A Typechecker Plugin for Units of Measure

Domain-Specific Constraint Solving in GHC Haskell

Adam Gundry

Well-Typed LLP, UK

adam@well-typed.com

Abstract

Typed functional programming and units of measure are a natural combination, as F# ably demonstrates. However, encoding statically-checked units in Haskell's type system leads to inevitable disappointment with the usability of the resulting system. Extending the language itself would produce a much better result, but it would be a lot of work! In this paper, I demonstrate how typechecker plugins in the Glasgow Haskell Compiler allow users to define domain-specific constraint solving behaviour, making it possible to implement units of measure as a type system extension without rebuilding the compiler. This paves the way for a more modular treatment of constraint solving in GHC.

Categories and Subject Descriptors D.3.3 [*Programming Languages*]: Language Constructs and Features; F.3.3 [*Logics and Meanings of Programs*]: Studies of Program Constructs

Keywords Dimensions, type inference, modular typechecking

1. Introduction

Dimensions (such as length and time) and units of measure (such as metres, feet and seconds) are a very useful mechanism to reduce the chances of making a costly error[1], and make it easier to perform calculations. As Kennedy (2010) put it, "Units-of-measure are to science what types are to programming." It is natural, therefore, to consider the extension of typed programming languages with support for units of measure. At a minimum, such support should allow the programmer to declare the units of quantities, and prevent them making errors such as adding incompatible quantities.

There has been much work in this direction, notably by Kennedy (2010) in the context of the F# functional programming language. He has shown that units of measure fit particularly well with Hindley-Milner type inference, leading to a simple but powerful system.

[1] It is traditional here to cite the Mars Climate Orbiter, or the Gimli Glider (`http://lamar.colostate.edu/~hillger/unit-mixups.html`).

For example, in F# one can write

$$\begin{array}{l}
> [<\mathsf{Measure}>]\ \mathbf{type}\ \mathsf{m};;\\
> [<\mathsf{Measure}>]\ \mathbf{type}\ \mathsf{s};;\\
> \mathbf{let}\ \mathit{time} \quad\ \ = 3.0\,\langle \mathsf{s}\rangle;;\\
> \mathbf{let}\ \mathit{speed} \quad\ = 5.0\,\langle \mathsf{m/s}\rangle;;\\
> \mathbf{let}\ \mathit{distance} = \mathit{time} * \mathit{speed};;
\end{array}$$

and the system will correctly infer the units of $\mathit{distance}$:

$$\mathbf{val}\ \mathit{distance} : \mathsf{float}\langle \mathsf{m}\rangle = 15.0$$

In addition, Kennedy's system supports *unit polymorphism*: definitions can be checked abstractly, with the concrete units being determined at the use sites. For example, one can define the function

$$\begin{array}{l}
> \mathbf{let}\ \mathit{sqr}\ (x : \mathsf{float}\langle _\rangle) = x * x;;\\
\mathbf{val}\ \mathit{sqr} : x : \mathsf{float}\langle u\rangle \to \mathsf{float}\langle u\hat{}2\rangle
\end{array}$$

which is polymorphic in a unit variable u. The type annotation on the definition is necessary because overloaded arithmetic operators in F# do not have units by default.

Modern GHC Haskell supports a range of language features (in particular, type families) that make it possible to encode quite complex properties at the type level. Correspondingly, in the Haskell world there have been various attempts to encode units of measure, in particular the robust and expressive `units` library by Muranushi and Eisenberg (2014). This allows one to write[2]

$$\begin{array}{l}
\mathit{time} \quad\ \ = 3.0\ \%\ [\mathrm{si}|\ \mathsf{s}\ |]\\
\mathit{speed} \quad\ = 5.0\ \%\ [\mathrm{si}|\ \mathsf{m/s}\ |]\\
\mathit{distance} = \mathit{time} \circledast \mathit{speed}
\end{array}$$

although the inferred type of $\mathit{distance}$ is not F#'s $\mathsf{float}\langle \mathsf{m}\rangle$ but

$$\mathit{distance} :: \mathsf{Qu}\ [\mathsf{F\ Length\ One}]\ \mathsf{DefaultLCSU\ Double}\,.$$

This work is an impressive demonstration of advanced type-level Haskell programming that provides a very expressive system, but it is inevitably limited by the GHC features available to programmers. The main limitations are the inferior type inference behaviour and error messages produced by units of measure libraries compared to genuine language extensions (as in F#).

How might we go about extending the Haskell *language*, as implemented in GHC, with units of measure? Ideally, we want a modular design that does not unnecessarily bake features into the compiler, and allows their impact on the type system to be understood in isolation. Moreover, the development effort required to extend GHC with a new feature is substantial. It would be much better if we were able to plug in support for units of measure to the typechecker, *without* changing GHC itself.

[2] Provided $[\mathrm{si}|\ \cdot\ |]$ is defined as a quasiquoter for SI units, so for example $[\mathrm{si}|\ \mathsf{m/s}\ |]$ represents metres per second as a unit.

1.1 Type-Level Arithmetic

Additional motivation for being able to extend the typechecker comes from another desirable GHC extension: increasing the automated reasoning available to users of type-level arithmetic. For some time, thanks to the work of Iavor Diatchki[3], it has been possible to use natural number literals and arithmetic operators in types. For example, one can define vectors (lists indexed by their length):

$$
\begin{array}{l}
\mathbf{data}\ \mathsf{Vec}\ a\ (n :: \mathsf{Nat})\ \mathbf{where} \\
\quad \mathsf{Nil} \quad :: \mathsf{Vec}\ a\ 0 \\
\quad \mathsf{Cons} :: a \to \mathsf{Vec}\ a\ n \to \mathsf{Vec}\ a\ (1+n) \\
\mathit{myVec} :: \mathsf{Vec}\ \mathsf{Char}\ 3 \\
\mathit{myVec} = \mathsf{Cons}\ \texttt{'a'}\ (\mathsf{Cons}\ \texttt{'b'}\ (\mathsf{Cons}\ \texttt{'c'}\ \mathsf{Nil})) \\
\mathit{vhead} :: \mathsf{Vec}\ a\ (1+n) \to a \\
\mathit{vhead}\ (\mathsf{Cons}\ x\ \_) = x
\end{array}
$$

However, further progress is stymied by the lack of support for working with numeric *variables*. While *vhead* works,

$$
\begin{array}{l}
\mathit{vtail} :: \mathsf{Vec}\ a\ (1+n) \to \mathsf{Vec}\ a\ n \\
\mathit{vtail}\ (\mathsf{Cons}\ \_\ \mathit{xs}) = \mathit{xs}
\end{array}
$$

is not accepted by GHC 7.8.3, because it does not know that $(1+)$ is an injective function:

```
Could not deduce (n1 ~ n)
from the context ((1 + n) ~ (1 + n1))
  bound by a pattern with constructor Cons
  ... in an equation for ‘vtail’
```

It would be nice if the typechecker was able to prove more equations, using domain-specific knowledge about arithmetic. One possibility is to interface GHC to an SMT solver, so that the SMT solver can solve arithmetic equations left unsolved by GHC (see section 5.3). The `ghc-typelits-natnormalise`[4] typechecker plugin, and the `inch` preprocessor (Gundry 2013), demonstrate alternative approaches based on normalisation.

1.2 Compiler Plugins

Max Bolingbroke and Austin Seipp implemented support for compiler plugins in GHC version 7.2.1[5]. Inspired by a similar concept in the GNU Compiler Collection (GCC), they were originally intended for adding custom optimisations and analyses of GHC's internal Core language (System F_C). The basic idea is that a user package (distributed separately from GHC itself) contains a module M that exports a symbol *plugin* belonging to the type Plugin defined in the GHC API. Users can invoke GHC with an additional argument `-fplugin=M`, whereon the module will be dynamically linked into the running compiler, and invoked during compilation.

Crucially, plugins allow new compiler functionality to be added separately from the main development effort. This makes feature development quicker, as the entire system need not be recompiled when a plugin is changed, and makes it easier for programmers who are not compiler developers to contribute to and use plugins.

[3] `https://ghc.haskell.org/trac/ghc/wiki/TypeNats`

[4] `https://hackage.haskell.org/package/ghc-typelits-natnormalise`

[5] `https://downloads.haskell.org/~ghc/7.2.1/docs/html/users_guide/compiler-plugins.html`

1.3 Summary

In the sequel, I will first describe `uom-plugin`,[6] a Haskell library for units of measure, then explain the typechecker plugins feature that makes it possible. Iavor Diatchki, Eric Seidel and I implemented this feature in GHC 7.10.1.[7] I will specify typechecker plugins in general, and the constraint-solving algorithm used by `uom-plugin` in particular, using the OutsideIn(X) type inference framework.

Concretely, the contributions of this paper are:

- a design for a units of measure library with good type inference properties, showing the need for domain-specific constraint solving behaviour in the typechecker (section 2);
- an explanation of the typechecker plugins interface that enables constraint solver extension, both informally in Haskell and formally by relating it to OutsideIn(X) (section 3); and
- an algorithm for solving constraints in the equational theory of free abelian groups, which satisfies the properties required for sound and most general type inference (section 4).

Section 5 compares the resulting units of measure system to other approaches in F# and Haskell, and section 6 concludes with a discussion of future directions for this work.

2. Units of Measure

First, I will describe how to extend the language with the syntax of units of measure, then will go on to discuss their semantics.

2.1 The Syntax of Units and Quantities

A typical approach to units of measure in programming languages is to annotate numeric types with their units, such as the $\mathsf{int}\langle\cdot\rangle$ and $\mathsf{float}\langle\cdot\rangle$ type constructors in F#. In Haskell, the natural way to do this is through the definition

$$\mathbf{newtype}\ \mathsf{Quantity}\ a\ (u :: \mathsf{Unit}) = \mathsf{MkQuantity}\ a$$

which makes $\mathsf{Quantity}\ a\ u$ use the same runtime representation as the underlying (typically numeric) type a, but tagged with a *phantom type parameter* (Leijen and Meijer 1999) u of kind Unit. This means that using $\mathsf{Quantity}\ a\ u$ has no runtime overhead compared to using plain a, but it can have additional safety guarantees.

The Unit datatype is lifted to the kind level via *datatype promotion* (Yorgey et al. 2012). It has no constructors, but instead is accompanied by the following type-level definitions, implemented as type families without any equations:

$$
\begin{array}{lll}
\mathsf{Base} :: \mathsf{Symbol} \to \mathsf{Unit} & & \\
\mathbb{1} & :: \mathsf{Unit} & \text{-- \texttt{One} in ASCII} \\
(\circledast) & :: \mathsf{Unit} \to \mathsf{Unit} \to \mathsf{Unit} & \text{-- \texttt{*:} in ASCII} \\
(\oslash) & :: \mathsf{Unit} \to \mathsf{Unit} \to \mathsf{Unit} & \text{-- \texttt{/:} in ASCII}
\end{array}
$$

Base creates base units, which are represented as type-level strings (of kind Symbol) for simplicity. Dimensionless quantities are represented with $\mathbb{1}$, and the operators allow more complex units to be formed. Representing them as type families with no equations means they are essentially opaque symbols that may not be partially applied and are not injective; this avoids the equational theory of units conflicting with GHC's built-in equality rules for types.

