Type Inference for Units of Measure

Adam Gundry

University of Strathclyde, Glasgow
adam.gundry@cis.strath.ac.uk

Abstract. Units of measure and type-level numbers are examples of type system extensions involving equational theories. Type inference for such an extension requires unification in a nontrivial theory. This complicates the generalisation step required for let-polymorphism in ML-style languages, as variable occurrence does not imply dependency. Previous work on units of measure (by Kennedy in particular) integrated free abelian group unification into the Damas-Milner type inference algorithm, but struggled with generalisation. I describe an approach to problem solving based on incremental minimal-commitment refinements in a structured context, and hence present abelian group unification and type unification algorithms which make type generalisation direct.

1 Introduction

Consider the following function to reverse a list and append another:

$$\begin{array}{ll} \mathit{revApp}\ [\,] & \mathit{as} = \mathit{as} \\ \mathit{revApp}\ (x : \mathit{xs}) & \mathit{as} = \mathit{revApp}\ \mathit{xs}\ (x : \mathit{as}) \end{array}$$

What type should we give it? Instead of the usual Haskell type $[\,a\,] \to [\,a\,] \to [\,a\,]$, we could try to capture more information about the lengths of the lists involved, say $\mathit{Vec}\ a\ m \to \mathit{Vec}\ a\ n \to \mathit{Vec}\ a\ (m + n)$. Now integer variables m and n appear in types, and must be compared in the appropriate equational theory: in order for the recursive call to be accepted, the typechecker must verify that $(m + 1) + n \equiv m + (n + 1)$. This may hold as a consequence of the definition of the $(+)$ function, or it may require some algebra. It is unlikely that the same definition of $(+)$ will make both *revApp* and concatenate $(+\!\!+)$ typecheck directly.

Alternatively, consider this function, conventionally of type $\mathit{Float} \to \mathit{Float}$:

$$\begin{array}{l} \mathit{distanceTravelled}\ \mathit{time} = \mathit{velocity} * \mathit{time} + (\mathit{acceleration} * \mathit{time} * \mathit{time})\ /\ 2 \\ \quad \mathbf{where}\ \{\, \mathit{velocity} = 2.0; \mathit{acceleration} = 3.6 \,\} \end{array}$$

Kennedy [6] teaches us how to enforce **units of measure**: with velocity in $\mathbf{ms}^{-1}$ and acceleration in $\mathbf{ms}^{-2}$, the system could infer the type $\mathit{Float}\langle\mathbf{s}\rangle \to \mathit{Float}\langle\mathbf{m}\rangle$. The more specific types of both functions have subexpressions that must be compared in a more liberal equational theory than syntactic equality.

In previous work [4], McBride, McKinna and I described a rationalisation of syntactic unification and Hindley-Milner type inference in which term and type variables live in a single dependency-ordered context. We solve problems in small, easily verified steps, each of which is most general. The additional structure in the context makes type generalisation for let-polymorphism particularly easy: we simply 'skim off' generalisable type variables from the end of the context.

Having applied our technique to the Hindley-Milner type system as a feasibility study, in this paper I extend the unification algorithm (and hence type inference) to support the equational theory of free abelian groups. This is a particularly fruitful choice as it retains decidable principal type inference. It is not the end of the story, but rather an example of the value of our approach to problem solving: that it gives a clearer account of the subtle issues surrounding generalisation. Vytiniotis et al. [16] argue that "let should not be generalised" because of the difficulties generalisation presents in their setting (a complex equational theory including type-level functions and GADT local equality constraints). They may well be right; but perhaps a better account of generalisation will help us decide.

Kennedy [6–8] elegantly uses abelian groups to model units of measure with support for polymorphism, and has introduced them into the functional programming language F# [15]. However, he encounters problems with the approach to let-generalisation in the Damas-Milner type inference algorithm. It uses the occur-check to identify generalisable variables (those that are free in the type but not the environment), but **variable occurrence does not imply variable dependency** for the equational theory of abelian groups. Later we will see another way of looking at this: given the equation $\alpha \equiv \tau$, where α is a variable and τ is a type, the solution $\alpha := \tau$ is not necessarily most general!

He gives the example [8, p. 292, in slightly different notation]

$$\lambda x.\ \text{let}\ d := \text{div}\, x\ \text{in}\ (d\ \text{mass}, d\ \text{time}), \qquad \text{where}$$
$$\text{div} :: \forall ab\ .\ \mathbb{F}\langle ab\rangle \to \mathbb{F}\langle a\rangle \to \mathbb{F}\langle b\rangle, \qquad \text{mass} :: \mathbb{F}\langle \mathbf{kg}\rangle, \qquad \text{time} :: \mathbb{F}\langle \mathbf{s}\rangle.$$

A naïve extension to the Damas-Milner algorithm fails to infer a type for this term, because polymorphism is lost: d is given the monotype $\mathbb{F}\langle a\rangle \to \mathbb{F}\langle ca^{-1}\rangle$ where a and c are unification variables, and a cannot unify with **kg** and **s**. However, if d is given its principal type scheme $\forall a. \mathbb{F}\langle a\rangle \to \mathbb{F}\langle ca^{-1}\rangle$, then the original term can be given type $\mathbb{F}\langle c\rangle \to \mathbb{F}\langle c\,\mathbf{kg}^{-1}\rangle \times \mathbb{F}\langle c\,\mathbf{s}^{-1}\rangle$ (see Section 5).

One possible solution to this problem is Kennedy's notion of *generaliser*, "a substitution that 'reveals' the polymorphism available under a given type environment" [7, p. 23]. Such a substitution preserves types in the context (up to the equational theory) but permutes group variables so that the Damas-Milner generalisation rule can be used. Calculating a generaliser is specific to the equational theory, technically nontrivial, and not implemented in F#:

```
> fun x -> let d y = x / y in (d mass, d time) ;;
-----------------------------------------^^^^
error FS0001: Type mismatch.
Expecting a float<kg> but given a float<s>
The unit of measure 'kg' does not match the unit of measure 's'
```

In the algorithm I will describe, insufficiently general unification shows up clearly as the source of this problem, and it has a correspondingly straightforward solution. With more structure in the context than just a set of typing assumptions, it is easier to see where generality can be lost, and we can prevent the loss of polymorphism in the first place, rather than trying to recover it after the fact. Maintaining generality explains the need for a new algorithm for abelian group unification: while the problem can be reduced to linear integer constraint solving, standard algorithms do not preserve the properties we care about!

