

# Types for Units-of-Measure: Theory and Practice

Lecture notes for CAFP'09 (Komarno, Slovakia)  
and 'Types At Work' (Copenhagen, Denmark)

Andrew Kennedy  
akenn@microsoft.com

Microsoft Research, Cambridge, UK

## 1 Introduction

Units-of-measure are to science what types are to programming. In science and engineering, dimensional and unit consistency provides a first check on the correctness of an equation or formula, just as in programming the passing of a program by the type-checker eliminates one possible reason for failure.

Units-of-measure errors can have catastrophic consequences, the most famous of which was the loss in September 1999 of NASA's Mars Climate Orbiter probe, caused by a confusion between newtons (the SI unit of force) and lbf (pound-force, a unit of force sometimes used in the US). The report into the disaster made many recommendations [14]. Notably absent, though, was any suggestion that programming languages might assist in the prevention of such errors, either through static analysis tools, or through type-checking.

Over the years, many people have suggested ways of extending programming languages with support for static checking of units-of-measure. It's even possible to abuse the rich type systems of existing languages such as C++ and Haskell to achieve it, but at some cost in usability [18, 6]. More recently, Sun's design for its Fortress programming language has included type system support for dimensions and units [2].

In this short course, we'll look at the design supported by the F# programming language, the internals of its type system, and the theoretical basis for units safety. The tutorial splits into three parts:

- Section 2 is a programmer's guide to units-of-measure in F#, intended to be accessible to any programmer with some background in functional programming.
- Section 3 presents details of the type system and inference algorithm.
- Section 4 considers the semantics of units, including a link with classical dimensional analysis.

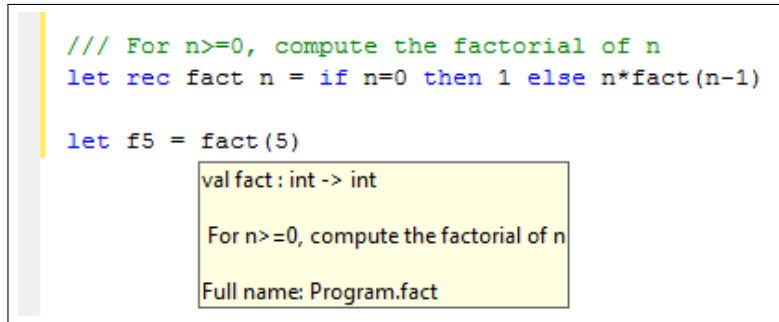


Fig. 1. Editing F# code in Visual Studio

## 2 An introduction to units-of-measure in F#

We begin by taking a gentle tour through the units-of-measure feature of F#. For this you will need the F# command-line compiler (`fsc`) and interactive environment (`fsi`), and optionally (available on Windows only), the Visual Studio integrated development environment. Throughout the tutorial we will present code snippets, like this,

```
let rec fact n = if n=0 then 1 else n * fact(n-1)
```

and also fragments of dialogue from `fsi`, like this:

```
> let rec fact n = if n=0 then 1 else n * fact(n-1);;
val fact : int -> int
```

If you are reading this tutorial online using a program such as Adobe Reader, you can copy-and-paste examples directly into `fsi`.

If you have Visual Studio you might instead prefer to try out code in its editor, as shown in Figure 1. The tooltips feature of VS is especially useful: if you hover the mouse cursor over an identifier, its type will be displayed in a box, along with information gathered from the “`///`” comment preceding its definition.

The F# programming language shares a subset with Caml; this subset will be familiar to SML programmers, and unsurprising for Haskell programmers too. For the most part, we stay within this subset, extended with units-of-measure of course, even though some examples can be made slicker by using the object-oriented and other advanced features of F#. (One significant departure from Caml is the use of F#’s indentation based layout, similar to Haskell’s ‘offside rule’.) Sticking to this subset also illustrates how units-of-measure types and inference could be usefully added to other functional languages. (Language designers, take note!)

## 2.1 Introducing units

We must first declare some base units. As it's the 21st century, we shall employ the SI unit system (*Système International d'Unités* [1]) and declare units for mass, length and time:

```
[<Measure>] type kg // kilograms
[<Measure>] type m // metres
[<Measure>] type s // seconds
```

The [`<Measure>`] attribute in front of `type` indicates that we're not really introducing types in the usual sense, but rather *measure constructors*, namely `kg`, `m` and `s`.

Now let's declare a well-known constant<sup>1</sup> with its units, which we can do simply by tacking the units onto a floating-point literal, in between angle brackets:

```
let gravityOnEarth = 9.808<m/s^2> // an acceleration
```

Notice how conventional notation for units is used, with `/` for dividing, and `^` for powers. Juxtaposition just means product (or you can write `*` if you prefer), and negative powers can be used in place of division. We could express the units of acceleration slightly differently:

```
let gravityOnEarth = 9.808<m * s^-2>
```

Or we might even go against recommended practice and write it this way:

```
let gravityOnEarth = 9.808<m/s/s>
```

What is the type of `gravityOnEarth`? In F# Interactive, the compiler tells us:

```
> let gravityOnEarth = 9.808<m/s/s>;;
val gravityOnEarth : float<m/s ^ 2> = 9.808
```

The `float` type is *parameterized* on units-of-measure, here instantiated with `m/s^2`, just as `list` takes a type parameter, as in `list<int>`. (In F#, ordinary type parameters can be written prefix, as in `int list`, or postfix in angle brackets, as in `list<int>`. Units-of-measure parameters must be written in the latter style.)

Now we can do some physics! If an object is dropped off a building that is 40 metres tall, at what speed will it hit the ground?<sup>2</sup>

```
> let heightOfBuilding = 40.0<m>;;
val heightOfBuilding : float<m> = 40.0
> let speedOfImpact = sqrt (2.0*gravityOnEarth*heightOfBuilding);;
val speedOfImpact : float<m/s> = 28.01142624
```

<sup>1</sup> constant at a particular point on the earth's surface, at least!

<sup>2</sup> Assuming no atmosphere!

```
let heightOfBuilding = 40.0<m>
let speedOfImpact =
    sqrt (2.0 * gravityOnEarth + heightOfBuilding)
The unit of measure 'm' does not match the unit of measure 'm/s ^ 2'
```

Fig. 2. A units error in Visual Studio

Magic! Units-of-measure are not just handy comments-on-constants: they are there in the types of values, and, moreover, F# knows the “rules of units”. When values of floating-point type are multiplied, the units are multiplied too; when they are divided, the units are divided too, and when taking square roots, the same is done to the units. So by the rule for multiplication, the expression inside `sqrt` above must have units  $m^2/s^2$ , and therefore the units of `speedOfImpact` must be  $m/s$ .

What if we make a mistake?

```
> let speedOfImpact = sqrt (2.0*gravityOnEarth+heightOfBuilding);;

let speedOfImpact = sqrt (2.0*gravityOnEarth+heightOfBuilding);;
-----^-----

stdin(142,50): error FS0001: The unit of measure 'm' does not match
the unit of measure 'm/s ^ 2'
```

We’ve tried to add a height to an acceleration, and F# tells us exactly what we’ve done wrong. The units don’t match up, and it tells us so! In Visual Studio, errors are shown interactively: a red squiggle will appear, and if you hover the mouse cursor over it, the error message will be shown, as in Figure 2.

Now let’s do a little more physics. What force does the ground exert on me to maintain my stationary position?

```
> let myMass = 65.0<kg>;;

val myMass : float<kg> = 65.0

> let forceOnGround = myMass*gravityOnEarth;;

val forceOnGround : float<kg m/s ^ 2> = 637.52
```

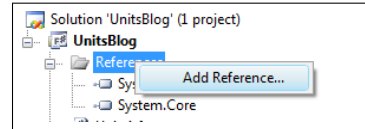
We’ve just applied Newton’s Second Law of motion. Newton’s eponymous unit, the newton, is the SI unit of force. Now instead of the cumbersome  $kg\ m/s^2$  we can introduce a derived unit and just write  $N$ , the standard symbol for newtons.

```
[<Measure>] type N = kg m/s^2
let forceOnGround:float<N> = myMass*gravityOnEarth
```

Derived units are just like type aliases: as far as F# is concerned,  $N$  and  $kg\ m/s^2$  mean exactly the same thing.

## 2.2 Interlude: the F# PowerPack

The F# ‘PowerPack’ library declares all of the SI base and derived units, under the namespace `Microsoft.FSharp.Math.SI`. It also defines various physical constants in the namespace `Microsoft.FSharp.Math.PhysicalConstants`. To import this library, you will need to reference it from your project in Visual Studio, by right-clicking on *References* (see right) and then selecting the `FSharp.PowerPack` component. Alternatively, you can use the `#r` directive in `fsi`, as below:



```
> #r "FSharp.PowerPack";;  
--> Referenced 'C:\Program Files\FSharp\bin\FSharp.PowerPack.dll'
```

Now let’s use the units and constants from the PowerPack to implement Newton’s law of universal gravitation:

$$F = G \frac{m_1 m_2}{r^2}$$

Here  $m_1$  and  $m_2$  are the masses of two bodies,  $r$  is the distance between them,  $G$  is the gravitational constant and  $F$  is the force of attraction between the two bodies. We can code this as follows:

```
> open Microsoft.FSharp.Math;;  
> open SI;;  
> let attract (m1:float<kg>) (m2:float<kg>) (r:float<m>) : float<N> =  
-   PhysicalConstants.G * m1 * m2 / (r*r);;  
  
val attract : float<kg> -> float<kg> -> float<m> -> float<N>  
>
```

Figure 3 shows this function being defined in Visual Studio.

