by Domain-Specific Scaling

One of the crucial applications of the LCSU mechanism is to provide consistent scaling of quantities for astronomers or chemists, whose research objects are much larger or smaller compared to daily scales. The use of SI units in such fields might cause overflows / underflows and produce nonsensical results.

For example, the Lennard-Jones potential is a model of molecular interaction commonly used in chemistry. The model takes two parameters ϵ and σ and gives the attractive force F as a function of the distance r between two molecules:

$$F = \frac{24\epsilon\sigma^6}{r^7} - \frac{48\epsilon\sigma^{12}}{r^{13}} \tag{1}$$

units can readily model this formula:

$$\begin{aligned}&\mathit{ljForce} :: \mathit{Energy}\ \ell\ \mathit{Float} \to \mathit{Length}\ \ell\ \mathit{Float}\\ &\quad \to \mathit{Length}\ \ell\ \mathit{Float} \to \mathit{Force}\ \ell\ \mathit{Float}\\ &\mathit{ljForce}\ \mathit{eps}\ \mathit{sigma}\ r\\ &\quad = (24 *|\ \mathit{eps}\ |*|\ \mathit{sigma}\ |\hat{}\ \mathit{pSix})\ |/|\ (r\ |\hat{}\ \mathit{pSeven})\\ &\quad |-|(48 *|\ \mathit{eps}\ |*|\ \mathit{sigma}\ |\hat{}\ \mathit{pTwelve})\ |/|\ (r\ |\hat{}\ \mathit{pThirteen})\end{aligned}$$

Then, how many Newtons is the attractive force between two Argon atoms at distance 4 Å? If we type the following into GHCi

```
λ> let sigmaAr = 3.4e-8    % Meter
       epsAr   = 1.68e-21 % Joule
       r       = 4.0e-8    % Meter
λ> (ljForce epsAr sigmaAr r :: Force SI Float) # Newton
```

we get a blunt *NaN* as answer. This is because the Lennard-Jones model involves the inverse 13th to 6th powers of lengths, and $(10^{-8})^6$ is already out of range of the single precision float. (Recall 1 Å= 10^{-8} m.)

Define an LCSU that is more suitable for chemistry, say

```
type CU = MkLCSU '[ '(Length, Angstrom),
   '(Mass, ProtonMass), '(Time, Pico :@ Second)]
```

and you will get a meaningful response just by replacing *SI* with *CU*.

```
λ> (ljForce epsAr sigmaAr r :: Force CU Float) # Newton
9.3407324e-14
```

Laws with such powers are very common, and there are many applications so computationally heavy that the use of double precision is not an option. Most CPUs can perform single precision float arithmetic twice as fast as double precision, and GPUs (Graphic Processing Units) can do even more (4–20 times). As an extreme case, GRAPE series [3, 12], the gravity-force-specific computers widely used by astrophysicists, adopt fixed-point real number representations for speed. On such cutting-edge hardware where real number range is traded off for better performance, users are responsible for choosing the right scale, and units supports doing so.

6. Writing an Astrophysics Research Paper in unittyped and units

In this section, we report the experience of writing an astrophysics paper, titled "Observation of Lightning in Protoplanetary Disks by Ion Lines" [14]. Notably, the paper draft is written in the form of a Haskell program that generates LaTeX.

The development history of the paper consists of two generations, broadly speaking, Gen 1 and Gen 2. The paper draft was initially written using unittyped (Gen 1) and later rewritten using units (Gen 2). It took 9 months to write the initial draft, and 5 working days to rewrite the draft (2447 lines of Haskell code). As a result, we have two programs that deriver the same meaning using different libraries, an ideal material for a comparative study.

Dimension and unit as independent concepts In Gen 1 we had difficulty dealing with dimensions and units separately. In unittyped, the kind of *dim* and *uni* are both $[(\star, Z)]$. Writing "functions" for kinds other than $\star$ requires type-level programming, which is still awkward in Haskell. In Gen 1, the library-supported way to wrap *dim*, *unit* :: $[(\star, Z)]$ into kind $\star$ is by *Value*, which takes a dimension and a unit in pair.

In Gen 2, there are two ways to refer to units and dimensions. One is by unit and dimension combinators (§ 3.2) that produces values of kind $\star$. The other is by type-level list of kind [*Factor*] . The ability to refer to dimensions or units separately, and the existence of two kinds of representations make Gen 2 more expressive.

Defining type synonyms for compound units (c.f. Table 1, (9)) is a frequently used technique in our work (28 such synonyms are defined). In Gen 1, we must specify the destination unit in the type signature, along with its dimensions. Unit conversions (c.f. Table 1, (7)) are another place where we want to specify only units. In Gen 2, we specify only units; the dimension for the unit is inferred. The combinators over kind $\star$ offer ease to the programmer by allowing value-level units (for example *Meter* :^ *pTwo*) that are quite similar to their type-level counterparts (*Meter* :^ *Two*). The more-internal types of kind [*Factor*] make the implementation tractable.