[6] `https://github.com/adamgundry/uom-plugin`

[7] `https://downloads.haskell.org/~ghc/7.10.1/docs/html/users_guide/compiler-plugins.html#typechecker-plugins`

2.1.1 Constructing Quantities

Crucially, the MkQuantity constructor should not be used by client code, so that users of library work with a unit-safe interface. If the constructor were available, users could write code like this, which would destroy all unit safety guarantees provided by the library:

$$\begin{array}{l} \mathit{unsafeConvertQuantity} :: \mathsf{Quantity}\ a\ u \to \mathsf{Quantity}\ a\ v \\ \mathit{unsafeConvertQuantity}\ (\mathsf{MkQuantity}\ x) = \mathsf{MkQuantity}\ x \end{array}$$

Of course, users need some way to produce and consume quantities, i.e. convert between a and Quantity $a\ u$. It is fine for the library to expose

$$\begin{array}{l} \mathit{unQuantity} :: \mathsf{Quantity}\ a\ u \to a \\ \mathit{unQuantity}\ (\mathsf{MkQuantity}\ x) = x \end{array}$$

but not

$$\mathsf{MkQuantity} :: a \to \mathsf{Quantity}\ a\ u$$

as their composition yields $\mathit{unsafeConvertQuantity}$.

The real problem here is that MkQuantity should only be *monomorphic* (sometimes known as 'weakly polymorphic') in its unit. It is fine for u to be any concrete unit, but it must not be generalised over to become a universally quantified type variable. Such variables are permitted in types in Caml (Garrigue 2004), but not in Haskell.

As a workaround, the library offers a Template Haskell *quasiquoter* [u| · |] that enables the user to write concrete quantities in a convenient syntax, translating them into safe applications of MkQuantity:

$$\begin{array}{lll} \mathit{mass} &= [\mathrm{u}|\ 65\,\mathsf{kg}\ |] & \\ g &= [\mathrm{u}|\ 9.808\,\mathsf{m/s}\hat{\ }2\ |] & \text{-- equivalent to } \mathsf{m}/(\mathsf{s} * \mathsf{s}) \end{array}$$

For example, mass translates into

$$\mathsf{MkQuantity}\ 65 :: \mathsf{Quantity}\ a\ (\mathsf{Base}\ \texttt{"kg"})$$

Bare numeric literals are interpreted as dimensionless constants by the quasiquoter, except for zero, which is polymorphic in its units:

$$\begin{array}{lll} \alpha &= [\mathrm{u}|\ 0.00729735\ |] & :: \mathsf{Quantity\ Double}\ \mathbb{1} \\ \mathit{zero} &= [\mathrm{u}|\ 0\ |] & :: \mathsf{Quantity\ Double}\ u \end{array}$$

Omitting the numeric value yields a specialisation of MkQuantity to the appropriate type, which is useful when units need to be attached to numeric values that are not literal constants, for example:

$$\begin{array}{l} \mathit{readMass} :: \mathsf{IO}\ (\mathsf{Quantity\ Double}\ (\mathsf{Base}\ \texttt{"kg"})) \\ \mathit{readMass} = \mathit{fmap}\ [\mathrm{u}|\ \mathsf{kg}\ |]\ \mathit{readLn} \end{array}$$

2.1.2 Arithmetic Operations on Quantities

The library includes the following (written `+:` and `*:` in ASCII):

$$\begin{array}{l} (\oplus) :: \mathsf{Num}\ a \Rightarrow \\ \qquad \mathsf{Quantity}\ a\ u \to \mathsf{Quantity}\ a\ u \to \mathsf{Quantity}\ a\ u \\ \mathsf{MkQuantity}\ x \oplus \mathsf{MkQuantity}\ y = \mathsf{MkQuantity}\ (x + y) \\ (\circledast) :: \mathsf{Num}\ a \Rightarrow \\ \qquad \mathsf{Quantity}\ a\ u \to \mathsf{Quantity}\ a\ v \to \mathsf{Quantity}\ a\ (u \circledast v) \\ \mathsf{MkQuantity}\ x \circledast \mathsf{MkQuantity}\ y = \mathsf{MkQuantity}\ (x * y) \end{array}$$

The $(\oplus)$ and $(\circledast)$ operators on quantities are analogous to the $(+)$ and $(*)$ operators on numbers, except that the phantom parameter makes sure the units are kept in order. Quantities may be multiplied regardless of their units, but may be added only if the units match.[8]

[8] Unfortunately this means that Quantity $a\ u$ cannot be an instance of the standard Haskell Num typeclass, which bundles addition, subtraction and multiplication together. An instance may be given only for Quantity $a\ \mathbb{1}$.

For example, if we have some values

$$\begin{array}{l} \mathit{mass} \quad :: \mathsf{Quantity\ Double}\ (\mathsf{Base}\ \texttt{"kg"}) \\ \mathit{distance} :: \mathsf{Quantity\ Double}\ (\mathsf{Base}\ \texttt{"m"}) \end{array}$$

then we can we define

$$\begin{array}{l} \mathit{prod} :: \mathsf{Quantity\ Double}\ (\mathsf{Base}\ \texttt{"kg"} \circledast \mathsf{Base}\ \texttt{"m"}) \\ \mathit{prod} = \mathit{mass} \circledast \mathit{distance} \end{array}$$

but attempting to add mass to $\mathit{distance}$ gives a type error:

```
Couldn't match type ‘Base "m"’ with ‘Base "kg"’
Expected type: Quantity Double (Base "m")
  Actual type: Quantity Double (Base "kg")
In the first argument of ‘(+:)’, namely ‘mass’
In the expression: mass +: distance
```

In addition to addition and multiplication, similar definitions are given for other standard numeric operations such as negation, division and square root. Since fractional units are not supported, the type of the latter is

$$\mathit{sqrt} :: \mathsf{Floating}\ a \Rightarrow \mathsf{Quantity}\ a\ (u \circledast u) \to \mathsf{Quantity}\ a\ u$$

The user can define their own numeric primitives by accessing the internal MkQuantity constructor directly. They are then responsible for ensuring unit safety of the resulting code.

2.2 The Equational Theory of Units

Are we done? Not quite. Our definitions so far allow us to write the *syntax* of units of measure, but we have not accounted for their *equational theory*. We would expect quantities with the units Base "kg" $\circledast$ Base "m" and Base "m" $\circledast$ Base "kg" to be interchangeable; unit multiplication should be commutative. But adding $\mathit{mass} \circledast \mathit{distance}$ to $\mathit{distance} \circledast \mathit{mass}$ gives:

```
Couldn't match type ‘'Base "m" *: 'Base "kg"’
               with ‘'Base "kg" *: 'Base "m"’
NB: ‘*:’ is a type function,
    and may not be injective
Expected type:
    Quantity Double ('Base "kg" *: 'Base "m")
  Actual type:
    Quantity Double ('Base "m" *: 'Base "kg")
In the second argument of ‘(+:)’, namely
  ‘(distance *: mass)’
In the expression:
  (mass *: distance) +: (distance *: mass)
```

In addition to the usual GHC Haskell rules for type equality (Sulzmann et al. 2007), we would like additional equations to hold to characterise the operations. As in Kennedy's system in F#, these equations are the standard laws of an abelian group:

$$\begin{array}{lll} \forall\ u\ v\,.\,((u \circledast v) \circledast w) & \sim & (u \circledast (v \circledast w)) \\ \forall\ u\ v\,.\,(u \circledast v) & \sim & (v \circledast u) \\ \forall\ u\ v\,.\,(u \circledast \mathbb{1}) & \sim & u \\ \forall\ u\ v\,.\,(u \circledast (\mathbb{1} \oslash u)) & \sim & \mathbb{1} \end{array}$$

But how can we make them hold? GHC allows new axioms to be introduced using a type family, but type families (like functions) may pattern match only on *constructors*, not other type families, in the interests of checking consistency and termination of constraint solving (Schrijvers et al. 2008). In any case, type families are typically useful only if they define a terminating rewrite system, but associativity and commutativity are hardly going to do so!

2.2.1 An Attempted Solution

This is the point where most Haskell units of measure libraries give up on providing exactly the desired equational theory given above. Instead, a common approach is to write a normalisation function for concrete unit expressions (typically implemented as a type family). Provided one is very careful to talk only about equality of normal forms, not the original syntax of unit expressions, this allows some of the desired behaviour. For example, the `units` package defines addition of quantities with this type:[9]

$$
\begin{aligned}
&(\boxplus) :: (d_1 \mathbin{@\sim} d_2, \mathsf{Num}\ n) \Rightarrow \\
&\quad \mathsf{Qu}\ d_1\ l\ n \to \mathsf{Qu}\ d_2\ l\ n \to \mathsf{Qu}\ d_1\ l\ n
\end{aligned}
$$

The constraint $d_1 \mathbin{@\sim} d_2$ means that d_1 and d_2 should be compared up to equality of normal forms, a weaker condition than $d_1 \sim d_2$, which would require d_1 and d_2 to be equal in GHC's equational theory. This weaker constraint means that GHC can determine that $\mathsf{kg} * \mathsf{m}$ and $\mathsf{m} * \mathsf{kg}$ have the same normal form, and hence quantities with those units may be added.

Unfortunately, this has rather drastic consequences for error messages, because they are expressed in terms of normal forms. In `units`, the erroneous addition gives this:

```
Couldn't match
  type '[F Mass One, F Length (P Zero)]'
  with '[]'
In the expression: mass |+| distance
```

Apart from introducing yet another equivalence relation for the user to understand, and mystifying them with error messages, we lose something important in the shift from $(\oplus)$ to $(\boxplus)$: well-behaved *unit polymorphism*. The normalisation approach breaks down when there are variables or other non-canonical unit expressions present. It cannot conclude that $u \boxast v$ is interchangeable with $v \boxast u$, because it cannot compute the normal form of variables such as u and v. If we are lucky, we may be able to postpone the constraint until we have concrete values for the variables, and hence get away with only some messy types. If not, we may not be able to write the unit-polymorphic program we want.