Many authors have proposed designs for systems of units of measure, and relationships with linear algebra are well-established [5]. Following Kennedy, I will use integer powers, so units form a free abelian group. Some authors use rational powers (giving a vector space), including Rittri [13], who discusses the merits of both approaches. Chen et al. [3] give a useful overview of work on units of measure, and describe an alternative approach using static analysis.

Several impressive implementations of units of measure use advanced type system features such as GHC Haskell extensions [1] and C++ templates [14]. However, the difficulty of expressing a nontrivial equational theory at the type level means that they are complex, have limited inference capabilities and tend to expose the internal implementation in unfriendly error messages. Making units a type system extension, as in F#, results in a much more user-friendly system.

Rémy [12] extends the ML type system with other equational theories, using ranked unification to achieve easy generalisation; he does not address theories for which variable occurrence does not imply dependency, such as that of abelian groups. Variable ranking (as formalised by Kuan and MacQueen [9]) is also used in many ML type checkers for efficient generalisation; the algorithm described in the present paper implicitly manages ranks, by permuting the context.

In this paper, I describe a framework for contextual problem solving (Section 2) and an algorithm for abelian group unification in this framework (Section 3). Using this, I extend type unification to handle units of measure (Section 4), identifying a refinement needed for most general results. I sketch the corresponding type inference algorithm (Section 5) and conclude with some possible future directions (Section 6). A Haskell implementation of the algorithms is available at `http://personal.cis.strath.ac.uk/~adam/units-of-measure/`.

Acknowledgments I thank Conor McBride for helpful supervision and (along with James McKinna) for collaboration on previous work. This work was supported by Microsoft Research through its PhD Scholarship Programme.

2 Unification in context

Let me begin by defining the interconnected notions of context and contextualised statement. Informally, a well-formed context is one in which every declaration is explained by those which precede it. The declarations in a context induce an equational theory, so we can consider how to evolve a context to solve an equation. This requires a notion of 'information increase' between contexts, capturing legitimate steps towards a solution. A solution is most general if all others can be obtained from it by information increases.

A *context* is a list of variable declarations; I write $\mathcal{E}$ for the empty context and let Γ, Δ, Θ range over contexts. Variables come in different *sorts*: TY for syntactic type variables, GR for group variables and TM for term variables; all are bound in a single context. I write $\mathcal{T} = \{\mathrm{TY}, \mathrm{GR}, \mathrm{TM}\}$ for the set of sorts. Let $\mathcal{V}_T$ be a distinct sets of variables for each sort $T \in \mathcal{T}$. I use x as a variable of any sort or sort TM; α, β, γ for sort TY (or one of TY or GR) and a, b, c for sort GR only.

Statements are assertions that can be judged in contexts. Write $\Gamma \vdash S$ if statement S holds in context Γ. The statement forms we will consider are:

$$\begin{array}{rll} S ::= & \mathbf{valid} & \text{the context is well-formed;} \\ \mid & S \wedge S' & \text{both statements } S \text{ and } S' \text{ hold;} \\ \mid & e \equiv_T e' & e \text{ and } e' \text{ are equivalent expressions of sort } T. \end{array}$$

I regard $\Gamma \vdash \cdot \equiv_T \cdot$ as a partial equivalence relation, so it is reflexive on *well-formed* expressions, and write e **is** T for $e \equiv_T e$. Thus τ is a well-formed type in Γ if $\Gamma \vdash \tau$ **is** TY. A statement is well-formed if it contains well-formed expressions.

A *declaration* xD in a context assigns a *property* D to a variable x. For each sort T, we must describe the set of properties that can be assigned, and explain when they are valid context extensions by giving a *validity map* $\mathbf{ok}_T$ from properties to statements. We must also explain what declarations mean by giving a *interpretation map* $[\![\cdot]\!]_T$ from declarations of sort T to statements.

For $T \in \{\mathrm{TY}, \mathrm{GR}\}$, variables may either be unknown or defined, and a declared variable is a well-formed expression that is equal to its definition, if any:

$$D ::= \,?\, \mid \,:=e\,, \qquad \begin{array}{lll} \mathbf{ok}_T(\,?) & \mapsto & \mathbf{valid}, \\ \mathbf{ok}_T(:=e) & \mapsto & e \ \mathbf{is}\ T, \end{array} \qquad \begin{array}{lll} [\![\alpha ?]\!]_T & \mapsto & \alpha \ \mathbf{is}\ T, \\ [\![\alpha := e]\!]_T & \mapsto & \alpha \equiv_T e. \end{array}$$

Note that from $\alpha \equiv_T e$ we will be able to conclude α **is** T (i.e. $\alpha \equiv_T \alpha$) by symmetry and transitivity. I will introduce the sort TM in Section 5.

$$\frac{}{\mathcal{E} \vdash \mathbf{valid}} \qquad \frac{\Gamma \vdash \mathbf{valid} \quad \Gamma \vdash \mathbf{ok}_T D}{\Gamma, xD \vdash \mathbf{valid}}\ x \in \mathcal{V}_T \setminus \mathcal{V}_T(\Gamma) \qquad \frac{\Gamma \vdash \mathbf{valid}}{\Gamma \fatsemi \vdash \mathbf{valid}}$$

$$\textsc{Lookup}\ \frac{xD \in \Gamma}{\Gamma \vdash [\![xD]\!]_T} \qquad \frac{S \quad S'}{S \wedge S'}$$

$$\frac{d \equiv_T e}{e \equiv_T d} \qquad \frac{d \equiv_T e \quad e \equiv_T f}{d \equiv_T f} \qquad \frac{\tau_0 \equiv_{\mathrm{TY}} \upsilon_0 \quad \tau_1 \equiv_{\mathrm{TY}} \upsilon_1}{\tau_0 \to \tau_1 \equiv_{\mathrm{TY}} \upsilon_0 \to \upsilon_1}$$

Fig. 1. Rules for context validity, lookup, conjunction and equivalence

Figure 1 gives rules to construct a valid context and look up properties of variables in the context. Note that validity of properties $\mathbf{ok}_T D$ is used to establish validity of the context, whereas the interpretation $[\![xD]\!]_T$ holds by Lookup if the declaration xD is found in the context. I will discuss $\fatsemi$ in Section 2.1.

The figure also gives rules to prove conjunctions and make $\Gamma \vdash \cdot \equiv_T \cdot$ an equivalence relation on well-formed expressions. The sort TY of types has a binary constructor $\to$ for function types. I omit the context when it is constant.