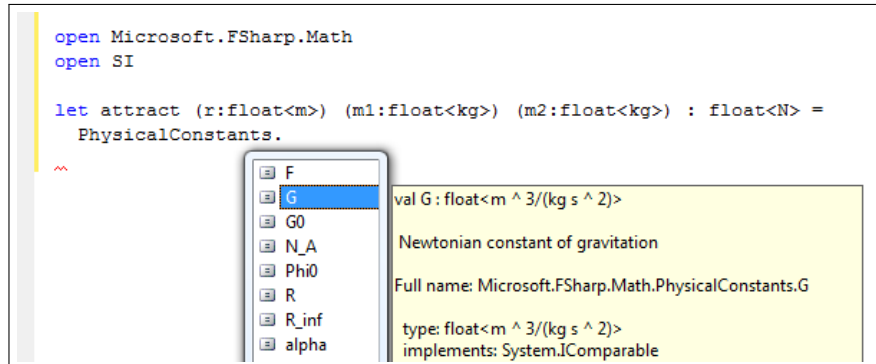
## 2.3 Unit conversions

So for physicists, at least, there is no excuse to go non-metric. But if you insist, you can define units from other systems. Here is our earlier example, using feet instead of metres as the unit of length.

```
[<Measure>] type ft  
let gravityOnEarth = 32.2<ft/s^2>  
let heightOfBuilding = 130.0<ft>  
let speedOfImpact = sqrt (2.0 * gravityOnEarth * heightOfBuilding)
```

What if you need to convert between feet and metres? First, define a conversion factor.

```
let feetPerMetre = 3.28084<ft/m>
```



**Fig. 3.** Using the PowerPack in Visual Studio

Naturally enough, the units of `feetPerMetre` are feet per metre, or `ft/m` for short. Now we can convert distances...

```
let heightOfBuildingInMetres = heightOfBuilding / feetPerMetre
```

...and speeds...

```
let speedOfImpactInMPS = speedOfImpact / feetPerMetre
```

...and we can convert back the other way by multiplying instead of dividing:

```
let speedOfImpactInFPS = speedOfImpactInMPS * feetPerMetre
```

As far as F# is concerned, `ft` and `m` have nothing to do with each other. It's up to the programmer to define appropriate conversion factors. But the presence of units on the conversion factors makes mistakes much less likely. For example, what happens if we divide instead of multiply above? The type of the result suggests that something is awry, and will probably lead to a compile-time error later in the code:

```
> let speedOfImpactInFPS = speedOfImpactInMPS / feetPerMetre;;
val speedOfImpactInFPS : float<m ^ 2/(ft s)> = 8.500500698
```

It's probably a good idea to package up conversion factors with the unit-of-measure to which they relate. A convenient way to do this in F# is to make use of the ability to define static 'members' on types:

```
[<Measure>]
type ft =
    static member perMetre = 3.28084<ft/m>
```

Now we can just write `ft.perMetre`.

## 2.4 Interfacing non-unit-aware code

We've seen how to use syntax such as `2.0<s>` to introduce units-of-measure into the types of floating-point values. But what if a quantity is stored in a file, or

entered by the user through a GUI, or in a web form? In that case it'll probably start out life as a `string`, to be parsed and converted into a `float`. How can we convert a vanilla `float` into, say, a `float<s>`? That's easy: just multiply by `1.0<s>`! Here's an example:

```
let rawString = reader.ReadLine()
let rawFloat = System.Double.Parse(rawString)
let timeInSeconds = rawFloat * 1.0<s>
```

If we want to convert back to a vanilla `float`, say, to pass to a non-units-aware function or method, we just divide by `1.0<s>`:

```
let timeSpan = System.TimeSpan.FromSeconds(timeInSeconds / 1.0<s>)
```

## 2.5 Dimensionless quantities

But hold on – what's going on with that last example? The variable `timeInSeconds` has type `float<s>`, and we divided it by `1.0<s>` which has type `float<s>`. So the units cancel out, producing units which we write simply as the digit `1`. Hence the type of `timeInSeconds / 1.0<s>` is `float<1>`. Such a quantity is called *dimensionless*. Conveniently, F# defines the ordinary `float` type to be an alias for `float<1>`, as if there is a definition

```
type float = float<1>
```

which makes use of overloading on the arity of the type constructor.

Traditionally, angles have been considered dimensionless, since they are defined as the ratio of arc length to radius. The built-in trigonometric functions `sin`, `cos`, and so on, accept dimensionless `floats`. There are, however, several *units* used to measure angles, such as *degrees*, or *revolutions*, in addition to the 'natural' unit *radians*. It's easy enough to define such units, and appropriate conversion factors, if stricter type-checking is required:

```
[<Measure>]
type deg =
    static member perRadian = 360.0<deg> / (2.0 * System.Math.PI)
[<Measure>]
type rev =
    static member perRadian = 1.0<rev> / (2.0 * System.Math.PI)

let l = sin (90.0<deg> / deg.perRadian)
```

## 2.6 Parametric polymorphism

So far, we've seen how to write code that uses specific units-of-measure. But what if we're writing code that doesn't care what units it is working in? Let's start simple. What is the type of `fun x -> x*x`? Well, multiplication is overloaded, and F# defaults to integers:

```
> let sqr x = x*x;;  
  
val sqr : int -> int
```

But if we annotate the argument, we can define squaring for all kinds of floats:

```
> let sqrLength (x:float<m>) = x*x;;  
  
val sqrLength : float<m> -> float<m ^ 2>  
> let sqrSpeed (x:float<m/s>) = x*x;;  
  
val sqrSpeed : float<m/s> -> float<m ^ 2/s ^ 2>
```

This is very painful: we'd really like to write a single, *generic* squaring function, and then re-use it on values with differing units. And indeed, we can do just that:

```
> let sqr (x:float<_>) = x*x;;  
  
val sqr : float<'u> -> float<'u ^ 2>
```

The underscore notation in `float<_>` tells the F# compiler to *infer* the unit-of-measure parameter, and as can be seen from the output, it infers a *generic* or *parametrically-polymorphic* type for squaring. The notation `'u` looks like a type variable, but is in fact a unit-of-measure variable that can be instantiated with any unit-of-measure expression. Let's use it on lengths and speeds:

```
> let d2 = sqr 3.0<m>;;  
  
val d2 : float<m ^ 2> = 9.0  
  
> let v2 = sqr 4.0<m/s>;;  
  
val v2 : float<m ^ 2/s ^ 2> = 16.0
```

F# can fully infer polymorphic unit-of-measure types, with type annotations required only in situations where overloaded operators must be resolved. Moreover, as with ordinary ML-style type inference, it infers the most general, or *principal* type, of which all other possible types are instances. Here are some simple examples:

```
> let cube x = x*sqr x;;  
  
val cube : float<'u> -> float<'u ^ 3>  
  
> let pythagoras x y = sqrt (sqr x + sqr y);;  
  
val pythagoras : float<'u> -> float<'u> -> float<'u>  
  
> let average (x:float<_>) y = (x+y)/2.0;;  
  
val average : float<'u> -> float<'u> -> float<'u>
```



Here's one that requires a bit of head-scratching to understand:

```
> let silly x y = sqr x + cube y;;  
val silly : float<'u ^ 3> -> float<'u ^ 2> -> float<'u ^ 6>
```

We can now see that many of the built-in arithmetic operators and functions have a unit-polymorphic type as one possible overloading:

```
> let add (x:float<_>) y = x+y;;  
val add : float<'u> -> float<'u> -> float<'u>  
  
> let sub (x:float<_>) y = x-y;;  
val sub : float<'u> -> float<'u> -> float<'u>  
  
> let mul (x:float<_>) y = x*y;;  
val mul : float<'u> -> float<'v> -> float<'u 'v>  
  
> let div (x:float<_>) y = x/y;;  
val div : float<'u> -> float<'v> -> float<'u/'v>  
> fun (x : float<_>) -> sqrt x;;  
val it : float<'u ^ 2> -> float<'u> = <fun:clo0>  
> fun x -> sqrt x;;  
val it : float -> float = <fun:clo0-1>
```

The last example here illustrates that without a type annotation, `sqrt` defaults to dimensionless `float`. This is in part to avoid units-of-measure confusing novice programmers, and in part to retain some Caml-compatibility.

*Exercise 1.* Without trying it out, what do you think is the most general type of the following function?

```
let sillier x y z = sqr x + cube y + sqr z * cube z
```

Now try it. Were you right?

## 2.7 Zero

Suppose we want to sum the elements of a list. In true functional style, we use one of the fold operators.

```
> let sum xs = List.fold (+) 0.0 xs;;  
val sum : float list -> float
```

Oops – we don't have a nice polymorphic type! The reason is simple: unless units are specified, constants, including zero, are assumed to have no units, i.e. to be dimensionless. (This means that the subset of F# that coincides with Caml has the same static and dynamic behaviour as in Caml.) So instead, let's give 0.0 some units, but not tell F# what they are, by writing 0.0<\_>:

```
> let sum xs = List.fold (+) 0.0<_> xs;;
val sum : float<'u> list -> float<'u>
```

That's better!