For example, one would like the following function to be accepted with this type signature (or without it)

$$
\begin{aligned}
&f :: \mathsf{Num}\ a \Rightarrow \\
&\quad \mathsf{Quantity}\ a\ u \to \mathsf{Quantity}\ a\ v \to \mathsf{Quantity}\ a\ (u \circledast v) \\
&f\ x\ y = (x \circledast y) \oplus (y \circledast x)
\end{aligned}
$$

but the `units` approach leads to an inferred type like this, involving several internal type families used to implement normalisation:

$$
\begin{aligned}
&(\mathsf{Num}\ a \\
&, [\,] \sim \mathsf{Normalize}\ (\mathsf{Normalize}\ (d_1 \mathbin{@@+} \mathsf{Reorder}\ d_2\ d_1) \\
&\qquad\qquad\qquad \mathbin{@-} \mathsf{Normalize}\ (d_2 \mathbin{@@+} \mathsf{Reorder}\ d_1\ d_2)) \\
&) \Rightarrow \mathsf{Qu}\ d_1\ l\ a \to \mathsf{Qu}\ d_2\ l\ a \\
&\quad \to \mathsf{Qu}\ (\mathsf{Normalize}\ (d_1 \mathbin{@@+} \mathsf{Reorder}\ d_2\ d_1))\ l\ a
\end{aligned}
$$

The crucial observation of this paper is that we need to introduce support for a *domain-specific equational theory*. In the following section, we will see how this is possible.

[9] Instead of annotating quantities Qu with units directly, the `units` package uses a combination of dimension (d) and local coherent system of units (l); the difference is discussed in section 5.2.

Boxed operators are definitions from `units`, to distinguish them from the circled operators of `uom-plugin`. $\boxplus$ is written `|+|` in ASCII.

3. Domain-Specific Constraint Solving

Haskell type inference is essentially a problem of generating and solving constraints. These may be equalities, which arise from the typing rules (e.g. in the application $f\ x$, the compiler must check that f has a function type with domain equal to the type of x), or typeclass constraints, which arise from uses of overloaded functions. Similarly, standard Hindley-Milner type inference amounts to a constraint generation and solving process in which the solver performs first-order unification (Sulzmann et al. 1999).

GHC uses the OutsideIn(X) algorithm (Vytiniotis et al. 2011) to handle the constraints it generates. This is notionally parametric in the choices of constraint domain X and solving algorithm, and provides domain-independent conditions that the constraint solver must satisfy. However, in practice there is only one choice for the solver: GHC implements the solver for type equality constraints (including type families) and typeclasses also described by Vytiniotis et al. (2011). To permit domain-specific equational theories, this solver must be made user-extensible. The user is not expected to replace the solver entirely, although the capability might be interesting (e.g. to experiment with other algorithms).

In this section, I will describe how such a plugin constraint solver works, first as a practical Haskell program interfacing with GHC, then in the more formal theoretical setting of OutsideIn(X).

3.1 Plugging in to GHC

Once GHC's built-in constraint solver has finished its work, it is left with a set of constraints that it could not solve. The job of a plugin solver is to take this set of wanted constraints and either

- identify impossible constraints that GHC has failed to reject outright, for example $\mathsf{kg} \circledast \mathsf{kg} \sim \mathsf{m}$; or
- solve or further simplify the constraints, perhaps generating others in the process.

When a plugin yields new constraints, the main GHC constraint solver will be re-invoked in case it can make further progress, the plugin will be called again, and so on.

To be more precise, a plugin solver is a Haskell function supplied separately with 'given', 'derived'[10] and 'wanted' constraints:

$$
\begin{aligned}
&\mathit{solve} :: [\mathsf{Ct}] \to [\mathsf{Ct}] \to [\mathsf{Ct}] \to \mathsf{TcPluginM}\ \mathsf{TcPluginResult} \\
&\mathit{solve}\ \mathit{givens}\ \mathit{deriveds}\ \mathit{wanteds} = \ldots
\end{aligned}
$$

Here Ct is GHC's internal type of constraints, TcPluginM is a monad providing effects suitable for plugins, and TcPluginResult captures possible outcomes of constraint solving:

$$
\begin{aligned}
&\mathbf{data}\ \mathsf{TcPluginResult} \\
&\quad = \mathsf{TcPluginOk}\ \{\mathit{solved} :: [(\mathsf{EvTerm}, \mathsf{Ct})], \mathit{new} :: [\mathsf{Ct}]\} \\
&\quad \mid\ \mathsf{TcPluginContradiction}\ \{\mathit{impossible} :: [\mathsf{Ct}]\}
\end{aligned}
$$

The TcPluginOk case includes a list of *solved* constraints along with associated evidence (to be discussed in subsection 3.1.2), and a list of *new* constraints to be processed by the main solver. Note that it is possible for 'given' or 'derived' constraints to be solved, which simply means to drop them from consideration since they provide no useful information (e.g. consider $a \circledast \mathbb{1} \sim a$). The result TcPluginOk [ ] [ ] indicates that no progress was made: no constraints could be solved and no new constraints were generated.

[10] Derived constraints arise during the constraint solving process, e.g. from functional dependencies; they will not be considered in any detail here.

$tcPluginIO$ :: IO $a \to$ TcPluginM a
-- Perform arbitrary IO
$tcPluginTrace$ :: String $\to$ SDoc $\to$ TcPluginM ()
-- Print debug message
$tcLookupGlobal$ :: Name $\to$ TcPluginM TyThing
-- Look up a type or definition in the context
$newFlexiTyVar$ :: Kind $\to$ TcPluginM TcTyVar
-- Create a fresh unification variable

Figure 1. A sample of the TcPluginM interface

The details of the TcPluginM monad interface is not important; a few example type signatures are shown in Figure 1. These include the ability to query the context (e.g. look up the definitions of types), generate fresh variables and perform IO operations. Arbitrary IO is not used in `uom-plugin`, but it is useful in other plugins.

Of course, plugins should be essentially pure, but this is a matter for the plugin implementor. More generally, what does it mean for a plugin to be well-behaved? One would expect it to be:

- *pure*, i.e. producing the same result for the same inputs;
- *order-insensitive*, i.e. regarding the constraint lists passed to the $solve$ function as sets (arguably the types should enforce this!);
- *sound*, i.e. claiming to solve constraints only if they can actually be solved, to be elaborated on in subsection 3.1.2; and
- *most general*, i.e. solving constraints without 'guessing', which I will return to in section 4.3.

3.1.1 Plugin-Aware Constraint Solving

The algorithm GHC uses when solving constraints in the presence of a typechecker plugin is as follows:

1. Run the built-in constraint solver, producing a set of constraints that it could neither solve nor show inconsistent.
2. Call the plugin with the remaining constraints:
   - if it returns TcPluginContradiction, report the impossible constraints and stop;
   - if it returns TcPluginOk with some new constraints, remove the solved constraints from the constraint set, add the new ones, then start again from the beginning;
   - if it returns TcPluginOk with no new constraints, remove the solved constraints from the constraint set and stop.

For example, suppose GHC has arrived at a point in the typechecking process where it has some type family F :: Unit $\to *$, a given constraint $\mathsf{F}\ (\mathsf{m} \oslash \mathsf{s}) \sim ()$, an as-yet unsolved unification variable α, and wanted constraints

$$\mathsf{F}\ \alpha \sim (),$$
$$(\alpha \circledast \mathsf{s}) \sim \mathsf{m},$$

that have already been simplified as far as possible by the built-in constraint solver. The plugin solver can now run and output a new wanted constraint $\alpha \sim \mathsf{m} \oslash \mathsf{s}$, leading to the wanted constraints

$$\mathsf{F}\ \alpha \sim (),$$
$$(\alpha \circledast \mathsf{s}) \sim \mathsf{m},$$
$$\alpha \sim (\mathsf{m} \oslash \mathsf{s}).$$

Now the built-in solver can make further progress, substituting for α and using the given constraint to discharge the first goal, leaving

Typeclasses	$\mathtt{D}$		
Data types	$\mathtt{T}$		
Type families	$\mathtt{F}, \mathtt{G}$		
Rigid variables	a, b, c		
Unification variables	α, β, γ		
Type variables	x, y, z	$::=$	$a \mid \alpha$
Types	τ	$::=$	$x \mid \mathtt{T} \mid \mathtt{F}\,\overline{\tau} \mid \tau_1\,\tau_2 \mid \ldots$
Constraints	Q	$::=$	$\epsilon \mid Q_1 \wedge Q_2 \mid \tau_1 \sim \tau_2 \mid \mathtt{D}\,\overline{\tau}$
Substitutions	θ, ϕ	$::=$	$[\overline{x \mapsto \tau}]$

Top-level axiom schemes
$$\mathcal{Q} ::= Q \mid \mathcal{Q}_1 \wedge \mathcal{Q}_2 \mid \forall \overline{a}\,.\,Q \Rightarrow \mathtt{D}\,\overline{\tau} \mid \forall \overline{a}\,.\,\mathtt{F}\,\overline{\tau} \sim \tau$$

Figure 2. Syntax of OutsideIn(X) types and constraints

$$(\mathsf{m} \oslash \mathsf{s}) \circledast \mathsf{s} \sim \mathsf{m},$$

which can be solved directly by another run of the plugin solver. Note that even this simple example involved two runs of the built-in solver and two runs of the plugin; while that could be avoided in this case if the plugin performed substitution and type family reduction itself, in general we would not want plugins to have to reimplement GHC's entire solver!

3.1.2 Evidence of Soundness

If a plugin claims to have solved a constraint, why should we believe it? It would be very easy to produce a plugin that erroneously[11] reported constraints as solved when in fact they were not, potentially introducing type unsoundness and causing runtime crashes. GHC already has a mechanism for detecting such errors: it does not merely typecheck code, but *elaborates* it into System $\mathrm{F_C}$ (Sulzmann et al. 2007), a very explicit core calculus that includes easily-checked evidence for type equality. This does not prevent all compiler bugs, but it does make constraint solver misbehaviour easier to detect.

Thus the actual implementation of plugins demands evidence for each constraint that the plugin claims to have solved. Some plugins may not be able to generate bona fide evidence, in which case they may use the equivalent of $unsafeCoerce$ and assert a constraint without proof. On the other hand, the author of a plugin may create their own axioms and build genuine evidence from them, in which case they can be sure of the type soundness of the resulting system (provided the axioms they introduce are consistent, of course!).

In the implementation, the type EvTerm returned with a constraint in a TcPluginOk result represents terms in the evidence language. Forms of evidence include variables, axioms, typeclass dictionaries and a variety of deduction rules for equality proofs. I will not consider evidence further here, but it is discussed in more detail by Vytiniotis et al. (2012).