I write $\mathcal{V}_T(\Gamma)$ for the variables that are **bound** in the context Γ. The rules ensure that a valid context has no duplicated variables. This set is different from the set of **free** variables in a context suffix or expression X, which we write $\mathrm{FV}_T(X)$. Thus $\mathcal{V}_{\mathrm{TY}}(\alpha ?, \beta := \alpha) = \{\alpha, \beta\}$ while $\mathrm{FV}_{\mathrm{TY}}(\beta := \alpha) = \{\alpha\}$. A valid context defines all the variables it refers to, so it has no free variables! Or, to put it another way, free variables of an expression are those bound in its context.

For example, $\Gamma_0 = \alpha ?, \beta := \alpha \to \alpha$ is a valid context and $\Gamma_0 \vdash \beta \equiv \alpha \to \alpha$ by Lookup. However, $\beta := \alpha, \alpha ?$ is not a valid context because β is not well-defined.

Derivations possess a monadic substitution structure analogous to that of expressions: the Lookup axiom is to derivations as variables are to expressions.

2.1 Solving problems by increasing information

Problem solving requires evolving the context in which a problem is posed into a context in which it is solved. For example, a unification problem consists of a context and two well-formed expressions; a solution must evolve the context to equate the expressions. So what are the legal evolutions of contexts?

A *substitution* δ *from* Γ *to* Δ is given by maps $\delta_T : \mathcal{V}_T(\Gamma) \to \{e \mid \Delta \vdash e \ \mathbf{is}\ T\}$ from variables in Γ to well-formed expressions over Δ for each sort $T \in \mathcal{T}$. This substitution can be applied to a well-formed expression e (or statement S) over Γ, replacing every variable x of sort T with $\delta_T(x)$ to give a well-formed expression δe (or statement δS) over Δ.

If δ is a substitution from Γ to Δ, and θ is a substitution from Δ to Θ, then $\theta \cdot \delta$ is the substitution from Γ to Θ given by $(\theta \cdot \delta)_T(x) = \theta(\delta_T(x))$ for $x \in \mathcal{V}_T(\Gamma)$. Equivalence of substitutions is considered up to the equational theory, comparing values at all variables in the source context: if δ and θ are substitutions from Γ to Δ then $\delta \equiv \theta$ means $\forall T \in \mathcal{T}.\ \forall x \in \mathcal{V}_T(\Gamma).\ \Delta \vdash \delta x \equiv_T \theta x$.

Substitutions let us move from one context to another, but a legitimate evolution of a context must also preserve information in the context. In particular, the interpretation $[\![xD]\!]_T$ of a context entry xD must hold in the new context.

However, we must also keep track of the order in the context, while allowing some permutation to deal with dependencies. I delimit *localities* within the context using $\fatsemi$ separators. These will be placed by the type inference algorithm when inferring the type of a let-definition, so it can be generalised over the declarations in the locality. Making a context entry less local (moving it from the right to the left of a separator) reduces the ability to generalise over it, so should be done only when essential for solving the problem. On the other hand, it is never possible to make a context entry more local (move it left to right).

Let $\downharpoonright$ be the partial function from contexts and natural numbers to contexts such that $\Gamma \downharpoonright n$ is Γ truncated after n occurrences of $\fatsemi$ separators, that is,

$$\Xi_0 \fatsemi \Xi_1 \fatsemi \cdots \fatsemi \Xi_m \downharpoonright n \;\mapsto\; \begin{cases} \Xi_0 \fatsemi \cdots \fatsemi \Xi_n, & \text{if } n \leq m, \\ \text{undefined}, & \text{if } n > m. \end{cases}$$

A substitution δ from Γ to Δ is an *information increase*, written $\delta : \Gamma \sqsubseteq \Delta$, if for all $n \in \mathbb{N}$ with $xD \in \Gamma \downharpoonright n$, we have that $\Delta \downharpoonright n$ is defined and $\Delta \downharpoonright n \vdash \delta[\![xD]\!]_T$. I write $\Gamma \sqsubseteq \Delta$ if δ is the identity substitution ι.

The idea is that the localities of Γ and Δ line up, and definitions in a locality of Γ hold as equations in the corresponding locality of Δ. An example increase is $\alpha? \fatsemi \beta? \sqsubseteq \beta?, \alpha := \beta \fatsemi$, but on the other hand, $\beta?, \alpha := \beta \fatsemi \not\sqsubseteq \alpha? \fatsemi \beta?$ as the first locality of the new context does not support β **is** TY or $\alpha \equiv \beta$.

A context Γ and well-formed expressions d and e form a *unification problem* $d \equiv_T e$. A *solution* is a context Δ and an information increase $\delta : \Gamma \sqsubseteq \Delta$ such that $\Delta \vdash \delta d \equiv_T \delta e$. We say this solution is *minimal* or *most general* if every other solution $\theta : \Gamma \sqsubseteq \Theta$ factors through it, i.e. there is a substitution $\zeta : \Delta \sqsubseteq \Theta$ such that $\theta \equiv \zeta \cdot \delta$. If the identity substitution is minimal, write $\Gamma \mathrel{\widehat{\sqsubseteq}} \Delta \vdash d \equiv_T e$. For example, in the context $\alpha? \fatsemi \beta?$, the type unification problem $\alpha \equiv_{\mathrm{TY}} \beta \to \beta$ has minimal solution $\alpha? \fatsemi \beta? \mathrel{\widehat{\sqsubseteq}} \beta?, \alpha := \beta \to \beta \fatsemi \vdash \alpha \equiv_{\mathrm{TY}} \beta \to \beta$.

We must ensure that statements we consider are *stable*: if S holds in a context, δS must hold after an information increase δ. This is easy to ensure by construction: we use only the LOOKUP rule to extract information from the context. Once a problem is expressed as a stable statement, we can solve it using a minimal commitment strategy, making the smallest information increases possible until the problem is solved. Thanks to stability, this strategy delivers most general solutions. This is essentially McBride's "optimistic optimisation" strategy [10].