Zero is the only numeric literal that is 'polymorphic' in its units, as is illustrated by the dialogue below.

```
> 0.0<_>;
val it : float<'u> = 0.0
> 1.0<_>;
val it : float = 1.0
```

*Exercise 2.* Can you think why this is so? (When we study the semantics of units, we will see the reason why it *must* be the case.) In fact, there are some other very special floating-point values that are polymorphic. Can you guess what they are?

## 2.8 Application area: statistics

Now let's do some statistics. First, we define the arithmetic mean  $\mu$  of a list of  $n$  numbers  $[a_1; \dots; a_n]$ , as given by

$$\mu = \frac{1}{n} \sum_{i=1}^n a_i.$$

This is easy, using the `sum` function defined earlier:

```
> let mean xs = sum xs / float (List.length xs);;
val mean : float<'u> list -> float<'u>
```

*Exercise 3.* Write a unit-polymorphic function to compute the standard deviation  $\sigma$  of a list of  $n$  numbers  $[a_1; \dots; a_n]$ , as given by

$$\sigma^2 = \frac{1}{n} \sum_{i=1}^n (a_i - \mu)^2.$$

The *geometric mean* is given by

$$g = (a_1 \cdot a_2 \cdots a_n)^{\frac{1}{n}}$$

A straightforward implementation of this function does not get assigned a nice polymorphic type:

```
> let recipen x = 1.0 / float (List.length x)
val recipen : 'a list -> float
```

```

> let gmean (x:float<_> list) = List.reduce ( * ) x ** recipLen x;;

let gmean (x:float<_> list) = List.reduce ( * ) x ** recipLen x;;
-----^

stdin(15,21): warning FS0191: This code is less generic than
indicated by its annotations. A unit-of-measure specified using
'_' has been determined to be '1', i.e. dimensionless. Consider
making the code generic, or removing the use of '_'.

val gmean : float list -> float

```

We have ‘hinted’ to the compiler that we want a parameterized `float` type, but it has inferred a dimensionless type, and so it helpfully emits a warning. But why did it not infer the type `float<'u> list -> float<'u>`? The answer lies in the type of the product *and* in the type of the exponentiation operator `**`. Consider first the expression `List.reduce ( * ) x`, which implements the product  $a_1 \cdot a_2 \cdots a_n$ . If `x` has type `float<'u> list` then we might expect it to have type `float<'un>`. But  $n$  is the length of the list, which is not known statically – we want a *dependent* type! Furthermore, the type of exponentiation is

```
val ( ** ) : float -> float -> float
```

in which both its arguments and result are dimensionless.

*Exercise 4.* With a little work, it is possible to write a function `gmean` that is assigned the polymorphic type that we expected. Hint: consider ‘normalizing’ the list before computing the product.

## 2.9 Application area: calculus

Now let’s do some calculus. Higher-order functions abound here: for example, differentiation is a higher-order function  $D : (\mathbb{R} \rightarrow \mathbb{R}) \rightarrow (\mathbb{R} \rightarrow \mathbb{R})$ . To numerically differentiate a function  $f$  to obtain its approximate derivative  $f'$ , we can use the formula

$$f'(x) \approx \frac{f(x+h) - f(x-h)}{2h},$$

where  $h$  represents a small change in  $x$ . Let’s write this in F#:

```

> let diff (h:float<_>) (f:_ -> float<_>) = fun x -> (f(x+h)-f(x-h))
  / (2.0*h);;
-
val diff : float<'u> -> (float<'u> -> float<'v>) -> float<'u> ->
float<'v/'u>

```

Notice how the type of `diff` precisely describes the effect of differentiation on the units of a function. For example, let’s use it to compute the rate of change of the gravitational force between the earth and the author as the author moves away from the earth:

```

> let earthMass = 5.9736e24<kg>;

val earthMass : float<kg> = 5.9736e+24

> diff 0.01<m> (attract myMass earthMass);;
val it : (float<m> -> float<kg/s ^ 2>) = <fun:it54-6>

```

The units  $\text{kg/s}^2$  appear a little strange, but we can confirm that they really are ‘force per unit length’:

```

> (diff 0.01<m> (attract myMass earthMass) : float<m> -> float<N/m>)
;;
val it : (float<m> -> float<N/m>) = <fun:it57-8>

```

We can likewise integrate a function, using one of the simplest methods: the trapezium rule [17, Section 4.1]. It is defined by the following formula, which gives an approximation to the area under the curve defined by  $f$  in the interval  $a \leq x \leq b$  using  $n + 1$  values of  $f(x)$ :

$$\int_a^b f(x) dx \approx \frac{h}{2} (f(a) + 2f(a+h) + \dots + 2f(b-h) + f(b)), \quad h = \frac{b-a}{n}.$$

Here is an implementation in F#:

```

let integrate (a:float<_>) (b:float<_>) n (f:_ -> float<_>) =
  let h = (b-a) / (float n)
  let rec iter x i =
    if i=0 then 0.0<_>
    else f x + iter (x+h) (i-1)
  h * (f a / 2.0 + iter (a+h) (n-1) + f b / 2.0)

```

*Exercise 5.* Without typing it in, what do you think the unit-polymorphic type of `integrate` is?

Our final example is an implementation of the Newton-Raphson method for finding roots of equations, based on the iteration of

$$x_{n+1} = x_n - \frac{f(x_n)}{f'(x_n)}.$$

This method calculates a solution of  $f(x) = 0$ , making use of the derivative  $f'$ . The F# code is as follows:

```

let rec newton (f:float<_> -> float<_>) f' x xacc =
  let dx = f x / f' x
  let x' = x - dx
  if abs dx / x' < xacc
  then x'
  else newton f f' x' xacc

```

It accepts a function `f`, its derivative `f'`, an initial guess `x` and a relative accuracy `xacc`. Its type is

```

val newton :
  (float<'u> -> float<'v>) ->
    (float<'u> -> float<'v/'u>) -> float<'u> -> float -> float<'u>

```

## 2.10 Unit-parameterized types

So far we have seen units applied to `float`, the built-in primitive type of double-precision floating-point numbers. The types `float32` and `decimal` can be parameterized by units too:

```

[<Measure>] type USD
[<Measure>] type yr
let fatcatsalary = 1000000M<USD/yr> // decimal constants have suffix M
let A4paperWidth = 0.210f<m> // float32 constants have suffix f

```

It's natural to parameterize other numeric types on units, such as a type for complex numbers, or a type for vectors in 3-space. Or a unit-parameterized numeric type might be required for a component of some other type, such as a scene model type for a ray-tracer, and so that type must be parameterized too. Such types are supported in F# simply by marking the parameters with the `[<Measure>]` attribute:

```

// Record types parameterized by units
type complex< [Measure] 'u> = { re:float<'u>; im:float<'u> }
type vector3< [Measure] 'u> = { x:float<'u>; y:float<'u>; z:float<'u> }
type sphere< [Measure] 'u> = { centre:vector3<'u>; radius:float<'u> }
// A discriminated union parameterized by units
type obj< [Measure] 'u> =
  | Sphere of sphere<'u>
  | Group of obj<'u> list

```

We can now instantiate such types with concrete units:

```

> let gravity = { x = 0.0<_>; y = 0.0<_>; z = -9.808<m/s^2> };;

val gravity : vector3<m/s ^ 2> = {x = 0.0;
                               y = 0.0;
                               z = -9.808;}

> let scene = Group [Sphere {centre={x=2.0<m>;y=3.0<m>;z=4.0<m>};
                           radius = 1.5<m> }];;

val scene : obj<m> = Group [Sphere {centre = {x = 2.0;
                                             y = 3.0;
                                             z = 4.0;};
                              radius = 1.5;}]

```

It's straightforward to write functions over such types, with appropriate unit-polymorphic types. For example, the dot product of two vectors can be defined as follows, with its type inferred automatically:

```
> let dot v w = v.x*w.x + v.y*w.y + v.z*w.z;;
val dot : vector3<'u> -> vector3<'v> -> float<'u 'v>
```

Here are functions to convert from and to the polar representation of complex numbers:

```
let fromPolar (m:float<_>, p) = { re = m*cos p; im = m*sin p }
let magnitude c = sqrt (c.re*c.re + c.im*c.im)
let phase c = atan2 c.im c.re
```

It's desirable to define unit-polymorphic arithmetic operators for types such as `complex` and `vector3`. This is supported through the use of 'static members':<sup>3</sup>

```
type complex< [Measure] 'u> = { re:float<'u>; im:float<'u>} with
static member (+) (a:complex<'u>,b) = { re = a.re+b.re; im = a.im+a.im }
static member (-) (a:complex<'u>,b) = { re = a.re-b.re; im = a.im-a.im }
static member ( * ) (a:complex<'u>,b:complex<'v>) =
    { re = a.re * b.re - a.im * b.im; im = a.im * b.re + b.im * a.re }
```

We can now use complex numbers to do arithmetic on AC voltages:

```
> let voltage1 : complex<V> = polar 2.0<V> 0.0;; // 2 volts
val voltage1 : complex<V> = {re = 2.0;
                             im = 0.0;}

> let voltage2 = polar 3.0<V> (System.Math.PI/4.0);; // 3 volts at 45
    degree phase
val voltage2 : complex<kg m ^ 2/(A s ^ 3)> = {re = 2.121320344;
                                             im = 2.121320344;}

> magnitude (voltage1 + voltage2);;
val it : float<kg m ^ 2/(A s ^ 3)> = 4.635221826
> phase (voltage1 + voltage2);;
val it : float = 0.4753530566
```

## 2.11 Polymorphic recursion

For our final example of unit-parameterized types we return to calculus. We saw that given a function of type `float<'u> -> float<'v>`, its derivative has type `float<'u> -> float<'v/'u>`. Naturally enough, its second-order derivative (the derivative of the derivative) has type `float<'u> -> float<'v/'u^2>`. But what if we want to repeat this process, and create a list of successive derivatives? We can't just use the built-in `list` type, because it requires list elements to each have the same type. We want a list in which the types of successive elements are *related* but not the same. We can define such a custom type as follows:

<sup>3</sup> Full units support only from F# beta1 onwards

```

type derivs<[<Measure>] 'u, [<Measure>] 'v> =
| Nil
| Cons of (float<'u> -> float<'v>) * derivs<'u,'v/'u>

```

An expression `Cons(f,Cons(f',Cons(f'',...)))` of type `derivs<'u,'v>` represents a function `f` of type `float<'u> -> float<'v>`, its first-order derivative `f'` of type `float<'u> -> float<'v/'u>`, its second-order derivative `f''` having type `float<'u> -> float<'v/'u^2>`, and so on. The type makes use of *polymorphic recursion* in its definition, meaning that the recursive reference (`derivs`) is used at a type distinct from its definition. (This is also called a *nested* or *non-regular* datatype in the literature.)