3.2 Plugging in to OutsideIn(X)

Having seen how the plugin mechanism works in practice, let us step back and consider the theory justifying it. The OutsideIn(X) framework expects a constraint solver which takes four inputs (with the syntax given in Figure 2):

- user-defined top-level axiom schemes $\mathcal{Q}$ (e.g. from typeclass and type family instances);
- 'given' constraints Q_{given} known to be true locally (e.g. from type signatures or GADT pattern matches);

[11] Or maliciously, though plugins are assumed to be trusted: they can run arbitrary IO actions from within the typechecker, which is dangerous!

$$\begin{array}{lr}
Q \in \mathcal{Q} \text{ implies } \mathcal{Q} \Vdash Q & \text{(R1)} \\
\mathcal{Q} \Vdash Q_1 \text{ and } \mathcal{Q} \wedge Q_1 \Vdash Q_2 \text{ implies } \mathcal{Q} \Vdash Q_2 & \text{(R2)} \\
\mathcal{Q} \Vdash Q \text{ implies } \theta\,\mathcal{Q} \Vdash \theta\, Q & \text{(R3)} \\
\mathcal{Q} \Vdash \tau \sim \tau & \text{(R4)} \\
\mathcal{Q} \Vdash \tau_1 \sim \tau_2 \text{ implies } \mathcal{Q} \Vdash \tau_2 \sim \tau_1 & \text{(R5)} \\
\mathcal{Q} \Vdash \tau_1 \sim \tau_2 \text{ and } \mathcal{Q} \Vdash \tau_2 \sim \tau_3 \text{ implies } \mathcal{Q} \Vdash \tau_1 \sim \tau_3 & \text{(R6)} \\
\mathcal{Q} \Vdash Q_1 \text{ and } \mathcal{Q} \Vdash Q_2 \text{ implies } \mathcal{Q} \Vdash Q_1 \wedge Q_2 & \text{(R7)} \\
\mathcal{Q} \Vdash \tau_1 \sim \tau_2 \text{ implies } \mathcal{Q} \Vdash [a \mapsto \tau_1]\,\tau \sim [a \mapsto \tau_2]\,\tau & \text{(R8)}
\end{array}$$

Figure 3. Properties of entailment[13]

- 'touchable' unification variables $\overline{\alpha}_{\mathrm{tch}}$ (those for which the algorithm is allowed to solve); and
- 'wanted' constraints Q_{wanted} for which solutions are to be found.

In response, the constraint solver must produce two outputs:

- a substitution θ for the touchable variables $\overline{\alpha}_{\mathrm{tch}}$; and
- residual constraints Q_{residual} that could not be solved (but may have been simplified).

The behaviour of the constraint solver is described by the judgment

$$\mathcal{Q}\,;\, Q_{\mathrm{given}}\,;\, \overline{\alpha}_{\mathrm{tch}} \ \vdash\!\blacktriangleright^{\mathrm{simp}} Q_{\mathrm{wanted}} \ \rightsquigarrow\ Q_{\mathrm{residual}}\,;\,\theta$$

which must satisfy certain conditions in order for OutsideIn(X) type inference to be sound and most general:[12]

$$\begin{array}{ll}
\text{(Soundness)} & \mathcal{Q} \wedge Q_{\mathrm{given}} \wedge Q_{\mathrm{residual}} \Vdash \theta\, Q_{\mathrm{wanted}} \\
\text{(Principality)} & \mathcal{Q} \wedge Q_{\mathrm{given}} \wedge Q_{\mathrm{wanted}} \Vdash Q_{\mathrm{residual}} \wedge \mathcal{E}_\theta \\
 & \quad \text{where } \mathcal{E}_\theta = \{x \sim u \mid [x \mapsto u] \in \theta\}
\end{array}$$

That is, the constraint solver must deliver a solution that is sound, i.e. the residual constraints solve the original problem, and most general, i.e. the simplifier has not 'guessed' any values for variables or invented constraints not entailed by the original problem. Here $\mathcal{Q} \Vdash Q$ is the constraint entailment relation, part of the X parameter of OutsideIn(X), which satisfies the properties given in Figure 3.

3.2.1 Defining a Plugin Constraint Solver

A plugin constraint solver can be simpler than the description in the OutsideIn(X) framework, since it need not stand alone but will be combined with the built-in solver. In particular, the plugin need not deal with producing a substitution for unification variables directly. Instead, it may simply add constraints that define variables.

Suppose we have a judgment form

$$\mathcal{Q}\,;\, Q_{\mathrm{given}}\,;\, \overline{\alpha}_{\mathrm{tch}} \ \vdash\!\blacktriangleright^{\mathrm{p}} Q_r \ \rightsquigarrow\ Q_s$$

meaning that the constraints Q_r can be simplified to Q_s under the given assumptions. Figure 4 shows how such a judgment can be combined with the built-in solver judgment $\vdash\!\blacktriangleright^{\mathrm{simp}}$ to produce $\vdash\!\blacktriangleright^{\mathrm{psimp}}$, which conforms to the OutsideIn(X) interface.[14]

The basic idea is that of the implementation, discussed in subsection 3.1.1: run the main constraint solver once, then pass the residual constraints to the plugin. If the plugin generates new constraints (i.e. $Q_s \nsubseteq Q_r$), the Go rule applies and invokes the combined solver judgment again. If not (i.e. $Q_s \subseteq Q_r$), the Stop rule will simply return the remaining constraints.

$$\textsc{Go}\quad \frac{\begin{array}{c}\mathcal{Q}\,;\,Q_g\,;\,\overline{\alpha_0} \ \vdash\!\blacktriangleright^{\mathrm{simp}} Q_w \ \rightsquigarrow\ Q_r\,;\,\theta_0 \\ \mathcal{Q}\,;\,Q_g\,;\,\overline{\alpha_1} \ \vdash\!\blacktriangleright^{\mathrm{p}} Q_r \ \rightsquigarrow\ Q_s \qquad Q_s \nsubseteq Q_r \\ \mathcal{Q}\,;\,Q_g\,;\,\overline{\alpha_2} \ \vdash\!\blacktriangleright^{\mathrm{psimp}} Q_s \ \rightsquigarrow\ Q_t\,;\,\theta_1\end{array}}{\mathcal{Q}\,;\,Q_g\,;\,\overline{\alpha_0} \ \vdash\!\blacktriangleright^{\mathrm{psimp}} Q_w \ \rightsquigarrow\ Q_t\,;\,\theta_0 \circ \theta_1|_{\overline{\alpha_0}}}$$

$$\textsc{Stop}\quad \frac{\begin{array}{c}\mathcal{Q}\,;\,Q_g\,;\,\overline{\alpha_0} \ \vdash\!\blacktriangleright^{\mathrm{simp}} Q_w \ \rightsquigarrow\ Q_r\,;\,\theta \\ \mathcal{Q}\,;\,Q_g\,;\,\overline{\alpha_1} \ \vdash\!\blacktriangleright^{\mathrm{p}} Q_r \ \rightsquigarrow\ Q_s \qquad Q_s \subseteq Q_r\end{array}}{\mathcal{Q}\,;\,Q_g\,;\,\overline{\alpha_0} \ \vdash\!\blacktriangleright^{\mathrm{psimp}} Q_w \ \rightsquigarrow\ Q_s\,;\,\theta}$$

Figure 4. Plugin-extended OutsideIn(X) solver

Note that this process can be iterated, starting with the basic solver and extend it with multiple plugins.

The combined judgment $\vdash\!\blacktriangleright^{\mathrm{psimp}}$ will satisfy the OutsideIn(X) conditions on the assumption that $\vdash\!\blacktriangleright^{\mathrm{simp}}$ satisfies them, and provided that $\vdash\!\blacktriangleright^{\mathrm{p}}$ satisfies the conditions

$$\begin{array}{ll}
\text{(Plugin soundness)} & \mathcal{Q} \wedge Q_{\mathrm{given}} \wedge Q_s \Vdash Q_r \\
\text{(Plugin principality)} & \mathcal{Q} \wedge Q_{\mathrm{given}} \wedge Q_r \Vdash Q_s
\end{array}$$

i.e. Q_r and Q_s should be equivalent under the given constraints. In section 4.3 I will show that the units of measure plugin I am about to describe satisfies the soundness condition as-is, but satisfies only a weakened form of the principality condition.

4. Units of Measure as a Typechecker Plugin

Having seen the general structure of typechecker plugins, let us consider a specific example. The `uom-plugin` constraint solver is designed to deal with equality constraints between types of kind `Unit`. Essentially it performs equational unification for the theory of free abelian groups. Recalling the earlier example, GHC's built-in constraint solver might have been left with the unsolved constraint

$$\mathsf{Base}\ \texttt{"m"} \circledast \mathsf{Base}\ \texttt{"kg"} \sim \mathsf{Base}\ \texttt{"kg"} \circledast \mathsf{Base}\ \texttt{"m"}$$

but it is easy to see that this constraint is trivial simply by normalisation up to the group axioms.

For constraints involving unification variables, Kennedy (1996, 2010) describes an algorithm for AG-unification that proceeds by a variant of Gaussian elimination, and shows how to extend this to types containing units of measure. For example, given the constraint

$$\alpha \circledast \alpha \sim \beta \circledast \beta \circledast \beta$$

the most general solution is

$$\alpha \sim \gamma \circledast \gamma \circledast \gamma, \beta \sim \gamma \circledast \gamma$$

for some fresh unification variable γ. Since AG-unification is decidable and possesses most general unifiers, type inference in an ML-like setting is well-behaved, though the let-generalisation step is slightly subtle (Gundry 2013).[15]

The situation is slightly more complex in the case of the full GHC Haskell type system, in particular because of the possible presence of

[12] There are also some technical conditions on the domain of the substitution, which require that it substitutes only for touchable variables not occurring in the given or residual constraints.

[13] Slightly reformulated from Vytiniotis et al. (2011)

[14] The details of how to calculate the sets of touchable variables α_1 and α_2 are omitted; it is straightforward but messy to add newly generated unification variables and remove those that have been substituted away.

[15] GHC no longer generalises let-bindings by default in the presence of type families or GADTs, for essentially the same reason.

Unit constraints	U	$::=$	$\epsilon \mid U_1 \wedge U_2 \mid u_1 \sim u_2$
Unit normal forms	u	$::=$	$\mathbb{1} \mid r_1^{i_1} \cdots r_n^{i_n}$
Atoms	r	$::=$	$x \mid \mathsf{b} \mid \mathtt{F}(\overline{\tau})$
Base units	b	$::=$	$\mathsf{kg} \mid \mathsf{m} \mid \ldots$

Figure 5. Syntax of unit constraints

universally quantified variables, type families and local constraints. Thus the plugin constraint solver may encounter constraints like

$$a \circledast a \sim b \circledast b \circledast b$$

where a and b are universally quantified variables, or

$$\mathsf{F}\ a \circledast \mathsf{F}\ b \sim \mathsf{F}\ b \circledast \mathsf{F}\ a$$

where F is a user-defined type family. Moreover, it has to deal with constraints that are 'given' as well as 'wanted', so it must simplify hypotheses as well as solving goals.

The essence of the plugin's constraint solving algorithm is to

1. identify unsolved equality constraints between units;
2. normalise both sides of each constraint up to the group axioms;
3. incrementally simplify given constraints by rewriting them to simpler, equivalent constraints;
4. incrementally simplify wanted constraints, making use of the information from simplifying the givens.