3 Abelian group unification

Let us consider unification problems for abelian groups in the framework. A *group expression d (with constants in K)* is an expression of sort GR given by

$$d \;::=\; a \;\mid\; k \;\mid\; 0 \;\mid\; d + d \;\mid\; -d\,,$$

where $a \in \mathcal{V}_{\mathrm{GR}}$ and $k \in K$. As shown in Figure 2, I extend the rules for equivalence of expressions given in Figure 1 by reflexivity and congruence (making group expressions well-formed), together with the four abelian group axioms of commutativity, associativity, inverses and identity. I write a, b, c for group variables; d, e, f for group expressions and m, n for integers.

$$\boxed{d \equiv_{\mathrm{GR}} e}$$

$$\frac{}{0 \equiv_{\mathrm{GR}} 0} \qquad \frac{}{k \equiv_{\mathrm{GR}} k}\; k \in K \qquad \frac{d \equiv_{\mathrm{GR}} e}{-d \equiv_{\mathrm{GR}} -e} \qquad \frac{d_0 \equiv_{\mathrm{GR}} e_0 \quad d_1 \equiv_{\mathrm{GR}} e_1}{d_0 + d_1 \equiv_{\mathrm{GR}} e_0 + e_1}$$

$$\frac{d \textbf{ is } \mathrm{GR} \quad e \textbf{ is } \mathrm{GR}}{d + e \equiv_{\mathrm{GR}} e + d} \qquad \frac{d \textbf{ is } \mathrm{GR} \quad e \textbf{ is } \mathrm{GR} \quad f \textbf{ is } \mathrm{GR}}{(d + e) + f \equiv_{\mathrm{GR}} d + (e + f)}$$

$$\frac{d \textbf{ is } \mathrm{GR}}{d + (-d) \equiv_{\mathrm{GR}} 0} \qquad \frac{d \textbf{ is } \mathrm{GR}}{d + 0 \equiv_{\mathrm{GR}} d}$$

Fig. 2. Declarative rules for group expression equivalence

Let nd mean d added to itself n times, $(-n)d$ mean $-(nd)$ and $d-e$ mean $d+(-e)$. Group expressions have a normal form $\sum_i n_i d_i$ where the n_i are nonzero integers and the d_i are distinct atoms (variables or constants) sorted in some order. For example, the expression $a + a + b + 0 + b + a$ has normal form $3a + 2b$.

Consider the equation $3a + 2b \equiv 0$ in the context $a?, b?$. Since $2 \nmid 3$, we cannot simply solve for b, but we can simplify the problem by setting $b := c - a$ where c is a fresh variable. This gives $a + 2c \equiv 0$ in the context $a?, c?$, which can be solved by rearranging and taking $a := -2c$ to give $c?, a := -2c, b := c - a$.

More generally, when solving such an equation, we will ask whether a variable has the largest coefficient, and if not, reduce the other coefficients by it to simplify the problem. Some notation is in order. Suppose $d \equiv \sum_i n_i d_i$ and define:

$$\begin{aligned} \mathrm{maxc(d)} &= \max\{|n_i| \mid d_i \text{ is a variable}\}, && \text{highest absolute coefficient;} \\ Q_n(d) &= \textstyle\sum_i (n_i \operatorname{quot} n) d_i, && \text{quotient by } n \text{ of every coefficient;} \\ R_n(d) &= \textstyle\sum_i (n_i \operatorname{rem} n) d_i, && \text{remainder by } n \text{ of every coefficient;} \end{aligned}$$

where $\cdot \operatorname{quot} \cdot$ is integer division truncated towards zero, and $\cdot \operatorname{rem} \cdot$ is the corresponding remainder, so for example $-3 \operatorname{quot} 2 = -1$ and $-3 \operatorname{rem} 2 = -1$. The important points, which I will make use of later, are that for every d,

$$n Q_n(d) + R_n(d) \equiv d \qquad \text{and} \qquad \mathrm{maxc(R_n(d))} < n.$$

3.1 The abelian group unification algorithm

I must explain how to solve unification problems of the form $d \equiv_{\mathrm{GR}} e$. Thanks to the inverse operation, it suffices to consider the equivalent matching problem $d - e \equiv 0$, which I will write $d - e \;\; \mathbf{id}$.

Figure 3 shows the algorithm presented as a collection of inference rules. Given as input a context Γ, Ψ and a group expression d, the judgment $\Gamma, [\Psi] \twoheadrightarrow \Delta \vdash d \; \mathbf{id}$ means that the algorithm outputs the context Δ such that $\Delta \vdash d \; \mathbf{id}$. Note that the rules are entirely syntax-directed: at most one rule applies for any possible initial context and group expression. They lead directly to an implementation.

$$\boxed{\Gamma, [\Psi] \twoheadrightarrow \Delta \vdash d \ \mathbf{id}}$$

$$\textsc{Trivial}\ \frac{}{\Gamma, [\mathcal{E}] \twoheadrightarrow \Gamma \vdash 0 \ \mathbf{id}} \qquad \textsc{Repossess}\ \frac{\Gamma, [\Psi] \twoheadrightarrow \Delta \vdash d \ \mathbf{id}}{\Gamma ⨾, [\Psi] \twoheadrightarrow \Delta ⨾ \vdash d \ \mathbf{id}}$$

$$\textsc{Expand}\ \frac{\Gamma, \Psi, [\mathcal{E}] \twoheadrightarrow \Delta \vdash nf + e \ \mathbf{id}}{\Gamma, a{:=}f, [\Psi] \twoheadrightarrow \Delta, a{:=}f \vdash na + e \ \mathbf{id}}$$

$$\textsc{Ignore}\ \frac{\Gamma, [\Psi] \twoheadrightarrow \Delta \vdash d \ \mathbf{id}}{\Gamma, xD, [\Psi] \twoheadrightarrow \Delta, xD \vdash d \ \mathbf{id}}\ x \notin \mathrm{FV}_{\mathrm{GR}}(d)$$

$$\textsc{Define}\ \frac{}{\Gamma, a?, [\Psi] \twoheadrightarrow \Gamma, \Psi, a{:=}{-f} \vdash na + nf \ \mathbf{id}}\ n \neq 0$$

$$\textsc{Reduce}\ \frac{\Gamma, \Psi, [b?] \twoheadrightarrow \Delta \vdash nb + R_n(e) \ \mathbf{id}}{\Gamma, a?, [\Psi] \twoheadrightarrow \Delta, a{:=}b - Q_n(e) \vdash na + e \ \mathbf{id}}\ |n| \leqslant \mathrm{maxc(e)}, b \text{ fresh}$$

$$\textsc{Collect}\ \frac{\Gamma, [a?] \twoheadrightarrow \Delta \vdash na + e \ \mathbf{id}}{\Gamma, a?, [\mathcal{E}] \twoheadrightarrow \Delta \vdash na + e \ \mathbf{id}}\ |n| > \mathrm{maxc(e)}$$

Fig. 3. Algorithmic rules for abelian group unification

So how does the algorithm work? If the problem is 0 **id**, then it is TRIVIAL. Otherwise, we move back through the context, skipping over variables that do not occur in the problem (including type and term variables) using IGNORE, and moving through localities using REPOSSESS. When we encounter a defined variable, we must substitute it out (with EXPAND) to simplify the problem.