In order to use such a type in a recursive function, the type of the function must be annotated fully with its type, as type inference for polymorphic recursion is in general undecidable. Here is a function that makes use of our earlier numerical differentiation function `diff` to compute a list of the first `n` derivatives of a function `f`.

```

let rec makeDerivs<[<Measure>] 'u, [<Measure>] 'v>
(n:int)
(h:float<'u>)
(f:float<'u> -> float<'v>) : derivs<'u,'v> =
if n=0 then Nil else Cons(f, makeDerivs (n-1) h (diff h f))

```

### 3 Polymorphic type inference for units-of-measure

Ever since Robin Milner’s classic paper on polymorphic type inference [13], researchers have developed ever more sophisticated means of automatically inferring types for programs. Pottier’s survey article presents a good overview of the state-of-the-art [16, §10]. The variety of parametric polymorphism described in Milner’s original work has become known as “**let**-polymorphism”, being equivalent in expressivity to the simple typing of a program after expanding all **let**-bindings. One fruitful direction for research has been to support ‘first-class’ polymorphism in the style of System F and beyond [11, 10, 20]. Others have extended inference to handle new type constructs such as Generalized Algebraic Data Types (GADTs) or existentials [19]. Another direction is to consider polymorphism over entities other than just types, for example records or effects. Units-of-measure are an instance of this idea.

Immediately we should address the question: how are units *different* from types? Even the original ML type system can be used to encode all sorts of invariants such as well-formed contexts [12] and well-formed lambda terms [4]. We could perhaps define dummy type constructors `UOne`, `UProd` and `UInv`, together with base units, and then build derived units such as `UProd<m, UInv<UProd<s, s>>>` for accelerations. The crucial aspect of units-of-measure that is not captured here is *equations* that hold between syntactically distinct units. For example, the units `m s` are equivalent to `s m`; and `s s^-1` can be simplified to `1`. Any encoding of units in types must somehow encode this *equational theory* of units, typically requiring some kind of witnesses to applications of properties such as commutativity and associativity. Better to build this theory in: and moreover, it turns out that the equational theory has some very pleasant properties that support full type inference.

#### 3.1 Grammar for units

First let’s pin down a formal grammar for unit expressions:

$$u, v, w ::= b \mid \alpha \mid 1 \mid u * v \mid u^{-1}$$

We use  $u, v$  and  $w$  for unit expressions. They’re built from

- base units such as `kg`, ranged over by  $b$ , and drawn from a set `UBase`;
- unit variables, which we write as  $\alpha, \beta$ , etc., and drawn from a set `UVars`;
- the ‘unit’ unit, written `1`, representing dimensionless quantities without units;
- product of units, written  $u * v$ ; and
- inverse of units, written  $u^{-1}$ .

Let  $\text{vars}(u)$  be the unit variables occurring in unit expression  $u$ . For example,  $\text{vars}(\alpha * \beta^{-1}) = \{\alpha, \beta\}$ .

For the surface syntax, we de-sugar quotients and integer powers of units:

$$u/v = u * v^{-1}$$

$$u^n = \begin{cases} u * u^{(n-1)} & \text{if } n > 0, \\ 1 & \text{if } n = 0, \\ u^{-1} * u^{(n+1)} & \text{if } n < 0. \end{cases}$$



$$\begin{array}{ccc}
\frac{}{u =_U u} \text{ (refl)} & \frac{u =_U v}{v =_U u} \text{ (sym)} & \frac{u =_U v \quad v =_U w}{u =_U w} \text{ (trans)} \\
\frac{u =_U v}{u^{-1} =_U v^{-1}} \text{ (cong1)} & \frac{u =_U v \quad u' =_U v'}{u * u' =_U v * v'} \text{ (cong2)} & \frac{}{u * 1 =_U u} \text{ (id)} \\
\frac{}{(u * v) * w =_U u * (v * w)} \text{ (assoc)} & \frac{}{u * v =_U v * u} \text{ (comm)} & \frac{}{u * u^{-1} =_U 1} \text{ (inv)}
\end{array}$$

**Fig. 4.** Equational theory of units  $=_U$

For clarity, we will often write  $u^n$  for powers.

### 3.2 Equations for units

Units of measure obey rules of associativity, commutativity, identity and inverses, and thus form an Abelian group. We can formalize this as an equivalence relation on unit expressions  $=_U$ , defined inductively by the rules of Figure 4. These rules just say that  $=_U$  is the smallest congruence relation that is closed under the Abelian group axioms.

The relation  $=_U$  is an example of an *equational theory*. Other common examples are AC (with just the axioms for associativity and commutativity), AC1 (adding the identity axiom to AC), and ACI (adding an axiom for idempotency to AC). These three theories are all *regular*, meaning that if a term  $t$  is equivalent to a term  $t'$  under the equational theory, then  $\text{vars}(t) = \text{vars}(t')$ . In contrast, our theory AG (Abelian groups) is *non-regular*, due to the axiom (inv) whose left and right sides do not have matching sets of variables. This is the source of many challenges, as we shall see!

Baader and Nipkow's book on term rewriting provides a good introduction to equational theories, and the problem of unification that we study later [3].

### 3.3 Deciding equations

An obvious first question is: how can we *decide* whether two unit expressions  $u$  and  $v$  are equivalent, *i.e.* is the equation  $u =_U v$  *valid*? This is straightforward, as unit expressions can be put into a unique *normal form* with respect to  $=_U$ , as follows:

$$\alpha_1^{x_1} * \dots * \alpha_m^{x_m} * b_1^{y_1} * \dots * b_n^{y_n}$$

Here the  $x$ 's and  $y$ 's are non-zero integers, and the unit variables and base units are ordered alphabetically. (In fact the F# compiler does something similar when displaying units-of-measure, except that units with negative powers are separated from those with positive powers.) Hence to check whether two unit-of-measure expressions are equivalent, they must simply be put into normal form and then checked syntactically for equality.

*Exercise 6.* Write a family of functions  $exp_u : (\text{UBase} \cup \text{UVars}) \rightarrow \mathbb{Z}$  that, given a base unit or unit variable, returns an integer representing its exponent in the unit expression  $u$ .

It's easy to show that  $exp_u = exp_v$  if and only if  $u =_U v$ . The function  $exp_u$  is a more 'semantic' representation of units than unit expressions, and is often nicer to work with. Indeed, a recent Coq formalization of ideas presented in Section 4 simply makes the following definitions for unit variables and unit expressions [9]:

```
Definition Var := nat.
Definition Unt := Var -> Z.
```

### 3.4 Solving equations

For languages such as Pascal and Java that support only type *checking*, it is relatively easy to add support for checking units-of-measure, using the procedure outlined in the previous section. But for F#, we go further, and support full type *inference*. To do this, we must not only *check* equations; we must *solve* them. Consider the following dialogue:

```
> let area = 20.0<m^2>;;
val area : float<m ^ 2> = 20.0
> let f (y:float<_>) = area + y*y;;
val f : float<m> -> float<m ^ 2>
```

In order to infer a type for `f`, the compiler will generate a fresh unit variable  $\alpha$  for the units of `y`, and will then solve the equation

$$\alpha^2 =_U \text{m}^2$$

in order to satisfy the requirements of the addition operation. Of course, this equation is easy to solve (just set  $\alpha := \text{m}$ ), but in general, the equation can have arbitrarily complex units on either side.

Moreover, there may be many ways to solve the equation. Consider the equation

$$\alpha * \beta =_U \text{m}^2$$

for which  $\{\alpha := \text{m}, \beta := \text{m}\}$ ,  $\{\alpha := \text{m}^2, \beta := 1\}$ , and  $\{\alpha := 1, \beta := \text{m}^2\}$  are three distinct solutions. These solutions are all *ground*, meaning that they contain only base units and no variables. All three, though, are subsumed by the non-ground, 'parametric' solution  $S = \{\alpha := \beta^{-1} * \text{m}^2\}$ , in the following sense: each can be presented as  $\{\beta := u\} \circ S$  for some  $u$ . (Check this: set  $u$  to be `m`, `1` and `m^2` respectively.) In fact, here  $S$  is the *most general* solution for this equation from which all others can be derived.