For example, the wanted constraint $\alpha \circledast \alpha \sim (\beta \circledast \beta) \circledast \beta$ equates two types of kind Unit, which normalise to give $\alpha^2 \sim \beta^3$. Once normalised, the constraint $\alpha^2 \sim \beta^3$ can first be rewritten to $\alpha^2 \cdot \beta^{-3} \sim \mathbb{1}$. This can be simplified by substituting by $\alpha \sim \gamma \cdot \beta$ where γ is fresh, leading to $\gamma^2 \cdot \beta^{-1} \sim \mathbb{1}$. Rearranging this gives $\beta \sim \gamma^2$. Hence the solution is $\alpha \sim \gamma^3 \ \wedge\ \beta \sim \gamma^2$.

Normal forms will be written in mathematical notation, as shown in Figure 5, to contrast them with Haskell type expressions. A unit normal form u is a product of distinct atoms r with nonzero integer exponents. $\mathbb{1}$ represents the empty product. If two types of kind Unit are equal under the group axioms in Figure 6, then they will have the same normal form (e.g. $x \circledast \mathbb{1}$ and $(y \circledast x) \oslash y$ both denote the same normal form x^1). I use U for constraints Q that include only equations between units.

The presence of universally quantified variables or type families means that some constraints may not be solved immediately, but they may become feasible once other information has become available. This motivates a *dynamic* unification algorithm: one that makes progress on some constraints in the hope that others may become easier to solve. Since each step replaces a constraint with an equivalent constraint (up to the equational theory), it is most general, and so we can apply simplification steps in any order.

4.1 The Constraint Solving Algorithm

For units of measure unification, the algorithm is given by the rules in Figure 7, which define the relation $U_0 \mapsto_{\overline{z}} U_1\ (\theta, \phi)$. This explains how to rewrite a unit constraint U_0 into a new constraint U_1 and a pair of substitutions (θ, ϕ) that express the equivalence of the original and simplified constraints, modulo the group laws. The list of type variables $\overline{z}$ records those that may not be modified during unification, including both rigid and unification variables.[16]

[16] A *rigid* variable is one that arises from universal quantification. Unification solves for *flexible* unification variables, but may not choose values for rigid ones. Parameterising the rules by forbidden rather than touchable variables is a notational shortcut, to save changing the set when adding a fresh variable.

ASSOCIATIVE
$$\frac{}{\mathcal{Q} \Vdash u_1 \circledast (u_2 \circledast u_3) \sim (u_1 \circledast u_2) \circledast u_3}$$

IDENTITY
$$\frac{}{\mathcal{Q} \Vdash u \circledast \mathbb{1} \sim u}$$

COMMUTATIVE
$$\frac{}{\mathcal{Q} \Vdash u_1 \circledast u_2 \sim u_2 \circledast u_1}$$

INVERSE
$$\frac{}{\mathcal{Q} \Vdash u \circledast (\mathbb{1} \oslash u) \sim \mathbb{1}}$$

TORSION-FREE
$$\frac{\mathcal{Q} \Vdash u \circledast u \circledast \cdots \circledast u \sim \mathbb{1}}{\mathcal{Q} \Vdash u \sim \mathbb{1}}$$

Figure 6. Constraint entailment rules for units

In the interests of simplicity, failure is not represented explicitly here, although in practice it is useful to identify obviously impossible constraints (such as $\mathsf{kg} \sim \mathsf{m}$), and the implementation does this using the TcPluginContradiction result (see section 3.1).

Rule (1) simply ensures that all unit equations are in the form $u \sim \mathbb{1}$. Rule (2) solves trivial equations; since unit normal forms are being considered up to the abelian group laws, this includes cases such as $\alpha \cdot \alpha^{-1} \sim \mathbb{1}$.

Rule (3) is the first to produce an output substitution, in the case where some variable can be instantiated to solve the equation. For example, $\mathsf{m}^4 \cdot \alpha^2 \sim \mathbb{1}$ is solved by substituting $[\alpha \mapsto \mathsf{m}^2]$. Of course, the variable must not belong to the list of fixed variables $\overline{z}$. Again this rule is interpreted up to the group laws.

The most complex rule is (4), which shows how progress can be made in cases where rule (3) does not apply and so the equation cannot immediately be solved. It relies on the fact that any unit can be expressed as a product of distinct atoms $r_1^{i_1} \cdots r_n^{i_n}$. By replacing x with a fresh variable y multiplied by a suitably-chosen unit v, the exponents of the atoms can be reduced. Note that y should be a rigid variable iff x is rigid. For example, this rule introduces a fresh variable c to simplify $a^2 \cdot b^{-3} \sim \mathbb{1}$ to $c^2 \cdot b \sim \mathbb{1}$ with $\theta = [a \mapsto c \cdot b^2]$, $\phi = [c \mapsto a \cdot b^{-2}]$.

Rule (5) says that a conjunction of constraints can be simplified by simplifying one and applying the resulting substitution to the other. Just as units are considered up to the abelian group laws, conjunctions should be treated as sets, so this rule allows any constraint to be simplified.

The rules can be iterated in the obvious way to define a relation $U \mapsto^*_{\overline{z}} U'\ (\theta, \phi)$ that makes multiple simplification steps, composing the resulting substitutions.

4.2 Instantiating the OutsideIn(X) Plugin Framework

Recall that a plugin must supply a judgement

$$\mathcal{Q}\,;\, Q_g\,;\, \overline{\alpha}_{\mathrm{tch}} \ \Vdash\!\blacktriangleright^{\mathrm{p}} Q_r \ \rightsquigarrow\ Q_s$$

that explains how the given constraints Q_g and wanted constraints Q_r are simplified to produce the residual constraints Q_s. This judgment is defined by

$$\frac{U_g \mapsto^*_{\cdot} U'_g\ (\theta_g, \phi_g) \qquad \theta_g\, U_w \mapsto^*_{\overline{z}} U'_w\ (\theta_w, \phi_w) \\ \overline{z} = \mathrm{fv}(\theta_g U_w) \cup \mathrm{fuv}(\theta_g U_w) \setminus \overline{\alpha}_{\mathrm{tch}}}{\mathcal{Q}\,;\, Q_g \wedge U_g\,;\, \overline{\alpha}_{\mathrm{tch}} \ \Vdash\!\blacktriangleright^{\mathrm{p}} Q_w \wedge U_w \ \rightsquigarrow\ Q_w \wedge \phi_g\,(U'_w \wedge \mathcal{E}_{\theta_w})}$$

where Q_g and Q_w are the non-unit given and wanted constraints, respectively, and $\mathcal{E}_\theta = \{x \sim u \mid [x \mapsto u] \in \theta\}$ is the constraint form of a substitution θ.

$u \sim v$	$\mapsto_{\overline{z}}$	$u \cdot v^{-1} \sim \mathbb{1}$	$(\cdot,\cdot)$	if $v \not\equiv \mathbb{1}$	(1)
$\mathbb{1} \sim \mathbb{1}$	$\mapsto_{\overline{z}}$	ϵ	$(\cdot,\cdot)$		(2)
$x^k \cdot u^k \sim \mathbb{1}$	$\mapsto_{\overline{z}}$	ϵ	$([x \mapsto u^{-1}], \cdot)$	if $x \notin \overline{z}$	(3)
$x^k \cdot r_1^{i_1} \cdots r_n^{i_n} \sim \mathbb{1}$	$\mapsto_{\overline{z}}$	$y^k \cdot r_1^{i_1 \bmod k} \cdots r_n^{i_n \bmod k} \sim \mathbb{1}$	$([x \mapsto y \cdot v], [y \mapsto x \cdot v^{-1}])$	if $x \notin \overline{z}$, $\exists j. \lvert k\rvert \leqslant \lvert i_j\rvert$, y fresh, $v = r_1^{-\lfloor i_1/k \rfloor} \cdots r_n^{-\lfloor i_n/k \rfloor}$	(4)
$U_0 \wedge U_1$	$\mapsto_{\overline{z}}$	$U_0' \wedge \theta U_1$	(θ, ϕ)	if $U_0 \mapsto_{\overline{z}} U_0'\ (\theta,\phi)$	(5)

Figure 7. Plugin constraint-solving algorithm

This rule assumes without loss of generality that the unit constraints U_g and U_w are already in normal form; this is justified since every type of kind Unit is provably equal to its normal form according to the entailment relation.

First, the given unit constraints U_g are rewritten according to the simplification rules in Figure 7 until no more rules apply. This produces a substitution θ_g that may eliminate some rigid variables, possibly generating some fresh rigid variables in the process, but with a substitution ϕ_g that relates them back to the original variables. Here $\overline{z}$ is empty because rigid variables may be simplified using the given constraints; they will contain no unification variables. The simplified givens U_g' are discarded.

Next, the substitution θ_g is applied to the wanted unit constraints U_w (in order to eliminate rigid variables if possible), then they are rewritten according to the algorithm, producing a simplified set of constraints U_w' and a substitution θ_w. At this point, only the 'touchable' unification variables may be instantiated, so $\overline{z}$ contains all the free variables that are not listed in $\overline{\alpha}_{\mathrm{tch}}$.

Finally, the residual constraints returned by the rule consist of the unchanged non-unit wanteds Q_w, the simplified unit wanteds U_w' and the constraint form of the substitution θ_w. The substitution ϕ_g, which eliminates any fresh rigid variables introduced when simplifying the unit givens, is applied where necessary.

For example, suppose we have

$$U_g = \{a^2 \sim b^3\}, \quad U_w = \{\gamma^3 \sim a\}, \quad \overline{\alpha}_{\mathrm{tch}} = \{\gamma\}$$

where a and b are rigid variables and γ is a unification variable. Rewriting the given constraint generates a fresh rigid variable c and produces $\theta_g = [a \mapsto c^{-3}, b \mapsto c^{-2}]$, $\phi_g = [c \mapsto a \cdot b^{-2}]$. Applying θ_g leaves us with the wanted constraint $\gamma^3 \sim c^{-3}$, which is easily solved by $\theta_w = [\gamma \mapsto c^{-1}]$. In order to eliminate the variable c introduced by simplifying the given constraint, we apply ϕ_g, so we end up with the solution $\phi_g \theta_w = [\gamma \mapsto a^{-1} \cdot b^2]$.

4.3 Soundness and Generality

As discussed in section 3.2, OutsideIn(X) type inference is sound (i.e. it infers correct types for terms) and delivers principal types (i.e. any type that can be given to the term is an instance of the inferred type), under certain assumptions on the behaviour of the simplifier. These assumptions lead to conditions that the algorithm described above must satisfy.

The conditions are formulated in terms of the $\Vdash$ relation, which satisfies the properties in Figure 3. To justify the soundness and generality of the plugin, additional inference rules are required stating that Unit is an abelian group, as shown in Figure 6. The ASSOCIATIVE, IDENTITY, COMMUTATIVE and INVERSE rules are the usual abelian group laws; the role of TORSION-FREE (characterising *free* abelian groups) will be discussed later.