The boxed suffix Ψ will either be empty (written $\mathcal{E}$) or contain only the unknown variable with the strictly largest coefficient in d, if any. The REDUCE and COLLECT rules move this variable back in the context, since there is no simplification that can usefully be applied to it. Other rules will insert the variable into the context when it no longer has the largest coefficient.

The interesting cases arise when we reach an unknown variable a that occurs in the problem, which we write as $na + e$ **id** (always meaning that $a \notin \mathrm{FV}_{\mathrm{GR}}(e)$).

Suppose the normal form of e is $\sum_i n_i e_i$. There are four possibilities, either:

1. $n \mid n_i$ for all i;
2. $|n| \leq |n_i|$ for some i with e_i a variable, and the previous case does not apply;
3. $|n| > |n_i|$ for all i with e_i a variable, but e has at least one variable; or
4. e has no variables.

Case 1. If $n \mid n_i$ for all i, then there is some f such that $e \equiv nf$. The rule DEFINE applies and we set $a := -f$ to give $na + e \equiv na + nf \equiv -nf + nf \equiv 0$. This is clearly a solution, and it is most general for the free abelian group.

Case 2. If $|n| \leq |n_i|$ for some i with e_i a variable (but n does not divide all the coefficients), then the REDUCE rule applies and simplifies the problem by reducing the coefficients modulo n. Recall that $e \equiv nQ_n(e) + R_n(e)$ where $Q_n(e)$ is given by taking the quotient by n of the coefficients in e. We can generate a fresh variable b and define $a := b - Q_n(e)$, giving

$$na + e \equiv n(b - Q_n(e)) + e \equiv nb + (e - nQ_n(e)) \equiv nb + R_n(e).$$

Case 3. Here $|n| > |n_i|$ for all i, so neither of the two previous cases apply, but there are variables in e. Now n is the largest coefficient of a variable, so reducing the coefficients modulo n would leave them unchanged. Instead, we have to COLLECT a and move it further back in the context. This rule maintains the invariant that Ψ contains only the variable with the largest coefficient, if any; the invariant also guarantees that Ψ will be empty when the rule applies.

Case 4. Finally, if there are no variables in e then the problem is of the form $na + k$ **id**, where k is a constant expression and $n \nmid k$, so it has no solution.

3.2 Correctness of the algorithm

I prove correctness for the matching problem d **id**; correctness for the unification problem $d \equiv_{\text{GR}} e$ is a corollary. For details of the proofs, consult the appendix.

Lemma 1 (Soundness and generality of abelian group unification). *If unification succeeds with* $\Gamma, [\Psi] \twoheadrightarrow \Delta \vdash d$ **id**, *then* $\mathcal{V}_{\text{TY}}(\Gamma, \Psi) = \mathcal{V}_{\text{TY}}(\Delta)$, $\mathcal{V}_{\text{GR}}(\Gamma, \Psi) \subseteq \mathcal{V}_{\text{GR}}(\Delta)$ *and it gives a most general solution* $\Gamma, \Psi \mathrel{\hat{\sqsubseteq}} \Delta \vdash d \equiv_{\text{GR}} 0$.

Proof (Sketch). By structural induction on derivations. Each step preserves the meaning of the problem, so the result is a solution (soundness). Moreover, each step makes commitments only if they are essential to solving the problem, so the result is most general. The interesting part is proving generality of the REPOSSESS rule, since this involves moving Ψ into a new locality, which could restrict the solution. However, if Ψ contains a variable then it has the strictly largest coefficient, so the problem can be solved only by moving this variable. □

Lemma 2 (Completeness of abelian group unification). *If d is a well-formed group expression in Γ, and there is some $\theta : \Gamma \sqsubseteq \Theta$ such that $\Theta \vdash \theta d \equiv_{\text{GR}} 0$, then the algorithm produces Δ such that $\Gamma, [\mathcal{E}] \twoheadrightarrow \Delta \vdash d$* **id**.

Proof (Sketch). We can show that the algorithm terminates by exhibiting a termination metric, so we are justified in reasoning by structural induction on the call graph. Completeness is by the fact that the rules cover all solvable cases and preserve solutions, so if no rule applies then the original problem can have had no solutions. This occurs if a non-unit constant is equated to 0 or there is only one variable and its coefficient does not divide the coefficient of one of the constants (e.g. $2a + \mathrm{k}$ **id**). □

$$\boxed{\Gamma \twoheadrightarrow \Delta \vdash \tau \equiv \upsilon}$$

Decompose

$$\frac{\Gamma \twoheadrightarrow \Delta_0 \vdash \tau_0 \equiv \upsilon_0 \quad \Delta_0 \twoheadrightarrow \Delta \vdash \tau_1 \equiv \upsilon_1}{\Gamma \twoheadrightarrow \Delta \vdash \tau_0 \rightarrow \tau_1 \equiv \upsilon_0 \rightarrow \upsilon_1}$$

Idle

$$\frac{}{\Gamma, \alpha D \twoheadrightarrow \Gamma, \alpha D \vdash \alpha \equiv \alpha}$$

Define

$$\frac{}{\Gamma, \alpha? \twoheadrightarrow \Gamma, \alpha{:=}\beta \vdash \alpha \equiv \beta}\ \alpha \neq \beta$$

Expand

$$\frac{\Gamma \twoheadrightarrow \Delta \vdash \tau \equiv \beta}{\Gamma, \alpha{:=}\tau \twoheadrightarrow \Delta, \alpha{:=}\tau \vdash \alpha \equiv \beta}\ \alpha \neq \beta$$

Ignore

$$\frac{\Gamma \twoheadrightarrow \Delta \vdash \alpha \equiv \beta}{\Gamma, xD \twoheadrightarrow \Delta, xD \vdash \alpha \equiv \beta}\ x \notin \{\alpha, \beta\}$$

Skip

$$\frac{\Gamma \twoheadrightarrow \Delta \vdash \alpha \equiv \beta}{\Gamma\fatsemi \twoheadrightarrow \Delta\fatsemi \vdash \alpha \equiv \beta}$$

Solve

$$\frac{\Gamma \mid \mathcal{E} \twoheadrightarrow \Delta \vdash \alpha \equiv \tau}{\Gamma \twoheadrightarrow \Delta \vdash \alpha \equiv \tau}\ \tau \text{ not variable}$$

$$\boxed{\Gamma \mid \Xi \twoheadrightarrow \Delta \vdash \alpha \equiv \tau}$$

DefineS

$$\frac{\alpha \notin \mathrm{FV}_{\mathrm{TY}}(\tau, \Xi)}{\Gamma, \alpha? \mid \Xi \twoheadrightarrow \Gamma, \Xi, \alpha{:=}\tau \vdash \alpha \equiv \tau}$$