Numerical value accessibility In unittyped, users can convert a numerical value to a quantity, or vice versa, using two functions *mkVal* :: *n* → *Value d u n* and *val* :: *Value d u n* → *n* (c.f. Table 1, (5,6)). Moreover, the type constructor *Value d u* is a *Functor*, so users can apply any numerical function to the numerical value of a quantity.

On the other hand, units is designed to limit user access to numerical values, in order to protect unit consistency. The recommended way for converting between quantities and numerical values is by use of the (%) and (#) operators (c.f. § 3.2), which explicitly take a unit expression as an argument. The direct use of *Qu* constructor is discouraged. [*] *Qu d* ℓ is *not* a *Functor*. *Qu d* ℓ *n* is in the *Floating* class — the type class that allows use of transcendental functions such as *sin* and *exp* — only if *n* is *Floating* *and d* is dimensionless. This meets with our physical expectations of when these functions are applicable.

The expression *x* # *Number* converts *x* to numerical value with type-level assertion that *x* is a dimensionless quantity (c.f. Table 1, (6)). The Gen 1 counterpart, *val x*, has no such assertion. Users can still add the assertion by adding a type signature to *val*, but since it makes the expression longer, users cease to use it. Importantly, the shortest possible expression for conversion function in Gen 2, (#*Number*), comes *with* assertion, while in Gen 1 *val*, is *without*.

In Gen 1, 94 *mkVal*s and 19 *val*s were used to convert quantities of various units to numbers, and vice versa. They are changed to (#)s and (%)s with corresponding units in Gen 2. Each update represents extra checking added in the paper.

Pretty Printing Pretty printing is important because it is where the results are presented to outside world. We symbolically refer to a pretty-printer as *pp*. Several flavor of *pp*s are used in the paper [14], 63 times in total.

In Gen 1, the pretty printer converted only the numerical value to LaTeX expressions, and the unit symbols were typeset "by hand" outside of the checked system. Moreover, the combined use of the pretty printer and *fmap*, such as in *pp* (*fmap* (0.001$\star$) *x*) ++ `"kg"` (Table 1, (8)), occurs 3 times in the Gen 1 source code. Here, the user avoids the use of in-place unit conversion (which was tedious in Gen 1) and manipulates the numerical value and the unit symbol directly. With the pretty printer controlling only the numerical value, we cannot assert the correctness of the quantity expression at the LaTeX-level.

In Gen 2, we redefined the pretty-printers so that each takes a unit and a quantity as arguments, and prints the quantity in the given unit, followed by the unit symbol. This new design allows mapping of quantities from Haskell to LaTeX in the correct sense,

[*] units still exports *Qu* constructor in module *Data.Metrology.Unsafe*

since a quantity consists of a magnitude and a measurement unit. In Gen 1, it was difficult to write pretty-printers that take the unit as an argument, since the value-level representation of units is absent.

Error Messages The readability of the error messages is important to help the users locate and fix unit mistakes.

Error messages of units are relatively instructive, compare to those of unittyped. The separation of dimensions and units is one factor that contributes to the readability because it effectively halves the length of the printout of corresponding types. The error messages, such as `Couldn't match type 'Time' with 'Length'`, are quite suggestive.

Two major shortcomings in the clarity of error messages are with type-level integers (which print in unary), and the *SI* LCSU (which prints in seven lines). It is unclear how to improve this without more control over the error message generation system.

Unit Coherence In Gen 1, the quantities in the paper ranged over multiple systems of units: the cgs (centimeter-gram-second) units, SI units, and domain-specific units such as astronomical units (AUs) and electron volts. Several components of the paper depend on multiple systems of units. At each such place, we have to decide on which system to use and to convert other units. The resultant paper draft was chimera of unit systems.

This undesirable situation disappeared in Gen 2. We choose the SI as the LCSU in all parts of the paper. Still, we could define and pretty-print quantities using arbitrary units as was in Gen 1. The automatic conversion afforded by (%) and (#) allowed us to use our units of choice.

Summary of Comparison We ended our experience with a few takeaways:

- An effective way to achieve unit polymorphism is by representing a quantity as triples of its dimension, the system of units it belongs to, and its numerical value type.
- Having units and dimensions be representable as types of kind $\star$ makes them easier to work with, as we do not have to bother with proxies.
- A library can be designed to hide the value constructor for the quantity type, to refrain users from unit-unsafe computations, and yet remain user friendly. It is important that the shortest ways to write quantity calculus are unit-safe, and are short enough.

7. Discussion and Future Work

The units library allows us to work with a domain-specific system of units, like the chemists' units above. We can use multiple LCSUs in the case of interdisciplinary study such as astrochemistry. units's design of CSU-local computation and manual inter-CSU conversion encourages users to create large LCSU blocks and minimize conversions between them.