*Exercise 7.* Present solutions in  $\alpha$  and  $\beta$  to the equation

$$\alpha * \beta =_U \mathbf{kg} * \mathbf{s}$$

Can you express a *most general* solution? Can you express a most general solution without using inverse?

The idea of solving equations by computing a syntactic substitution for its variables is known as *unification*, and will be familiar to anyone who has studied the type inference algorithms for ML, Haskell, or other similar languages. The well known *principal types* property for those languages relies on the following property of syntactic unification: if two types are unifiable, then there exists a *most general* unifier from which all other unifiers can be derived.

For units-of-measure we must solve equations with respect to the equational theory of Abelian groups. In general, for an equational theory  $=_E$ , an *E-unification problem* is the following: for given terms  $t$  and  $u$ , find a substitution  $S$  such that  $S(t) =_E S(u)$ .

If the terms are unifiable at all, there can be an infinite number of possible unifiers. We say that a unifier  $S_1$  is *more general than* a unifier  $S_2$  and write  $S_1 \preceq_U S_2$  if  $R \circ S_1 =_E S_2$  for some substitution  $R$ . (And we say that  $S_2$  is an *instance of*  $S_1$ .) In contrast to syntactic unification, for equational unification there may not be a single most general unifier. For example, in the presence of nullary constants, unification problems in *AC1* may have a finite set of unifiers of which all unifiers are instances, but which are not instances of each other. This was hinted at in Exercise 7: under *AC1*, in which the rule (inv) of Abelian groups is not available, there are in fact four incomparable unifiers, namely  $\{\alpha := \mathbf{kg}, \beta := \mathbf{s}\}$ ,  $\{\alpha := \mathbf{s}, \beta := \mathbf{kg}\}$ ,  $\{\alpha := 1, \beta := \mathbf{kg} * \mathbf{s}\}$  and  $\{\alpha := \mathbf{kg} * \mathbf{s}, \alpha := 1\}$ .

### 3.5 A unification algorithm

We are fortunate that the theory of Abelian groups with nullary constants is *unitary*, the technical term for “possesses most general unifiers”. (Rather few equational theories have this property; one other is the theory of Boolean Rings.) Moreover, unification is decidable, and the algorithm is straightforward.

Figure 5 presents the algorithm *Unify*, which takes a pair of unit expressions  $u$  and  $v$  and either returns a substitution  $S$  such that  $S(u) =_U S(v)$ , or returns fail indicating that no unifier exists. First observe that the equation

$$u =_U v$$

is equivalent (i.e. has the same set of solutions) to

$$u * v^{-1} =_U 1,$$

and so unification can be reduced to the problem of matching against 1.

The core algorithm *UnifyOne* is a variant of Gaussian elimination, and works by iteratively applying a solution-set-preserving substitution that reduces the

$$\text{Unify}(u, v) = \text{UnifyOne}(u * v^{-1})$$

$$\begin{aligned} \text{UnifyOne}(u) = & \\ \text{let } u = \alpha_1^{x_1} * \dots * \alpha_m^{x_m} * b_1^{y_1} * \dots * b_n^{y_n} \text{ where } |x_1| \leq |x_2|, \dots, |x_m| & \\ \text{in} & \\ \text{if } m = 0 \text{ and } n = 0 \text{ then } id & \\ \text{if } m = 0 \text{ and } n \neq 0 \text{ then fail} & \\ \text{if } m = 1 \text{ and } x_1 \mid y_i \text{ for all } i \text{ then } \{\alpha_1 \mapsto b_1^{-y_1/x_1} * \dots * b_n^{-y_n/x_1}\} & \\ \text{if } m = 1 \text{ otherwise then fail} & \\ \text{else } S_2 \circ S_1 \text{ where} & \\ S_1 = \{\alpha_1 \mapsto \alpha_1 * \alpha_2^{-\lfloor x_2/x_1 \rfloor} * \dots * \alpha_m^{-\lfloor x_m/x_1 \rfloor} * b_1^{-\lfloor y_1/x_1 \rfloor} * \dots * b_n^{-\lfloor y_n/x_1 \rfloor}\} & \\ S_2 = \text{UnifyOne}(S_1(u)) & \end{aligned}$$

**Fig. 5.** Unification algorithm for units-of-measure

size of the minimum exponent in the normal form (the ‘else’ clause) until it reaches an equation containing at most one variable. Consider each subcase in turn:

- If there are no variables ( $m = 0$ ) then either we have 1 ( $n = 0$ ), and so we’re done, or we have just base units ( $n \neq 0$ ), and so there is no unifier.
- If there is exactly one variable ( $m = 1$ ) then we check to see if its exponent divides all of the exponents of the base units. If not, then there is no unifier: an example would be the unification problem  $\alpha^2 =_U \text{kg}^3$ . If it does divide, then the unifier is immediate: an example would be the problem  $\alpha^2 =_U \text{m}^2 * \text{s}^{-4}$  with unifier  $\{\alpha := \text{m} * \text{s}^{-2}\}$ .
- Otherwise, we recurse, after applying a substitution  $S_1$ , whose effect is to transform the unit expression to

$$\alpha_1^{x_1} * \alpha_2^{x_2 \bmod x_1} * \dots * \alpha_m^{x_m \bmod x_1} * b_1^{y_1 \bmod x_1} * \dots * b_n^{y_n \bmod x_1}.$$

Termination is ensured as the size of the smallest exponent, which was  $x_1$ , has been reduced, as each of the  $\_ \bmod x_1$  is smaller than  $x_1$ .

The following theorem characterizes the result of *Unify* as the most general unifier.

**Theorem 1 (Soundness and completeness of *Unify*).**

(Soundness) If  $\text{Unify}(u, v) = S$  then  $S(u) =_U S(v)$ .

(Completeness) If  $S(u) =_U S(v)$  then  $\text{Unify}(u, v) \preceq_U S$ .

*Proof.* See [7].

*Exercise 8.* Compute the most general unifier to the equation

$$\alpha^2 * \beta =_U \text{m}^6$$

$$\begin{aligned}
TU\text{unify}(\alpha, \alpha) &= id \\
TU\text{unify}(\alpha, \tau) &= TU\text{unify}(\tau, \alpha) = \begin{cases} \text{fail} & \text{if } \alpha \text{ in } \tau \\ \{\alpha := \tau\} & \text{otherwise.} \end{cases} \\
TU\text{unify}(\text{float}\langle u \rangle, \text{float}\langle v \rangle) &= \text{Unify}(u, v) \\
TU\text{unify}(\tau_1 \rightarrow \tau_2, \tau_3 \rightarrow \tau_4) &= S_2 \circ S_1 \\
&\quad \text{where } S_1 = TU\text{unify}(\tau_1, \tau_3) \\
&\quad \text{and } S_2 = TU\text{unify}(S_1(\tau_2), S_1(\tau_4))
\end{aligned}$$

**Fig. 6.** Unification algorithm for types

### 3.6 Types

As we saw in Section 2.10, type constructors in  $F\#$  can take both types and units-of-measure as parameters. The distinction is enforced by a trivial *kind* system, in which type parameters are one of two kinds, *type*, the default, and *measure*, as indicated by the attribute [`<Measure>`]. We won't formalize this kind system here, instead focusing on parameterization by units-of-measure alone. But for the purposes of describing the inference algorithm, we do need to include a notion of type variable, which we will write as  $\alpha$ ,  $\beta$ , etc. There will always be enough context to distinguish type and units-of-measure variables.

Let's suppose that the grammar of types is really simple, consisting just of type variables, floating-point types parameterized by units and functions:

$$\tau ::= \alpha \mid \text{float}\langle u \rangle \mid \tau \rightarrow \tau$$

This will be enough to illustrate the process of type inference.

The relation  $=_U$  extends in an obvious way to types, the important point being that  $\text{float}\langle u \rangle =_U \text{float}\langle v \rangle$  if and only if  $u =_U v$ . A simple but inefficient unification algorithm for types with respect to  $=_U$  makes use of our existing unification algorithm for units in the case for `float`. This is shown in Figure 6. The most general unifier property for units of measure extends to types.

**Theorem 2 (Soundness and completeness of  $TU\text{unify}$ ).**

(*Soundness*) If  $TU\text{unify}(\tau_1, \tau_2) = S$  then  $S(\tau_1) =_U S(\tau_2)$ .

(*Completeness*) If  $S(\tau_1) =_U S(\tau_2)$  then  $TU\text{unify}(\tau_1, \tau_2) \preceq_U S$ .

*Proof.* See [7].

### 3.7 Type schemes

Polymorphic types as displayed in  $F\#$  are in fact shorthand for type *schemes* of the form

$$\sigma ::= \forall \alpha_1, \dots, \alpha_n. \tau$$

Top-level type schemes as presented to the programmer are always *closed*, meaning that all variables in the type  $\tau$  are bound by the  $\forall$  quantifier: the variables

are *implicitly* quantified. But internally, the type system and inference algorithm will make use of open schemes such as  $\forall\alpha.\text{float}\langle\alpha\rangle \rightarrow \text{float}\langle\beta\rangle$  in which  $\beta$  is *free*.

Type schemes can be *instantiated* to types by replacing occurrences of unit variables by unit expressions. Formally, we write  $\sigma \preceq \tau$  if  $\sigma = \forall\alpha_1, \dots, \alpha_n.\tau'$  instantiates to  $\tau$ , that is  $\tau = \{\alpha_1 := u_1, \dots, \alpha_n := u_n\}\tau'$  for some unit expressions  $u_1, \dots, u_n$ . We write  $\sigma \preceq_U \tau$  if  $\sigma \preceq \tau'$  and  $\tau' =_U \tau$  for some  $\tau'$ .