ExpandS

$$\frac{\Gamma, \Xi \twoheadrightarrow \Delta \vdash \upsilon \equiv \tau \qquad \alpha \notin \mathrm{FV}_{\mathrm{TY}}(\tau, \Xi)}{\Gamma, \alpha{:=}\upsilon \mid \Xi \twoheadrightarrow \Delta, \alpha{:=}\upsilon \vdash \alpha \equiv \tau}$$

IgnoreS

$$\frac{\Gamma \mid \Xi \twoheadrightarrow \Delta \vdash \alpha \equiv \tau}{\Gamma, xD \mid \Xi \twoheadrightarrow \Delta, xD \vdash \alpha \equiv \tau}\ \begin{array}{c}\alpha \neq x,\\ x \notin \mathrm{FV}_{\mathcal{T}}(\tau, \Xi)\end{array}$$

DependS

$$\frac{\Gamma \mid xD, \Xi \twoheadrightarrow \Delta \vdash \alpha \equiv \tau}{\Gamma, xD \mid \Xi \twoheadrightarrow \Delta \vdash \alpha \equiv \tau}\ \begin{array}{c}\alpha \neq x,\\ x \in \mathrm{FV}_{\mathcal{T}}(\tau, \Xi)\end{array}$$

Repossess

$$\frac{\Gamma \mid \Xi \twoheadrightarrow \Delta \vdash \alpha \equiv \tau}{\Gamma\fatsemi \mid \Xi \twoheadrightarrow \Delta\fatsemi \vdash \alpha \equiv \tau}$$

Symmetrical variants of Define, Expand and Solve omitted.

Fig. 4. Original algorithmic rules for type unification

4 Unification for types with units of measure

Having developed a unification algorithm for the theory of abelian groups, let us extend type unification to support units of measure. The unification algorithm from my previous work [4] is shown in Figure 4. As in the group unification algorithm in Section 3, the rules are entirely syntax directed and lead immediately to an implementation. There are two kinds of rule:

– 'Unify' steps start the process: given an input context Γ and well-formed types τ and υ, the judgment $\Gamma \twoheadrightarrow \Delta \vdash \tau \equiv \upsilon$ means that the unification problem $\tau \equiv_{\mathrm{TY}} \upsilon$ is solved with output context Δ.
– 'Solve' steps handle flex-rigid unification problems[1]: given a context Γ, Ξ, a type variable α in Γ and a well-formed non-variable type τ in Γ, Ξ, the judgment $\Gamma \mid \Xi \twoheadrightarrow \Delta \vdash \alpha \equiv \tau$ means that the problem $\alpha \equiv_{\mathrm{TY}} \tau$ is solved with output context Δ. The context suffix Ξ collects type or group variable declarations that τ depends on but that cannot be used to solve the problem.

[1] Recall that a *flex-rigid* problem is to unify a variable and a non-variable expression; a *flex-flex* problem has two variables and a *rigid-rigid* problem has two non-variables.

The algorithm starts by applying the DECOMPOSE rule to split up rigid-rigid problems into subproblems and solve them sequentially. If a flex-flex problem $\alpha \equiv \beta$ is reached, the input context is searched for α and β, moving past other entries with IGNORE or SKIP. When either variable is found, the problem is either ignored by IDLE as trivial, solved by DEFINE if the variable is unknown, or simplified by EXPAND substituting out the definition.

If a flex-rigid problem $\alpha \equiv \tau$ is reached, the SOLVE rule applies. Now the context is searched as in the flex-flex case, except that a list Ξ of hereditary dependencies of τ (either type or group variables) must be accumulated. These must be moved back in the context until DEFINES (solve α with τ) or EXPANDS (substitute out α) applies. Note the occur check performed by both these rules: if $\alpha \in \mathrm{FV}_{\mathrm{TY}}(\tau, \Xi)$ then $\alpha \equiv \tau$ has no solutions. The suffix Ξ may be moved into a previous locality by REPOSSESS, making its entries less generalisable, so DEPENDS only adds entries to it if necessary; otherwise IGNORES skips them.

For example, consider the context $\beta?, \alpha? \fatsemi \gamma?$ and constraint $\alpha \equiv \beta \to \gamma$. Since this is a flex-rigid problem the SOLVE rule applies, followed by DEPENDS as γ appears in the type. The REPOSSESS rule moves into the previous locality, making the accumulated γ less generalisable. Finally, DEFINES applies to solve the constraint giving the final context $\beta?, \gamma?, \alpha := \beta \to \gamma \fatsemi$.

4.1 Units of measure as an abelian group

A *unit (of measure)* is a group expression with constants in some set of base units. For familiarity's sake, I write units in multiplicative notation, with identity 1: if a, b are variables and $\mathbf{m}$, $\mathbf{s}$ are base units, then $ab\mathbf{m}^2\mathbf{s}^{-1}$ is a derived unit.

Let us extend the language of types with a single new type $\mathbb{F}\langle d \rangle$ of numbers parameterised by units, adding a congruence rule to the declarative system and a corresponding algorithmic rule that invokes abelian group unification:

$$\frac{d \equiv_{\mathrm{GR}} e}{\mathbb{F}\langle d \rangle \equiv_{\mathrm{TY}} \mathbb{F}\langle e \rangle}, \qquad \textsc{Unit}\ \frac{\Gamma, [\mathcal{E}] \twoheadrightarrow \Delta \vdash de^{-1}\ \mathbf{id}}{\Gamma \twoheadrightarrow \Delta \vdash \mathbb{F}\langle d \rangle \equiv \mathbb{F}\langle e \rangle}.$$

Suppose the algorithm is used to solve $\mathbb{F}\langle bc \rangle \to \alpha \equiv \mathbb{F}\langle b \rangle \to \mathbb{F}\langle c \rangle$ in the context $b?, \alpha?, c?$. First the constraint $\mathbb{F}\langle bc \rangle \equiv \mathbb{F}\langle b \rangle$ is reduced to $bc \equiv_{\mathrm{GR}} b$ by UNIT, and this is solved by group unification (Section 3) to give $b?, \alpha?, c := 1$. Then the constraint $\alpha \equiv \mathbb{F}\langle c \rangle$ is solved by moving c to give $b?, c := 1, \alpha := \mathbb{F}\langle c \rangle$.