Type safety depends on the fact that the $\Vdash$ relation is consistent (i.e. it cannot prove that two observably distinct types are equal). Consistency is not threatened by the group laws, because they refer only to type families without equations.[17] If $\circledast$ was a constructor rather than a type family, however, it would be possible to derive a contradiction.

In the following, I assume that the constraint entailment relation $\Vdash$ satisfies the conditions in Figure 3 and this additional condition:

$$\text{Suppose } \mathcal{Q} \Vdash \mathcal{E}_\theta. \text{ Then } \mathcal{Q} \Vdash Q \text{ iff } \mathcal{Q} \Vdash \theta Q \text{ and } \mathcal{Q} \wedge Q \Vdash Q' \text{ iff } \mathcal{Q} \wedge \theta Q \Vdash Q'. \qquad \text{(P)}$$

This says that equalities are substitutive: if the equality constraints $\mathcal{E}_\theta = \{x \sim u \mid [x \mapsto u] \in \theta\}$ hold, then applying the substitution θ does not change the truth of a proposition. This should be the case for any reasonable entailment relation, in particular the concrete entailment relation used by Vytiniotis et al. (2011).

The basic result about the rewrite system needed to show that solutions are both sound and most general is the following, which amounts to showing that rewriting produces equivalent constraints, assuming the substitutions hold as a equations as appropriate. I write $Q_0 \leftrightarrow Q_1$ to mean that the constraints Q_0 and Q_1 are equivalent in the sense that $Q_0 \Vdash Q_1$ and $Q_1 \Vdash Q_0$.

Lemma 1 (Soundness and generality of unification steps)**.** *If* $U_0 \mapsto_{\overline{z}} U_1\ (\theta, \phi)$ *then* $U_1 \wedge \mathcal{E}_\theta \leftrightarrow U_0 \wedge \mathcal{E}_\phi$.

Proof. By induction on the definition of the $\mapsto$ relation.

For rule (1), we need to show $u \cdot v^{-1} \sim \mathbb{1} \leftrightarrow u \sim v$, which follows straightforwardly from the group axioms. Similarly, rule (2) is trivial.

For rule (3), the interesting part is showing $x^k \cdot u^k \sim \mathbb{1} \Vdash x \sim u^{-1}$. The TORSION-FREE rule means that $(x \cdot u)^k \sim \mathbb{1}$ implies $x \cdot u \sim \mathbb{1}$.

For rule (4), we must show that

$$y^k \cdot w \sim \mathbb{1} \wedge x \sim y \cdot v \quad\leftrightarrow\quad x^k \cdot r_1^{i_1} \cdots r_n^{i_n} \sim \mathbb{1} \wedge y \sim x \cdot v^{-1}$$

where $v = r_1^{-\lfloor i_1/k \rfloor} \cdots r_n^{-\lfloor i_n/k \rfloor}$ and $w = r_1^{i_1 \bmod k} \cdots r_n^{i_n \bmod k}$, which follows from the fact that $r_1^{i_1} \cdots r_n^{i_n} \sim v^{-k} \cdot w$.

For rule (5), we must show $U_0' \wedge \theta\, U_1 \wedge \mathcal{E}_\theta \leftrightarrow U_0 \wedge U_1 \wedge \mathcal{E}_\phi$. By induction we have $U_0' \wedge \mathcal{E}_\theta \leftrightarrow U_0 \wedge \mathcal{E}_\phi$, and property (P) gives $\theta\, U_1 \wedge \mathcal{E}_\theta \leftrightarrow U_1$. □

In addition, the following lemma shows the relationship between the two substitutions θ and ϕ produced by the algorithm: applying ϕ to θ yields equations that follow from the input constraints U_0.

Lemma 2. *If* $U_0 \mapsto_{\overline{z}} U_1\ (\theta, \phi)$ *then* $U_0 \Vdash \mathcal{E}_{\phi \circ \theta}$.

[17] GHC 7.10 does not make it possible to enforce that a type family has no equations, but the next release will support empty closed type families. Additionally, this relies on the assumption that all user-defined type families at kind Unit are well-defined (terminating).

Proof. By induction on the definition of the $\mapsto$ relation. The only rules that extend the substitution θ are (4), for which $x \sim u^{-1}$ follows by TORSION-FREE, and (5), for which the composition is $[y \mapsto x \cdot v^{-1}] \circ [x \mapsto y \cdot v] = [x \mapsto (x \cdot v^{-1}) \cdot v]$, the identity up to the group axioms. □

From these results, which extend inductively in the obvious way to multiple reduction steps, it follows that the constraint solver is sound in the sense required by OutsideIn(X).

Theorem 1 (Soundness)**.** *If* $\mathcal{Q}\,;\,Q_1\,;\,\overline{\alpha}_{\mathrm{tch}} \;\vdash\!\!\blacktriangleright^{\mathrm{p}} Q_2 \;\rightsquigarrow\; Q_3$ *then* $\mathcal{Q} \wedge Q_1 \wedge Q_3 \Vdash Q_2$.

Proof. Recall from section 4.2 that we define $\vdash\!\!\blacktriangleright^{\mathrm{p}}$ by

$$\frac{\begin{array}{c} U_g \mapsto^*_{\cdot} U'_g\ (\theta_g, \phi_g) \qquad \theta_g\, U_w \mapsto^*_{\overline{z}} U'_w\ (\theta_w, \phi_w) \\ \overline{z} = \mathrm{fv}(\theta_g U_w) \cup \mathrm{fuv}(\theta_g U_w) \setminus \overline{\alpha}_{\mathrm{tch}} \end{array}}{\mathcal{Q}\,;\,Q_g \wedge U_g\,;\,\overline{\alpha}_{\mathrm{tch}} \;\vdash\!\!\blacktriangleright^{\mathrm{p}} Q_w \wedge U_w \;\rightsquigarrow\; Q_w \wedge \phi_g\,(U'_w \wedge \mathcal{E}_{\theta_w})}$$

hence we have

$$Q_1 = Q_g \wedge U_g,\; Q_2 = Q_w \wedge U_w,\; Q_3 = Q_w \wedge \phi_g\,(U'_w \wedge \mathcal{E}_{\theta_w})$$

and we need to show that

$$\mathcal{Q} \wedge (Q_g \wedge U_g) \wedge (Q_w \wedge \phi_g\,(U'_w \wedge \mathcal{E}_{\theta_w})) \Vdash Q_w \wedge U_w.$$

Since $Q_w \Vdash Q_w$ it is sufficient to show

$$U_g \wedge \phi_g\,(U'_w \wedge \mathcal{E}_{\theta_w}) \Vdash U_w.$$

We are justified in reasoning up to unit normal forms since if u and v are equivalent normal forms then $\epsilon \Vdash u \sim v$ so (R8) gives $\epsilon \Vdash [a \mapsto u]\,\tau \sim [a \mapsto v]\,\tau$.

Lemma 1 gives $\theta_g\,U_w \wedge \mathcal{E}_{\phi_w} \;\leftrightarrow\; U'_w \wedge \mathcal{E}_{\theta_w}$, so from (R3) we have $\phi_g\,(U'_w \wedge \mathcal{E}_{\theta_w}) \;\Vdash\; \phi_g\,(\theta_g\,U_w)$. Moreover Lemma 2 gives $U_g \Vdash \mathcal{E}_{\phi_g \circ \theta_g}$, so property (P) gives the required entailment. □

Principality is more interesting, however. This requires that the constraint solver delivers most general solutions, which intuitively means that it makes no 'guesses' that are not implied by the original wanted constraints. Vytiniotis et al. (2011) define 'guess-free solutions' as those where the wanted constraints entail the residual constraints; for the plugin this amounts to requiring

$$\mathcal{Q} \wedge (Q_g \wedge U_g) \wedge (Q_w \wedge U_w) \Vdash Q_w \wedge \phi_g\,(U'_w \wedge \mathcal{E}_{\theta_w}).$$

Unfortunately, this is not true! Consider the sole wanted constraint $\alpha^2 \sim \beta^3$, which according to the algorithm in Figure 7 can be solved by $\theta = [\alpha \mapsto \gamma^3, \beta \mapsto \gamma^2]$ where γ is a fresh unification variable. The guess-free solution condition would require us to show

$$\alpha^2 \sim \beta^3 \;\Vdash\; \alpha \sim \gamma^3 \wedge \beta \sim \gamma^2,$$

and this is simply not derivable, because the fresh variable γ has been conjured out of thin air.

So what has become of the claim that the steps of the algorithm described above are all most general? The unifier θ is indeed the most general unifier, in the sense that any unifying substitution for this equation must determine γ and agree with θ up to the abelian group laws. The fresh variable γ must be $\alpha \cdot \beta^{-1}$, but the definition of guess-free solutions does not make use of this knowledge.

Instead, we can prove a weaker result:

Theorem 2 (Generality)**.** *If* $\mathcal{Q}\,;\,Q_g\,;\,\overline{\alpha}_{\mathrm{tch}} \;\vdash\!\!\blacktriangleright^{\mathrm{p}} Q_r \;\rightsquigarrow\; Q_s$ *then* $Q_g \wedge Q_r \Vdash \psi\,Q_s$ *for some subsitution* ψ *for the freshly introduced variables, i.e. with domain* $\mathrm{fuv}(Q_s) \setminus \mathrm{fuv}(Q_r)$.

Proof. Taking $\psi = \phi_w$ we must show that

$$\mathcal{Q} \wedge (Q_g \wedge U_g) \wedge (Q_w \wedge U_w) \Vdash \phi_w\,(Q_w \wedge \phi_g\,(U'_w \wedge \mathcal{E}_{\theta_w}))$$

is derivable. Now $\mathrm{dom}(\phi_w) \,\#\, \mathrm{fuv}(Q_w)$ and $Q_w \Vdash Q_w$, so it is enough to show $U_g \wedge U_w \Vdash \phi_w\,\phi_g\,(U'_w \wedge \mathcal{E}_{\theta_w})$.