*Exercise 9.* Show that  $\forall\alpha.\text{float}\langle\alpha * \text{kg}\rangle \rightarrow \text{float}\langle\alpha * \text{kg}\rangle \preceq_U \text{float}\langle 1 \rangle \rightarrow \text{float}\langle 1 \rangle$ .

### 3.8 The type system and inference algorithm

We are finally ready to present a small subset of the F# type system! Suppose we have expressions with syntax

$$e ::= x \mid \lambda x.e \mid e \mid \text{let } x = e \text{ in } e$$

with constants and primitive operations assumed to be declared in some ‘pervasive’ environment. Figure 7 presents a polymorphic type system for this language. Here the typing judgment takes the form  $V; \Gamma \vdash e : \tau$ , with a map  $\Gamma$  from variables to type *schemes* whose free unit variables are drawn from the set  $V$ , and a type  $\tau$  (not a scheme) with unit variables also drawn from  $V$ .

$$\begin{array}{c}
\text{(var)} \frac{}{V; \Gamma, x:\sigma \vdash x : \sigma} \sigma \preceq \tau \qquad \text{(app)} \frac{V; \Gamma \vdash e_1 : \tau_1 \rightarrow \tau_2 \quad V; \Gamma \vdash e_2 : \tau_1}{V; \Gamma \vdash e_1 e_2 : \tau_2} \\
\text{(let)} \frac{V, \bar{\alpha}; \Gamma \vdash e_1 : \tau_1 \quad V; \Gamma, x:\forall\bar{\alpha}.\tau_1 \vdash e_2 : \tau_2}{V; \Gamma \vdash \text{let } x = e_1 \text{ in } e_2 : \tau_2} \qquad \text{(abs)} \frac{V; \Gamma, x:\tau_1 \vdash e : \tau_2}{V; \Gamma \vdash \lambda x.e : \tau_1 \rightarrow \tau_2} \\
\text{(eq)} \frac{V; \Gamma \vdash e : \tau_1}{V; \Gamma \vdash e : \tau_2} \tau_1 =_U \tau_2
\end{array}$$

**Fig. 7.** Typing rules

In essence, it is a very modest extension to the usual ML type system, such as that presented in Pierce’s book on types and programming languages [15]. Apart from the use of unit variables in place of type variables in schemes, the main addition is the rule (eq), that incorporates the equational theory  $=_U$  into the type system. (Compare the subsumption rule from systems that support subtyping.)

It would be tempting to think that ML’s inference algorithm can be adapted as easily, simply by replacing the use of syntactic unification by our equational

unification algorithm *TUnify*. This is *almost* the case: the algorithm constructed this way is sound (types that it infers are correct) but unfortunately not complete (there may be typeable programs that it rejects). The problem lies in the technique used to ‘generalize’ unit variables when type-checking `let` expressions. As we shall see in the next section, even the notion of ‘free unit variables in a scheme’ is not well-defined, and so the usual method of generalizing over all variables present in the type of the `let`-bound variable but not in the environment, does not work. There is not sufficient space in these notes to discuss the solution; the interested reader should consult [7].

### 3.9 Type scheme equivalence

A type scheme represents an infinite family of types, and we define

$$\text{Insts}_U(\sigma) = \{\tau \mid \sigma \preceq_U \tau\}.$$

This provides a natural notion of type scheme equivalence, namely that two schemes are equivalent if they instantiate to the same set of types. We write

$$\sigma_1 \cong_U \sigma_2 \text{ iff } \text{Insts}_U(\sigma_1) = \text{Insts}_U(\sigma_2)$$

For vanilla ML types, type scheme equivalence is trivial: the only way that schemes can differ is in naming and redundancy of type variables. For example, writing  $\cong$  for equivalence, we have

$$\forall \alpha \beta \gamma. \alpha \rightarrow \beta \rightarrow \alpha \cong \forall \alpha \beta. \beta \rightarrow \alpha \rightarrow \beta.$$

For units-of-measure, in addition to naming, we have the underlying equivalence  $=_U$  on types, but it goes further than that. For example, consider the following three equivalent type schemes for the division operation:

$$\begin{aligned} \forall \alpha \beta. \text{float}\langle \alpha \rangle \rightarrow \text{float}\langle \beta \rangle \rightarrow \text{float}\langle \alpha * \beta^{-1} \rangle \\ \forall \alpha \beta. \text{float}\langle \alpha * \beta \rangle \rightarrow \text{float}\langle \beta \rangle \rightarrow \text{float}\langle \beta \rangle \\ \forall \alpha \beta. \text{float}\langle \alpha \rangle \rightarrow \text{float}\langle \beta^{-1} \rangle \rightarrow \text{float}\langle \alpha * \beta \rangle \end{aligned}$$

The bodies of these type schemes differ even with respect to  $=_U$ , yet they instantiate under  $\preceq_U$  to the same set of types.

The number of quantified variables can differ, and additional ‘free’ variables may occur. Consider the following three type schemes for the reciprocal function `fun (x:float<_>) -> 1.0/x:`

$$\begin{aligned} \forall \alpha. \text{float}\langle \alpha \rangle \rightarrow \text{float}\langle \alpha^{-1} \rangle \\ \forall \alpha \beta. \text{float}\langle \alpha * \beta \rangle \rightarrow \text{float}\langle \alpha^{-1} * \beta^{-1} \rangle \\ \forall \alpha. \text{float}\langle \alpha * \beta^{-1} \rangle \rightarrow \text{float}\langle \alpha^{-1} * \beta \rangle \end{aligned}$$

It’s apparent from this example that the usual notion of ‘free variable’ is not respected with type scheme equivalence, even after applying the normal form with respect to  $=_U$ . This is one reason why the usual generalization procedure in type inference does not work.

However, it is easy to show the following.

**Lemma 1.** *Let  $\sigma_1 = \forall \bar{\alpha}. \tau_1$  and  $\sigma_2 = \forall \bar{\alpha}. \tau_2$ . Then  $\sigma_1 \cong_U \sigma_2$  iff  $\sigma_1 \preceq_U \tau_2$  and  $\sigma_2 \preceq_U \tau_1$ .*

In other words, we must merely find substitutions both ways to show that type schemes are equivalent.

*Exercise 10.* Demonstrate the equivalence of the types for reciprocal just presented.

### 3.10 A normal form for type schemes

As far as the theory of units is concerned, the form of type schemes doesn't much matter. But for the programmer, it could be confusing to be presented with seemingly-different schemes that are in fact equivalent. Partly for this reason,  $F\#$  presents type schemes to the programmer in a consistent way, putting them in a *normal form*. This normal form has pleasant properties (e.g. it minimizes the number of quantified and 'free' variables) and corresponds to a well-known form (the Hermite Normal Form) from algebra [7].



## 4 Semantics of units

In his original paper on polymorphic type inference, Milner defined a denotational semantics for a small ML-like language [13]. This semantics incorporated a value `wrong` to correspond to the detection of a failure at run-time, such as applying a non-functional value as the operator of an application. It was then proved that “well-typed programs don’t go wrong”, meaning that if a program passes the type-checker, then no run-time failure occurs.

When working with a small-step operational semantics, as has become popular more recently, one instead proves *syntactic type soundness*, stating that

1. reduction *preserves* types, namely that if  $e : \tau$  and  $e \rightarrow e'$  then  $e' : \tau$ ; and
2. reduction makes *progress*: if  $e$  is not a final value then there always exists some  $e'$  such that  $e \rightarrow e'$ .

Both the idea of `wrong` values and the notion that programs make ‘reductions’ until reaching a final value are convenient fictions that don’t really correspond to the actual execution of real programs. Nevertheless, for a language with, say, just integers and function values, it can be argued that the attempted ‘application’ of an integer really does constitute a run-time failure (and might be manifested as a memory violation), and hence syntactic type soundness is saying something about program safety.

### 4.1 Units going wrong

For units-of-measure, though, this argument makes no sense. What “goes wrong” if a program contains a unit error? If run-time values do not carry their units, as is the case with `F#` and other systems [18], then syntactic type soundness tells us precisely *nothing*. Of course, we could incorporate units into the semantics and ensure that operations with mismatched units evaluate to `wrong` (denotationally), or get stuck (in a small step operational semantics). But in a sense, that’s cheating: we have *instrumented* the semantics and thereby changed it to match some refinement of the type system, when in fact the behaviour of programs with and without unit annotations should be the same.

Instead, let’s use *nature* as our guide – after all, we are talking about units of measure! In nature, physical laws are independent of the units used, i.e. they are invariant under changes to the unit system. This, then, is the real essence of unit correctness: the ability to *change the unit system* without affecting behaviour.