Thus we have a unification algorithm for types, but is it correct? It certainly ought to be sound and complete, because the new algorithmic rule corresponds directly to the declarative rule. However, we shall see that generality fails.

4.2 Loss of generality and how to recover it

Suppose we seek $\alpha \equiv \mathbb{F}\langle b_0 b_1\rangle$ in the context $\alpha? \fatsemi b_0?, b_1?$. Following the algorithm, this flex-rigid problem is solved by moving b_0 and b_1 into the previous locality, and instantiating α, resulting in the context $b_0?, b_1?, \alpha := \mathbb{F}\langle b_0 b_1\rangle \fatsemi$. However, a more general solution exists, namely $c?, \alpha := \mathbb{F}\langle c\rangle \fatsemi b_0?, b_1 := c{b_0}^{-1}$, where c is a fresh group variable and b_0 is still local. Why did the algorithm fail to find this?

The syntactic equational theory has the property that equivalent expressions have the same sets of free variables.[2] Indeed, some other useful theories share this property [12]. However, it does not hold for the theory of abelian groups: for example, the equation $aa^{-1} \equiv 1$ has a free on the left only. Thus variable occurrence does not imply dependency. The syntactic occurs check performed by the unification algorithm is too hasty.

The problem is that, when solving a flex-rigid constraint, we do not actually know that the variable must be **syntactically** equal to the type: units need be equal only in the equational theory of abelian groups. We can decompose such constraints into a flex-rigid constraint on type variables, with fresh variables in place of units, and additional constraints to make the fresh variables equal to the units. A rigid type decomposes into a 'hull', or 'type skeleton'[3], that must match exactly, and a collection of constraints in the richer equational theory.

In our example, the constraint $\alpha \equiv \mathbb{F}\langle b_0 b_1\rangle$ becomes $\alpha \equiv_{\mathrm{TY}} \mathbb{F}\langle c\rangle \wedge c \equiv_{\mathrm{GR}} b_0 b_1$ in context $\alpha? \fatsemi b_0?, b_1?, c?$. After solving the first part we have $c?, \alpha := \mathbb{F}\langle c\rangle \fatsemi b_0?, b_1?$, and solving the second yields the principal solution $c?, \alpha := \mathbb{F}\langle c\rangle \fatsemi b_0?, b_1 := c{b_0}^{-1}$.

I write $\rho\langle -\rangle$ for the hull of the type ρ, parameterised by a vector of units: $\rho = \mathbb{F}\langle d\rangle \to \mathbb{F}\langle e\rangle$ has hull $\rho\langle -\rangle = \mathbb{F}\langle -\rangle \to \mathbb{F}\langle -\rangle$ and $\rho\langle \vec{a}\rangle = \mathbb{F}\langle a_0\rangle \to \mathbb{F}\langle a_1\rangle$.

Let us modify the rules to maintain the invariant that the only group variables a flex-rigid problem depends on (i.e. those in the rigid type τ or suffix Ξ) are fresh unknowns. This ensures group variables are never made less local by collecting them in Ξ as dependencies. Type unification does not prejudice locality of group variables: that is up to the group unification algorithm! The Solve and DependS rules are replaced by the following modified versions:

Solve′

$$\frac{\Gamma \mid \vec{b} \twoheadrightarrow \Delta_0 \vdash \alpha \equiv \rho\langle \vec{b}\rangle \qquad \Delta_0 \twoheadrightarrow \Delta \vdash \vec{b} \equiv_{\mathrm{GR}} \vec{e}}{\Gamma \twoheadrightarrow \Delta \vdash \alpha \equiv \rho\langle \vec{e}\rangle}\ \rho \text{ not variable}, \vec{b} \text{ fresh}$$

DependS′

$$\frac{\Gamma \mid \beta?, \Xi \twoheadrightarrow \Delta_0 \vdash \alpha \equiv \tau}{\Gamma, \beta? \mid \Xi \twoheadrightarrow \Delta \vdash \alpha \equiv \tau}\ \alpha \neq \beta, \beta \in \mathrm{FV}_{\mathrm{TY}}(\tau, \Xi)$$

[2] Such equational theories sometimes described as *regular* [2], but we avoid this term because it means too many different things in other contexts.

[3] This term was suggested by an anonymous reviewer of a previous version.

DEPENDS″

$$\frac{\Gamma \mid \vec{b}, \beta := \rho\langle\vec{b}\rangle, \Xi \twoheadrightarrow \Delta_0 \vdash \alpha \equiv \tau \qquad \Delta_0 \twoheadrightarrow \Delta \vdash \vec{b} \equiv_{\mathrm{GR}} \vec{e}}{\Gamma, \beta := \rho\langle\vec{e}\rangle \mid \Xi \twoheadrightarrow \Delta \vdash \alpha \equiv \tau} \; \begin{array}{l}\alpha \neq \beta, \vec{b} \text{ fresh,}\\ \beta \in \mathrm{FV}_{\mathrm{TY}}(\tau, \Xi)\end{array}$$

We solve vectors of equations one at a time, threading the context:

$$\frac{\Delta_0, [\mathcal{E}] \twoheadrightarrow \Delta_1 \vdash b_1 {e_1}^{-1} \; \mathbf{id} \quad \cdots \quad \Delta_{n-1}, [\mathcal{E}] \twoheadrightarrow \Delta_n \vdash b_n {e_n}^{-1} \; \mathbf{id}}{\Delta_0 \twoheadrightarrow \Delta_n \vdash b_1, \ldots, b_n \equiv_{\mathrm{GR}} e_1, \ldots, e_n}$$

4.3 Correctness of type unification

With the above refinement, type unification gives most general results.

Lemma 3 (Soundness and generality of type unification).

(a) If type unification succeeds with $\Gamma \twoheadrightarrow \Delta \vdash \tau \equiv \upsilon$, then $\mathcal{V}_{\mathrm{TY}}(\Gamma) = \mathcal{V}_{\mathrm{TY}}(\Delta)$, $\mathcal{V}_{\mathrm{GR}}(\Gamma) \subseteq \mathcal{V}_{\mathrm{GR}}(\Delta)$ and it gives a most general solution $\Gamma \mathrel{\widehat{\sqsubseteq}} \Delta \vdash \tau \equiv \upsilon$.