Lemma 1 gives $\theta_g\,U_w \wedge \mathcal{E}_{\phi_w} \;\leftrightarrow\; U'_w \wedge \mathcal{E}_{\theta_w}$, so by (R3) we have $\phi_w\,\phi_g\,(\theta_g\,U_w \wedge \mathcal{E}_{\phi_w}) \leftrightarrow \phi_w\,\phi_g\,(U'_w \wedge \mathcal{E}_{\theta_w})$, which implies that $\phi_w\,\phi_g\,\theta_g\,U_w \;\Vdash\; \phi_w\,\phi_g\,(U'_w \wedge \mathcal{E}_{\theta_w})$. Now we must have $\mathrm{dom}(\phi_w) \,\#\, \mathrm{fuv}(\phi_g \theta_g\,U_w)$ so $\phi_g\,\theta_g\,U_w \;\Vdash\; \phi_w\,\phi_g\,(U'_w \wedge \mathcal{E}_{\theta_w})$. Moreover Lemma 2 gives $U_g \;\Vdash\; \mathcal{E}_{\phi_g \circ \theta_g}$ so we can deduce the required entailment using property (P). □

That is, the solution found by the algorithm may not be guess-free in the original sense, but there is some substitution for the fresh variables it introduces by which it can be transformed into a guess-free solution. I conjecture that this weaker property is in fact sufficient for the proof that OutsideIn(X) type inference (if it succeeds) delivers principal types.[18]

The underlying problem here is that OutsideIn(X) does not have a clear notion of scope for type variables: it is not the case that

$$\alpha^2 \sim \beta^3 \quad \leftrightarrow \quad \alpha \sim \gamma^3 \wedge \beta \sim \gamma^2,$$

but rather we must *contextualise* the variables, as in

$$\exists \alpha.\exists \beta.\; \alpha^2 \sim \beta^3 \quad \leftrightarrow \quad \exists \alpha.\exists \beta.\exists \gamma.\; \alpha \sim \gamma^3 \wedge \beta \sim \gamma^2.$$

In fact the same problem shows up in the algorithm described by Vytiniotis et al. (2011), which reduces the wanted constraint $\mathtt{F}(\mathtt{G}(x)) \sim y$ to $\mathtt{F}(\beta) \sim y \wedge \mathtt{G}(x) \sim \beta$ where F and G are type families and β is a fresh unification variable; it would appear that

$$\mathtt{F}(\mathtt{G}(x)) \sim y \;\not\Vdash\; \mathtt{F}(\beta) \sim y \wedge \mathtt{G}(x) \sim \beta$$

contrary to their Lemma 7.2.

On another note, observe that the proofs relied on an additional rule, TORSION-FREE, beyond the usual laws of an abelian group. This is crucial for proving both that solutions to wanted constraints are most general, and that simplifications of given constraints are sound. It amounts to restricting models of Unit to being *free* abelian groups, i.e. those generated by the base units and abelian group laws but with no other equations.[19]

Without TORSION-FREE, the addition of an axiom $\mathsf{kg} \circledast \mathsf{kg} \;\sim\; \mathbb{1}$ would be consistent, but then it would no longer be most general to solve the wanted $\alpha \circledast \alpha \;\sim\; \mathbb{1}$ with $\alpha \;\sim\; \mathbb{1}$, nor would it be sound to simplify the given $a \circledast a \;\sim\; \mathbb{1}$ to $a \;\sim\; \mathbb{1}$, as in either case kg is an alternative solution.

5. Related Work

The design of `uom-plugin` owes a lot to Andrew Kennedy's implementation of units of measure in F#, and Richard Eisenberg's `units` Haskell library. I compare it with each of them in turn. While there are several other Haskell libraries for units of measure[20], making slightly different design choices, `units` represents the state of the art and the comparison is broadly representative.

[18] Theorems 3.2 and 5.2 of Vytiniotis et al. (2011)

[19] Free abelian groups are always torsion-free, and torsion-free finitely generated abelian groups are free.

[20] Notably `dimensional-dk` by Björn Buckwalter and Douglas McClean (`https://github.com/bjornbm/dimensional-dk`)

5.1 Units of Measure in F#

The plugin described in this paper provides support for units of measure that is inspired by, and broadly comparable with, Kennedy's implementation in F#.[21] Constants and numeric types can be annotated with units, units may be polymorphic, and unit equations that arise during typechecking are solved by abelian group unification.

Working in Haskell introduces many new feature interactions to explore, notably with typeclasses, GADTs, type families, higher-kinded and higher-rank types. For example, Haskell allows definitions that are polymorphic in type *constructors* of kind $\mathsf{Unit} \to *$.

On the other hand, while the GHC typechecker plugins support makes some exciting new things possible, a plugin cannot (yet) extend GHC with a completely new language feature. In particular, Template Haskell quasiquotation allows the introduction of new syntax (e.g. for expressions containing quantities with units, or types mentioning units), but this syntax will not be used in output (such as error messages or inferred types). Thus the user can write

$$[\mathrm{u}|\,5\,\mathsf{m/s}\,|] :: \mathsf{Quantity\ Int}\ [\mathrm{u}|\,\mathsf{m/s}\,|]$$

but the inferred type of this expression is the less easy to read

$$\mathsf{Quantity\ Int}\ (\mathsf{Base}\ \texttt{"m"} \oslash \mathsf{Base}\ \texttt{"s"})$$

Moreover, there is no way to simplify an inferred type in a domain-specific manner. Thus a type may sometimes be presented as

$$\mathsf{Num}\ a \Rightarrow \mathsf{Quantity}\ a\ (\mathsf{Base}\ \texttt{"s"} \circledast (\mathsf{Base}\ \texttt{"m"} \oslash \mathsf{Base}\ \texttt{"s"}))$$

rather than the (equivalent)

$$\mathsf{Num}\ a \Rightarrow \mathsf{Quantity}\ a\ (\mathsf{Base}\ \texttt{"m"})$$

It should be relatively straightforward to extend GHC's plugin support to allow extensions to pretty-printing and presentation of inferred types, but there will always be limitations of the plugin technique compared to building support into the language, as in F#.

5.2 The `units` Package

Another key inspiration for this work is the `units` library (Muranushi and Eisenberg 2014), which is the state of the art as far as units of measure in Haskell are concerned. As discussed in subsection 2.2.1, `uom-plugin` is able to achieve better type inference behaviour and more comprehensible error messages than `units` thanks to the use of a typechecker plugin, rather than encoding everything using type families and other existing GHC features. On the other hand, since `units` does not require a plugin it is more broadly compatible and avoids the potential for plugin-introduced bugs. Moreover, it makes use of Template Haskell to permit a relatively nice input syntax.

Another crucial difference in library design is that `units` is based around working with dimensions (such as length and mass), rather than units directly. A dimension has a 'canonical' unit that determines how quantities are represented, but they may be introduced or eliminated using other units, with appropriate conversions performed automatically. There is even support for working with multiple local coherent systems of units (choices of canonical units for dimensions) in different parts of a single program. This allows code to be typechecked for dimension safety, but remain polymorphic in the particular units, and makes it easier to avoid numeric overflow errors when working with quantities at vastly different scales.

In the interests of simplicity, the `uom-plugin` library follows F#'s approach of indexing types by units of measure alone, not including dimensions, but the approach described in this paper should be able to scale to handle dimensions. The best way to represent them, and provide features such as automatic conversion between units of the same dimension, is a matter of ongoing work.

[21] Prior to the upcoming F# 4.0, which will support fractional units.

5.3 Plugging in an SMT Solver

This paper described a plugin to support units of measure by providing a special-purpose constraint solving algorithm based on abelian group unification. In contrast, Diatchki (2015) describes `type-nat-solver`,[22] a plugin that interfaces with an SMT solver to handle constraints arising from type-level natural numbers. In principle, the SMT solver approach could be extended to deal with other domains, such as abelian groups.

However, an SMT solver is designed to determine whether or not a given collection of constraints is satisfiable. If so, it will typically produce a satisfying assignment of values to variables. This is not immediately enough for use in type inference, which requires finding most general solutions to constraints involving unification variables.

For example, a constraint like $\alpha \cdot \beta \sim \mathbb{1}$ has many satisfying assignments (such as $\alpha \sim \mathbb{1}, \beta \sim \mathbb{1}$) but we need to determine the most general solution (namely $\alpha \sim \beta^{-1}$). It is possible to 'improve' constraints in an ad-hoc or theory-specific way, by guessing a candidate constraint and testing whether it follows from the other constraints, but this makes it hard to specify exactly which type inference problems will be solved by the system.

Thus there is room for experimentation with both special-purpose unification algorithms (such as that described in the present paper) and application of general SMT solvers to type inference. The typechecker plugins framework described in section 3 offers a common theoretical basis for both techniques. A more radical step is to change the type system so that typechecking generates verification conditions directly, rather than unification problems, as in work on refinement types (Vazou et al. 2014).

6. Conclusion

In this paper, I have introduced the notion of typechecker plugins both as an implementation technique in GHC and in terms of the OutsideIn(X) framework. I have made use of this to define a library for units of measure with good type inference properties, in particular the ability to find most general solutions to constraints arising from unit polymorphism.

Practical use of plugins is still at an early stage, as they are quite low level and closely tied to GHC's constraint solver. There is much to do to build better abstractions on top of the low-level interface, and hence make it easier to write plugins without deep knowledge of GHC. Termination of constraint solving in the presence of plugins is a particularly tricky issue. It is quite easy for a poorly written plugin to create an infinite loop, for example by emitting a new but trivial constraint each time it is invoked. Moreover, while evidence generation gives some indication of soundness (albeit not consistency of the axiom system used to produce the evidence), it is hard to ensure that plugins deliver most general solutions to constraints.

Two main avenues for future work are extending the `uom-plugin` library itself, and adding features to GHC that make more powerful plugins possible. I will consider these, then suggest some possible other applications for the concept of typechecker plugins.

[22] `https://github.com/yav/type-nat-solver`

6.1 Further Support for Units of Measure

Evidence generation The prototype units of measure plugin does not yet support evidence generation (see subsection 3.1.2); rather, it follows the method of proof by blatant assertion. In principle it should be possible to generate proofs based on the abelian group axioms from Figure 6. This would allow GHC's `-dcore-lint` option to check that the plugin is generating correct output.

Formally, this would amount to translating the proofs of Lemma 1 and Theorem 1 into a program that generated evidence using the appropriate combination of group axioms for each constraint solving step. It would be slightly weaker than a fully mechanised correctness proof for the algorithm, however, as it would only ensure correctness on individual inputs (rather than for all possible inputs).

Automatic conversion inference As observed above, representing dimensions and inferring conversions between quantities of the same dimension is a matter of ongoing investigation. This is not essential, because the user can always write their own conversions, but it would be better if they were able to write something like

$$r = \mathit{convert}\ [\mathrm{u}|\ 10\,\mathsf{ft/min}\ |] \\ \qquad :: \mathsf{Quantity\ Double}\ (\mathsf{Base}\ \texttt{"m"} \oslash \mathsf{Base}\ \texttt{"s"})$$

and have the compiler automatically insert the conversion from $\mathsf{ft/min}$ to $\mathsf{m/s}$.