Consider the following ‘correct’ function and sample data that an airline might use as part of its check-in procedure:

```
let checkin(baggage:float<lb>, allowance:float<lb>) =  
    if baggage > allowance then printf "Bags_exceed_limit"  
  
checkin(88.0<lb>, 44.0<lb>)
```

Now suppose that we metricate the program by replacing `lb` with `kg` and converting all constants from pounds to kilograms.

```

let checkin(baggage:float<kg>, allowance:float<kg>) =
  if baggage > allowance then printf "Bags exceed limit"

checkin(40.0<kg>, 20.0<kg>)

```

The *behaviour remains the same*, namely, the passenger is turned away.<sup>4</sup>

Now consider an ‘incorrect’ program that breaks the rules of units, confusing the weight limit with the length limit:

```

let checkin(baggage:float<lb>, allowance:float<cm>) =
  if baggage > allowance then printf "Bags exceed limit"

checkin(88.0<lb>, 55.0<cm>)

```

No run-time errors occur, and the passenger is turned away – but clearly the business logic used to arrive at this conclusion is faulty. Again, let’s metricate:

```

let checkin(baggage:float<kg>, allowance:float<cm>) =
  if baggage > allowance then printf "Bags exceed limit"

checkin(40.0<kg>, 55.0<cm>)

```

This time, after metrication, the passenger can fly!

The fact that this program was *not* invariant under change-of-units revealed an underlying units error. This is the essence of unit correctness. Notice that it is inherently a *relational* property: it does not say anything about the behaviour of a single program, but instead tells us about the relationship between one program and a transformed one.

## 4.2 Polymorphic functions going wrong

Suppose we have a polymorphic function such as

```
val f : float<'u> -> float<'u^2>
```

How do we characterize ‘going wrong’ for such a function? We certainly know it when we see it: if we were told that **f** was implemented by

```
let f (x:float<'u>) = x*x*x
```

then we would rightly be suspicious of its type! But it’s clearly not enough to consider just changes to the base units, as none are used here.

Furthermore, **f** might not even be implemented in F#. Suppose that **f** were implemented in assembly code, or by dedicated hardware. What *test* applied to **f** would refute its type? The answer is: it should be *invariant under scaling* in ‘**u**’, in the following sense: if its argument is scaled by some factor *k* (corresponding to the units ‘**u**’) then its result should scale by *k*<sup>2</sup> (corresponding to the units ‘**u**<sup>2</sup>’). That is:

$$\forall k > 0, f(k*x) = k^2 * f(x)$$

So if we fed **f** the value 2, and it responded with 8, and we then fed it 4, and it responded with 64, we would know that its type ‘lied’.

<sup>4</sup> Depending on airline!

### 4.3 Parametricity

We’ve now seen how the semantics of base units and of unit polymorphism can be explained in terms of invariance under scaling. This is reminiscent of the idea of *representation independence* in programs: that the behaviour of a program can be independent of the representation of its data types. For example, we can choose to represent booleans as a zero/non-zero distinction on integers, or we could use a string **"F"** for false and **"T"** for true. As long as the underlying representation can’t be ‘broken’ by a client of the boolean type, then it should not be possible to observe the difference.

The similarity goes deep, in fact, as one popular characterization of representation independence, namely *relational parametricity*, can be applied very fruitfully to units-of-measure. We first define a map  $\psi$  from unit variables to positive ‘scale factors’, which we call a ‘scaling environment’. This can be extended to full unit *expressions* in an obvious way:

$$\begin{aligned}\psi(1) &= 1 \\ \psi(u * v) &= \psi(u) \cdot \psi(v) \\ \psi(u^{-1}) &= 1/\psi(u)\end{aligned}$$

So if  $\psi = \{\alpha \mapsto 2, \beta \mapsto 3\}$  then  $\psi(\alpha * \beta^2) = 18$ .

We then define a binary relation  $\sim_{\tau}^{\psi}$ , indexed by types, and parameterized by  $\psi$ , whose interpretation is roughly “has the same behaviour when scaled using  $\psi$ ”.

$$\begin{aligned}x \sim_{\text{float}\langle u \rangle}^{\psi} y &\Leftrightarrow y = \psi(u) * x \\ f \sim_{\tau_1 \rightarrow \tau_2}^{\psi} g &\Leftrightarrow \forall xy, x \sim_{\tau_1}^{\psi} y \Rightarrow f(x) \sim_{\tau_2}^{\psi} g(y)\end{aligned}$$

The interpretation of a type scheme  $\forall \bar{\alpha}. \tau$  is then “related for all possible scalings for unit variables  $\bar{\alpha}$ ”. Formally:

$$x \sim_{\forall \bar{\alpha}. \tau} y \Leftrightarrow \forall \bar{k}, x \sim_{\tau}^{\{\bar{\alpha} \mapsto \bar{k}\}} y$$

It’s then possible to prove a ‘fundamental theorem’ that states that an expression with closed type is related to itself at that type, i.e. if  $e : \sigma$ , then  $e \sim_{\sigma} e$ . From this, many consequences follow.

### 4.4 Theorems for free

For first-order types, we immediately get what Wadler describes as ‘theorems for free’ [21], namely theorems that hold simply due to the *type* of an expression, irrespective of the details of the code. For example, if  $\mathbf{f}$  has type  $\forall \alpha \beta. \text{float}\langle \alpha \rangle \rightarrow \text{float}\langle \beta \rangle \rightarrow \text{float}\langle \alpha * \beta^{-1} \rangle$  then we know that for any positive  $k_1$  and  $k_2$ , we have

$$\mathbf{f} (k_1 * x) (k_2 * x) = (k_1/k_2) * \mathbf{f} x y$$

We can even prove such theorems for higher-order types, such as the type of `diff` from Section 2.9:

$$\text{diff } h f x = \frac{k_2}{k_1} * \text{diff} \left( \frac{h}{k_1} \right) \left( \lambda x. \frac{f(x * k_1)}{k_2} \right) \left( \frac{x}{k_1} \right).$$

## 4.5 Zero

We can now explain semantically why zero is polymorphic in its units (i.e. it has type  $\forall\alpha.\text{float}\langle\alpha\rangle$ ) whilst all other values are dimensionless (i.e. they have type  $\text{float}\langle 1\rangle$ ). It's easy: zero is invariant under scaling (because  $k * 0.0 = 0.0$  for any scale factor  $k$ ) whilst other values are not.

## 4.6 Definability

Another well-known application of parametricity is to prove that certain functions cannot be defined with a particular type – and even, to show that there are *no* functions with a particular type, i.e. the type is uninhabited, or at least that there are no *interesting* functions with that type.

*Exercise 11.* In a statically-typed pure functional language with only total functions, what functions can you write with type  $\forall\alpha.\alpha \rightarrow \alpha \rightarrow \alpha$ . What happens if you add general recursion to the language?

*Exercise 12.* Do any functions have the type  $\forall\alpha\beta.\alpha \rightarrow \beta$ ? Can you explain?

For units-of-measure, it's possible to use parametricity to show similar definability results. For example, in a pure functional language with only total functions we can tell that the only function  $f$  with type  $\forall\alpha.\text{float}\langle 1\rangle \rightarrow \text{float}\langle\alpha\rangle$  is the constant zero function, because by invariance under scaling we must have  $f(x) = k * f(x)$  for any  $k$ .

*Exercise 13.* Can you think of any functions with the type  $\forall\alpha\beta.\text{float}\langle\alpha * \beta\rangle \rightarrow \text{float}\langle\alpha\rangle * \text{float}\langle\beta\rangle$ ? Can you think of any *interesting* ones? Why not? Can you give a formal argument based on parametricity?

Most interestingly of all, if the relation  $\sim_T^\psi$  is beefed up a bit, it's possible to show that in a language with only basic arithmetic (i.e.  $+$ ,  $-$ ,  $*$  and  $/$ ), there are no interesting functions with type  $\forall\alpha.\text{float}\langle\alpha^2\rangle \rightarrow \text{float}\langle\alpha\rangle$ . In other words, square root is not definable.

*Exercise 14.* Can you think why? Hint: try using Newton's method from Section 2.9 to construct such a function.

## 4.7 Type isomorphisms

Two types  $\tau$  and  $\tau'$  are said to be *isomorphic*, written  $\tau \cong \tau'$ , if there are functions  $i : \tau \rightarrow \tau'$  and  $j : \tau' \rightarrow \tau$  such that  $j \circ i = \text{id}_\tau$  and  $i \circ j = \text{id}_{\tau'}$  where  $\text{id}_\tau$  and  $\text{id}_{\tau'}$  are respective identity functions on  $\tau$  and  $\tau'$ . We need to be a bit more precise about what we mean by *function* (should it be definable in the programming language?) and  $=$  (is this observational equivalence?). For simple isomorphisms, the distinction doesn't matter. For example, it's obvious that  $\text{int} * \text{bool} \cong \text{bool} * \text{int}$  by the (definable) functions

```

let i (p:int*bool) = (snd p, fst p)
let j (q:bool*int) = (snd q, fst q)

```

*Exercise 15.* Show that the types `int*bool -> unit*int` and `bool*int -> int` are isomorphic.

More subtle isomorphisms make use of parametricity. For example:

$$\begin{aligned} \tau &\cong \forall\alpha.(\alpha \rightarrow \tau) \rightarrow \tau \\ \forall\alpha.(\tau_1 \rightarrow \tau_2 \rightarrow \alpha) \rightarrow \alpha &\cong \tau_1 * \tau_2 \end{aligned}$$

These latter isomorphisms are not definable in ML (or F#), because they require functions that take polymorphic arguments.

*Exercise 16.* For each isomorphism above, write down maps  $i$  and  $j$  that exhibit the isomorphisms, supposing that you had a language with ‘first-class’ polymorphism à la System F.