The assertion that all unit conversions are eliminated under each LCSU helps us to optimize the underlying computation. Hardware accelerators such as GPUs are becoming popular today, and multiple Haskell libraries for parallel array computation on CPUs and GPUs have been proposed [2, 7, 10, 11, 19]. Expressing laws of hydrodynamics in Haskell has been demonstrated [13]. Such libraries, if used together with easy, consistent scaling and the correctness check of units provided by units, constitute a powerful development environment for computational science.

What is more, we can write a well-documented library with the basic equations of high-school physics without choosing any particular system of units — the library would take the user's choice as a parameter. Putting all of this together, we will be able to teach and study physics in Haskell.

Acknowledgments

We thank Thijs Alkemade for the development of the unittyped library that greatly helped in writing Gen 1 of the astrophysics paper. We thank Iavor Diatchki for the improvements of the type-level naturals in GHC upon our request.

References

[1] Bureau International des Poids et Mesures. International vocabulary of metrology: Basic and general concepts and associated terms. *JCGM*, pages 1–91, 2012.

[2] M. M. T. Chakravarty, G. Keller, S. Lee, T. L. McDonell, and V. Grover. Accelerating Haskell array codes with multicore GPUs. In *Declarative Aspects of Multicore Programming (DAMP '11)*, pages 3–14, 2011.

[3] T. Ebisuzaki, J. Makino, T. Fukushige, M. Taiji, D. Sugimoto, T. Ito, and S. K. Okumura. GRAPE Project: an Overview. *Publications of the Astronomical Society of Japan*, 45:269–278, June 1993.

[4] R. A. Eisenberg and S. Weirich. Dependently typed programming with singletons. In *Haskell Symposium '12*, pages 117–130, 2012.

[5] R. A. Eisenberg, D. Vytiniotis, S. Peyton Jones, and S. Weirich. Closed type families with overlapping equations. In *Principles of Programming Languages (POPL '14)*, pages 671–683, 2014.

[6] P. Guo and S. McCamant. Annotation-less unit type inference for c. *Final Project, 6.883: Program Analysis*, 2005.

[7] G. Keller, M. M. T. Chakravarty, R. Leshchinskiy, S. Peyton Jones, and B. Lippmeier. Regular, shape-polymorphic, parallel arrays in Haskell. In *SIGPLAN Not.*, volume 45, pages 261–272. ACM, 2010.

[8] A. Kennedy. *Programming Languages and Dimensions*. PhD thesis, University of Cambridge, 1996.

[9] A. J. Kennedy. Types for units-of-measure: Theory and practice. In Z. Horvth, R. Plasmeijer, and V. Zsk, editors, *Central European Functional Programming School*, volume 6299 of *Lecture Notes in Computer Science*, pages 268–305. Springer, 2010. .

[10] B. Larsen. Simple optimizations for an applicative array language for graphics processors. In *Declarative Aspects of Multicore Programming (DAMP '11)*, pages 25–34, 2011.

[11] G. Mainland and G. Morrisett. Nikola: Embedding compiled GPU functions in Haskell. *SIGPLAN Not.*, 45(11):67–78, Sept. 2010.

[12] J. Makino, T. Fukushige, M. Koga, and K. Namura. GRAPE-6: Massively-Parallel Special-Purpose Computer for Astrophysical Particle Simulations. *Publications of the Astronomical Society of Japan*, 55:1163–1187, Dec. 2003.

[13] T. Muranushi. Paraiso: an automated tuning framework for explicit solvers of partial differential equations. *Computational Science & Discovery*, 5(1):015003, 2012.

[14] T. Muranushi, E. Akiyama, S. Inutsuka, N. Hideko, and S. Okuzumi. Observation of lightning in protoplanetary disks by ion lines. *The Astrophysical Journal, to be Submitted*, 2014.

[15] W. H. Nelson. The Gimli glider. *Soaring Magazine*, 1997.

[16] P. Roy and N. Shankar. Simcheck: An expressive type system for simulink. In *NASA Formal Methods*, pages 149–160, 2010.

[17] A. G. Stephenson, D. R. Mulville, F. H. Bauer, G. A. Dukeman, P. Norvig, L. S. LaPiana, P. J. Rutledge, D. Folta, and R. Sackheim. Mars climate orbiter mishap investigation board Phase I report, 44 pp. *NASA, Washington, DC*, 1999.

[18] M. Sulzmann, G. J. Duck, S. Peyton-Jones, and P. J. Stuckey. Understanding functional dependencies via constraint handling rules. *Journal of Functional Programming*, 17:83–129, 2007.

[19] J. Svensson, M. Sheeran, and K. Claessen. Obsidian: A domain specific embedded language for parallel programming of graphics processors. In *Implementation and Application of Functional Languages (IFL '08)*, pages 156–173, 2011.

[20] B. A. Yorgey, S. Weirich, J. Cretin, S. Peyton Jones, D. Vytiniotis, and J. P. Magalhães. Giving Haskell a promotion. In *Types in Language Design and Implementation (TLDI '12)*, pages 53–66, 2012.