One way to encode such automatic conversions is through the definition of a pair of additional type families: Pack, which converts a list of (base unit, integer exponent) pairs into the corresponding unit, and Unpack, which represents a fully known unit as a list of such pairs (in a canonical order). Thus we have:

$$\begin{array}{l} \mathbf{type\ family}\ \mathsf{Pack}\ (xs :: [(\mathsf{Symbol}, \mathsf{Integer})]) :: \mathsf{Unit} \\ \quad \mathbf{where} \\ \qquad \mathsf{Pack}\ [\,] \qquad\qquad = \mathbb{1} \\ \qquad \mathsf{Pack}\ ((b, i) : xs) = (\mathsf{Base}\ b \oslash\!\!\!\!\wedge\ i) \circledast \mathsf{Pack}\ xs \\ \mathbf{type\ family}\ \mathsf{Unpack}\ (u :: \mathsf{Unit}) :: [(\mathsf{Symbol}, \mathsf{Integer})] \end{array}$$

$$\begin{array}{l} \mathsf{Pack}\ [(\texttt{"m"}, \mathsf{Pos}\ 2), (\texttt{"s"}, \mathsf{Neg}\ 1)] \\ \quad = \mathsf{Base}\ \texttt{"m"} \oslash\!\!\!\!\wedge\ 2 \oslash \mathsf{Base}\ \texttt{"s"} \\ \mathsf{Unpack}\ (\mathsf{Base}\ \texttt{"m"} \oslash\!\!\!\!\wedge\ 2 \oslash \mathsf{Base}\ \texttt{"s"}) \\ \quad = [(\texttt{"m"}, \mathsf{Pos}\ 2), (\texttt{"s"}, \mathsf{Neg}\ 1)] \end{array}$$

(Here $\oslash\!\!\!\!\wedge$ represents exponentiation for units, and type-level integers are represented as natural numbers with a Pos or Neg constructor.)

Pack can be defined via a standard closed type family, but Unpack must be defined specially by the plugin because it observes the structure of the unit. It respects the equational theory on units, and hence does not break type soundness.

Together, these type families make it possible to encode the *convert* function and other advanced features using existing GHC Haskell type-level programming techniques. However, once more it becomes a challenge to make error messages simple and comprehensible. It is an interesting challenge to extend the units of measure library further while maintaining a suitable balance between features implemented in the plugin and those encoded using existing functionality.

Construction of quantities It is slightly unsatisfying that constructing literal quantities in a safe way fundamentally requires Template Haskell, rather than it providing mere syntactic sugar. One alternative is to expose the MkQuantity constructor to the user, and require them to follow a suitable syntactic discipline in its use: always instantiating its type to concrete units. A way to lift this restriction would be beneficial, but by no means essential.

Termination and completeness On a more theoretical note, it would be nice to prove that the plugin-extended constraint solver terminates, and is complete in an appropriate sense. Unfortunately, both of these are tricky issues in OutsideIn(X) even before plugins are added, and modular reasoning about termination is particularly difficult.

Extending the algebraic structure Finally, while indexing quantities by a single abelian group of units is a reasonable point in the design space, there are other choices for the model of units and quantities. For example:

- *Dimensions* such as length and time could be tracked separately, and their consistency checked, as in `units` (see section 5.2).
- *Fractional units* are sometimes useful, such as $\sqrt{\mathsf{Hz}}$ (i.e. $\mathsf{Hz}^{1/2}$), which arises when quantifying electronic noise levels.
- *Multiple origins* need to be considered to handle units of temperature, since $0C \approx 273K$. It may be possible to handle these by indexing quantities by an abelian group of *translations* as well as units (Atkey et al. 2013).
- *Logarithmic units* such as dBm require arithmetic operations like $\oplus$ to be given different types.

There is a direct trade-off between simplicity and expressivity of the system. The example of F# suggests that the simple abelian group model of units is useful in practice.

6.2 Extensions to the Plugins Mechanism

Apart from the constraint solver, there are many other points where it would be useful for typechecker plugins to be able to extend the compiler with domain-specific behaviour:

- control over simplification and presentation of inferred types, as discussed in section 5.1;
- easily defining special reduction behaviour for type families, such as the Unpack type family described in section 6.1;
- manipulating error messages, for example so that a DSL implementor can provide domain-specific guidance on likely reasons for a certain class of error, along the lines of error reflection in Idris (Christiansen 2014).

6.3 Other Applications for Plugins

Beyond units of measure and type-level numbers, there are many other potential applications for typechecker plugins:

- permitting injective type families (Stolarek et al. 2015);
- indexing a monad by the available effects, using a solver for a theory of sets, maps or boolean rings, as in the `effect-monad` library of Orchard and Petricek (2014);
- typeclasses such as Coercible (Breitner et al. 2014) and Typeable (Lämmel and Peyton Jones 2003), with non-standard search strategies rather than the usual instance search;
- adding η-laws for type-level tuples or record types; and
- record system extensions, such as extensible records via row polymorphism, or the proposed `OverloadedRecordFields`.[23]

[23] `https://ghc.haskell.org/trac/ghc/wiki/Records/OverloadedRecordFields`

In particular, it would be interesting to try factoring out an existing piece of GHC functionality (such as the Coercible or Typeable typeclasses) into a plugin, increasing modularity. One could even imagine disabling the entire built-in constraint solver, allowing experimentation with alternative algorithms, although this is likely to be practically difficult as it would require the plugin to represent substitutions more directly.

More generally, the existing typechecker plugin interface is at a relatively low level, requiring the plugin implementer to have a fairly detailed knowledge of the way type inference is implemented in GHC (e.g. to generate evidence using its internal data types). A broader challenge for future work is to find a suitable interface that is both powerful enough to implement special-purpose constraint solver behaviour, and simple enough to make the creation of domain-specific constraint solvers accessible to more users. Hopefully it should be possible to build such a higher-level interface on top of the existing typechecker plugins support in GHC.

Acknowledgments

Richard Eisenberg kindly gave detailed feedback on an early draft of this paper, in addition to offering inspiration through his `units` library. My thanks go to Neil Ghani and Conor McBride for ongoing discussion of this work. I'm grateful to attendees at various meetings, and to the anonymous reviewers, for their feedback. This paper was typeset using `lhs2TeX`.[24] The work was funded by the University of Strathclyde EPSRC Impact Acceleration Account.

References

Robert Atkey, Patricia Johann, and Andrew Kennedy. Abstraction and invariance for algebraically indexed types. In *Proceedings of the 40th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '13, pages 87–100. ACM, 2013.

Joachim Breitner, Richard A. Eisenberg, Simon Peyton Jones, and Stephanie Weirich. Safe zero-cost coercions for Haskell. In *Proceedings of the 19th ACM SIGPLAN International Conference on Functional Programming*, ICFP '14, pages 189–202. ACM, 2014.

David Raymond Christiansen. Reflect on your mistakes! Lightweight domain-specific error messages, 2014. URL `http://www.itu.dk/people/drc/drafts/error-reflection-submission.pdf`.

Iavor S. Diatchki. Improving Haskell types with SMT. In *Proceedings of the 2015 ACM SIGPLAN Symposium on Haskell*, Haskell '15. ACM, 2015. In press.

Jacques Garrigue. Relaxing the value restriction. In Yukiyoshi Kameyama and Peter J. Stuckey, editors, *Functional and Logic Programming*, volume 2998 of *LNCS*, pages 196–213. Springer, 2004.

Adam Gundry. *Type Inference, Haskell and Dependent Types*. PhD thesis, University of Strathclyde, 2013. URL `http://adam.gundry.co.uk/pub/thesis/`.

Andrew Kennedy. *Programming Languages and Dimensions*. PhD thesis, University of Cambridge, 1996. URL `http://research.microsoft.com/en-us/um/people/akenn/units/ProgrammingLanguagesAndDimensions.pdf`.

Andrew Kennedy. Types for units-of-measure: Theory and practice. In Zoltán Horváth, Rinus Plasmeijer, and Viktória Zsók, editors, *Central European Functional Programming School*, volume 6299 of *LNCS*, pages 268–305. Springer, 2010.

Ralf Lämmel and Simon Peyton Jones. Scrap your boilerplate: A practical design pattern for generic programming. In *Proceedings of the 2003 ACM SIGPLAN International Workshop on Types in Languages Design and Implementation*, TLDI '03, pages 26–37. ACM, 2003.

Daan Leijen and Erik Meijer. Domain specific embedded compilers. In *Proceedings of the 2nd Conference on Domain-Specific Languages*, DSL '99, pages 109–122. ACM, 1999.

Takayuki Muranushi and Richard A. Eisenberg. Experience report: Type-checking polymorphic units for astrophysics research in Haskell. In *Proceedings of the 2014 ACM SIGPLAN Symposium on Haskell*, Haskell '14, pages 31–38. ACM, 2014.

Dominic Orchard and Tomas Petricek. Embedding effect systems in Haskell. In *Proceedings of the 2014 ACM SIGPLAN Symposium on Haskell*, Haskell '14, pages 13–24. ACM, 2014.

Tom Schrijvers, Simon Peyton Jones, Manuel Chakravarty, and Martin Sulzmann. Type checking with open type functions. In *Proceedings of the 13th ACM SIGPLAN International Conference on Functional Programming*, ICFP '08, pages 51–62. ACM, 2008.

Jan Stolarek, Simon Peyton Jones, and Richard A. Eisenberg. Injective type families for Haskell. In *Proceedings of the 2015 ACM SIGPLAN Symposium on Haskell*, Haskell '15. ACM, 2015. In press.

Martin Sulzmann, Martin Müller, and Christoph Zenger. Hindley/Milner style type systems in constraint form. Res. Rep. ACRC-99-009, University of South Australia, School of Computer and Information Science, 1999.

Martin Sulzmann, Manuel M. T. Chakravarty, Simon Peyton Jones, and Kevin Donnelly. System F with type equality coercions. In *Proceedings of the 2007 ACM SIGPLAN International Workshop on Types in Language Design and Implementation*, TLDI '07, pages 53–66. ACM, 2007.

Niki Vazou, Eric L. Seidel, Ranjit Jhala, Dimitrios Vytiniotis, and Simon Peyton Jones. Refinement types for Haskell. In *Proceedings of the 19th ACM SIGPLAN International Conference on Functional Programming*, ICFP '14, pages 269–282. ACM, 2014.

Dimitrios Vytiniotis, Simon Peyton Jones, Tom Schrijvers, and Martin Sulzmann. OutsideIn(X): Modular type inference with local assumptions. *Journal of Functional Programming*, 21(4–5):333–412, 2011.

Dimitrios Vytiniotis, Simon Peyton Jones, and José Pedro Magalhães. Equality proofs and deferred type errors: A compiler pearl. In *Proceedings of the 17th ACM SIGPLAN International Conference on Functional Programming*, ICFP '12, pages 341–352. ACM, 2012.

Brent A. Yorgey, Stephanie Weirich, Julien Cretin, Simon Peyton Jones, Dimitrios Vytiniotis, and José Pedro Magalhães. Giving Haskell a promotion. In *Proceedings of the 8th ACM SIGPLAN Workshop on Types in Language Design and Implementation*, TLDI '12, pages 53–66. ACM, 2012.

[24] `http://www.andres-loeh.de/lhs2tex/`