For units-of-measure, some rather surprising isomorphisms hold. Suppose for a moment that all values of `float` type are positive, and that operations on floats preserve this invariant (this rules out ordinary subtraction, for example). Then the following isomorphism between types

$$\forall\alpha.\text{float}\langle\alpha\rangle \rightarrow \text{float}\langle\alpha\rangle \cong \text{float}\langle 1 \rangle$$

holds, *i.e.* the set of unary polymorphic functions that simply preserve their units is isomorphic to the set of dimensionless values! This isomorphism can be demonstrated by the maps  $i : (\forall\alpha.\text{float}\langle\alpha\rangle \rightarrow \text{float}\langle\alpha\rangle) \rightarrow \text{float}\langle 1 \rangle$  and  $j : \text{float}\langle 1 \rangle \rightarrow (\forall\alpha.\text{float}\langle\alpha\rangle \rightarrow \text{float}\langle\alpha\rangle)$  below:

$$\begin{aligned} i(f) &= f(1.0) \\ j(x) &= \lambda y.y * x \end{aligned}$$

We now prove that these maps compose to give the identity. First,  $i \circ j = id$ , by simple equational reasoning:

$$\begin{aligned} i \circ j & \\ &= \lambda x.i(j(x)) \quad (\text{definition of composition}) \\ &= \lambda x.i(\lambda y.y * x) \quad (\text{applying } j) \\ &= \lambda x.1.0 * x \quad (\text{applying } i) \\ &= \lambda x.x \quad (\text{arithmetic identity}) \end{aligned}$$

Now,  $j \circ i = id$ , by scaling invariance:

$$\begin{aligned} j \circ i & \\ &= \lambda f.j(i(f)) \quad (\text{definition of composition}) \\ &= \lambda f.j(f(1.0)) \quad (\text{applying } i) \\ &= \lambda f.\lambda y.y * f(1.0) \quad (\text{applying } j) \\ &= \lambda f.\lambda y.fy \quad (\text{instance of scaling invariance}) \\ &= \lambda f.f \quad (\text{eta}) \end{aligned}$$

This is a semi-formal argument. Alternatively, consider the inhabitants of the type  $\forall \alpha. \text{float}\langle \alpha \rangle \rightarrow \text{float}\langle \alpha \rangle$ . What can they do with an argument  $x$  of type  $\text{float}\langle \alpha \rangle$ ? They cannot add  $x$  to anything except for zero or  $x$  itself, as there are no other values of type  $\text{float}\langle \alpha \rangle$  available. They can multiply or divide by a dimensionless value, as this results in a value with the appropriate type. But that's it: in fact, the function must be observably equivalent to one having the form  $\lambda x. k * x$  for some  $k$ . In other words, a value  $k$  of type  $\text{float}\langle 1 \rangle$  completely determines the function.

*Exercise 17.* Prove the following isomorphism:

$$\forall \alpha. \text{float}\langle \alpha \rangle \rightarrow \text{float}\langle \alpha \rangle \rightarrow \text{float}\langle \alpha \rangle \cong \text{float}\langle 1 \rangle \rightarrow \text{float}\langle 1 \rangle$$

## 4.8 Dimensional analysis

The invariance of physical laws under changes to the units not only provides a simple check on the correctness of formulae, it also makes it possible to *derive* laws simply through consideration of the units. This is called *dimensional analysis* and its origins go back at least a century.

The idea is simple: when investigating some physical phenomenon, if the equations governing the phenomenon are not known but the parameters are known, one can use the dimensions of the parameters to narrow down the possible form the equations may take. For example, consider investigating the equation which determines the period of oscillation  $t$  of a simple pendulum. Possible parameters are the length of the pendulum  $l$ , the mass  $m$ , the initial angle from the vertical  $\theta$  and the acceleration due to gravity  $g$ . After performing dimensional analysis it is possible to assert that the equation must be of the form  $t = \sqrt{l/g} \phi(\theta)$  for some function  $\phi$  of the angle  $\theta$ . Of course it turns out that for small angles  $\phi(\theta) \approx 2\pi$ , but dimensional analysis got us a long way – in particular, the period of oscillation turned out to be independent of the mass  $m$ . In general, any dimensionally consistent equation over several variables can be reduced to an equation over a smaller number of dimensionless terms which are products of powers of the original variables. This is known as the Pi Theorem [5].

**Theorem 3 (Pi Theorem).** *Fix a set of  $m$  base dimensions and let  $x_1, \dots, x_n$  be positive variables with the dimensions of  $x_i$  given by the  $i$ 'th column of an  $m \times n$  matrix  $A$  of dimension exponents. Then any scale-invariant relation of the form*

$$f(x_1, \dots, x_n) = 0$$

*is equivalent to a relation*

$$f'(\Pi_1, \dots, \Pi_{n-r}) = 0$$

*where  $r$  is the rank of the matrix  $A$  and  $\Pi_1, \dots, \Pi_{n-r}$  are dimensionless power-products of  $x_1, \dots, x_n$ .*

*Proof.* See Birkhoff [5].

As before, we suppose that `float` values are positive, and we can prove the following analogous result concerning type isomorphisms for first-order types in a programming language with units-of-measure.

**Theorem 4 (Pi Theorem for programming).** *Let  $\tau$  be a closed type of the form*

$$\forall \alpha_1, \dots, \alpha_m. \text{float}\langle u_1 \rangle \rightarrow \dots \rightarrow \text{float}\langle u_n \rangle \rightarrow \text{float}\langle u_0 \rangle.$$

*Let  $A$  be the  $m \times n$  matrix of unit variable exponents in  $u_1, \dots, u_n$ , and  $B$  the  $m$ -vector of unit variable exponents in  $u_0$ . If the equation  $AX = B$  is solvable for integer variables in  $X$ , then*

$$\tau \cong \text{float}\langle 1 \rangle \rightarrow \dots \rightarrow \text{float}\langle 1 \rangle \rightarrow \text{float}\langle 1 \rangle$$

where  $r$  is the rank of  $A$ .

*Proof.* See [8].

We can now apply this theorem to the pendulum example, recast in type-theoretic terms. Suppose that the square of the period of a pendulum is determined by a function

$$p : \forall M. \forall L. \forall T. \text{float}\langle M \rangle \rightarrow \text{float}\langle L \rangle \\ \rightarrow \text{float}\langle L * T^{-2} \rangle \rightarrow \text{float}\langle 1 \rangle \rightarrow \text{float}\langle T^2 \rangle$$

whose arguments represent the mass and length of the pendulum, the acceleration due to gravity and the angle of swing. Then for positive argument values  $\text{float}\langle 1 \rangle \rightarrow \text{float}\langle 1 \rangle$  is an isomorphic type.

## References

1. The International System of Units (SI), 2006.
2. E. Allen, D. Chase, J. Hallett, V. Lunchangco, J.-W. Maessen, S. Ryu, G. L. Steele Jr., and S. Tobin-Hochstadt. *The Fortress Language Specification, Version 1.0*, 2008.
3. F. Baader and T. Nipkow. *Term Rewriting and All That*. Cambridge University Press, 1998.
4. R. S. Bird and R. Paterson. de bruijn notation as a nested datatype. *Journal of Functional Programming*, 9(1):77–92, 1999.
5. G. Birkhoff. *Hydrodynamics: A Study in Logic, Fact and Similitude*. Princeton University Press, Revised edition, 1960.
6. B. Buckwalter. *dimensional: statically checked physical dimensions for Haskell*, 2008.
7. A. J. Kennedy. *Programming Languages and Dimensions*. PhD thesis, University of Cambridge, 1995.
8. A. J. Kennedy. Relational parametricity and units of measure. In *POPL '97: Proceedings of the 24th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 442–455. ACM Press, 1997.

9. A. J. Kennedy. Formalizing an extensional semantics for units of measure. In *3rd ACM SIGPLAN Workshop on Mechanizing Metatheory (WMM)*, 2008.
10. D. Le Botlan and D. Rémy. MLF: Raising ML to the power of System F. In *8th ACM SIGPLAN International Conference on Functional Programming (ICFP)*, 2003.
11. D. Leijen. HMF: Simple type inference for first-class polymorphism. In *13th ACM SIGPLAN International Conference on Functional Programming (ICFP)*, 2008.
12. S. Lindley. Many holes in Hindley-Milner. In *Proceedings of the 2008 Workshop on ML*, 2008.
13. R. Milner. A theory of type polymorphism in programming. *Journal of computer and system sciences*, 17:348–375, 1978.
14. NASA. Mars Climate Orbiter Mishap Investigation Board: Phase I Report, November 1999.
15. B. C. Pierce, editor. *Types and Programming Languages*. MIT Press, 2002.
16. B. C. Pierce, editor. *Advanced Topics in Types and Programming Languages*. MIT Press, 2005.
17. W. H. Press, S. A. Teukolsky, W. T. Vetterling, and B. P. Flannery. *Numerical Recipes: The Art of Scientific Computing*. Cambridge University Press, 3rd edition, 2007.
18. M. C. Schabel and S. Watanabe. *Boost.Units*, 2008.
19. T. Schrijvers, S. Peyton Jones, M. Sulzmann, and D. Vytiniotis. Complete and decidable type inference for GADTs. In *International Conference on Functional Programming (ICFP)*, 2009.
20. D. Vytiniotis, S. Weirich, and S. Peyton Jones. FPH: First-class polymorphism for Haskell. In *13th ACM SIGPLAN International Conference on Functional Programming (ICFP)*, 2008.
21. P. Wadler. Theorems for free! In *Proceedings of the 4th International Symposium on Functional Programming Languages and Computer Architecture*, 1989.