(b) Correspondingly, if $\Gamma \mid \Xi \twoheadrightarrow \Delta \vdash \alpha \equiv \tau$, then $\mathcal{V}_{\mathrm{TY}}(\Gamma, \Xi) = \mathcal{V}_{\mathrm{TY}}(\Delta)$, $\mathcal{V}_{\mathrm{GR}}(\Gamma) \subseteq \mathcal{V}_{\mathrm{GR}}(\Delta)$ and $\Gamma, \Xi \mathrel{\widehat{\sqsubseteq}} \Delta \vdash \alpha \equiv \tau$.

Proof (Sketch). By structural induction on derivations, as in Lemma 1, noting that each step preserves solutions and follows a minimal commitment strategy. The new rules in Section 4.2 ensure the type τ in the flex-rigid problem $\alpha \equiv \tau$ contains only group variables, not compound units. When a $\fatsemi$ separator is found, any solution must move all the dependencies into the previous locality. □

Lemma 4 (Completeness of type unification).

(a) If the types υ and τ are well-formed in Γ and there is some $\theta : \Gamma \sqsubseteq \Theta$ such that $\Theta \vdash \theta\upsilon \equiv \theta\tau$, then unification produces Δ such that $\Gamma \twoheadrightarrow \Delta \vdash \upsilon \equiv \tau$.

(b) Moreover, if $\theta : \Gamma, \Xi \sqsubseteq \Theta$ is such that $\Theta \vdash \theta\alpha \equiv \theta\tau$ and the following conditions are satisfied:

$\alpha \in \mathcal{V}_{\mathrm{TY}}(\Gamma)$, $\quad \tau$ is not a variable,

*$\Gamma, \Xi \vdash \tau$ **is** TY, $\quad \Xi$ contains only type or group variable declarations*

$\beta \in \mathcal{V}_{T}(\Xi) \Rightarrow \beta \in \mathrm{FV}_{T}(\tau, \Xi)$;

then there is some context Δ such that $\Gamma \mid \Xi \twoheadrightarrow \Delta \vdash \alpha \equiv \tau$.

Proof (Sketch). As before, we show termination, then reason by structural induction. The rules preserve solutions, so if a recursive call fails then the whole problem must have no solution. The only cases not covered are rigid-rigid mismatches (such as unifying $\upsilon \rightarrow \tau$ with $\mathbb{F}\langle d\rangle$) and occur-check failures (such as unifying α with $\alpha \rightarrow \alpha$), neither of which have any solutions. □

$$\boxed{t : \tau}$$

$$\frac{x :: .\upsilon \succ t : \tau}{\lambda x.t : \upsilon \to \tau} \qquad \frac{f : \upsilon \to \tau \quad a : \upsilon}{f\,a : \tau} \qquad \frac{s :: \sigma \quad x :: \sigma \succ w : \tau}{\text{let } x := s \text{ in } w : \tau} \qquad \frac{t : \tau \quad \tau \equiv \upsilon}{t : \upsilon}$$

Fig. 5. Declarative rules for type assignment

5 Type inference

We have seen a unification algorithm for types containing units of measure, and this extends to a type inference algorithm for the corresponding type system. I will only sketch the extension here; it is detailed in the previous paper [4]. Besides the new unification algorithm and the ability to quantify over group variables, no changes to type inference are required.

A *type scheme* σ is a type quantified over by some context entries of sort TY or GR. For example, $\forall\alpha\forall(\beta := \alpha).\alpha \to \beta$ corresponds to the type $\alpha \to \beta$ quantified over by $\alpha?, \beta := \alpha$. Unknown variables are universally quantified, whereas defined variables represent abbreviations that are stored in the type scheme. (For a more conventional presentation, they could be substituted out.)

Just as a type scheme quantifies over a context extension, so a statement can be conditional on an extension: if S is a statement, then so is $\Xi \succ S$, with $\Gamma \vdash (\Xi \succ S)$ iff $\Gamma, \Xi \vdash S$ (omitting some freshness-related details). Let us introduce a new statement form $t : \tau$ for type assignment, where t is a term and τ is a well-formed type. We then define the scheme assignment statement $t :: \forall\Xi.\tau \;\mapsto\; \Xi \succ (t : \tau)$.

Now let us add declarations of the form $x :: \sigma$ to the context, where $x \in \mathcal{V}_{\text{TM}}$ is a term variable and σ is a type scheme. Recall from Section 2 that we must define the validity map, to say when a property makes sense, and the interpretation map, to explain declarations as statements:

$$\mathbf{ok}_{\text{TM}}(:: \forall\Xi.\tau) \;\mapsto\; \Xi \succ (\tau \textbf{ is } \text{TY}), \qquad\qquad [\![x :: \sigma]\!]_{\text{TM}} \;\mapsto\; x :: \sigma.$$

Thanks to the latter definition, the LOOKUP rule from Section 2 can be used to assign types to variables. The other rules for type assignment statements are given in Figure 5. These rules can be converted into an algorithm that is structurally recursive on terms, building up a context along the way:

– For a term variable x, look up its type scheme in the context and expand the scheme with fresh variables to produce a type.
– For a lambda abstraction $\lambda x.t$, create a fresh unknown type variable β, add it with $x :: \beta$ to the context, then infer the type of t.
– For an application $f\,a$, infer the types of f and a, then appeal to unification to ensure f is a function whose domain corresponds to the type of a.
– For a let binding let $x := s$ in w a few steps are required:
  1. place a marker $\fatsemi$ in the context, starting a new locality;

2. infer the type τ of s;
3. generalise τ over all type variables in the locality, producing a scheme σ;
4. extend the context with the new term variable x having scheme σ; and
5. infer the type of w.

Generalisation is easy and there is no need to complicate the type inference algorithm to deal with units of measure. We can extend the initial context with constant terms that use the new types. Moreover, thanks to the refinement of Section 4.2, the algorithm copes naturally with the problematic term from Section 1, correctly inferring its most general type. Recall the example:

$$\lambda x.\ \mathrm{let}\ d := \mathrm{div}\, x\ \mathrm{in}\ (d\ \mathrm{mass}, d\ \mathrm{time}), \qquad \text{where}$$
$$\mathrm{div} :: \forall ab\ .\ \mathbb{F}\langle ab\rangle \to \mathbb{F}\langle a\rangle \to \mathbb{F}\langle b\rangle, \qquad \mathrm{mass} :: \mathbb{F}\langle \mathbf{kg}\rangle, \qquad \mathrm{time} :: \mathbb{F}\langle \mathbf{s}\rangle.$$

With the new type inference algorithm, at the crucial point where the type of d is to be generalised, we have the context, type and constraint

$$\gamma?, x : \gamma \fatsemi a?, b? \ \vdash\ \mathrm{div}\, x : \mathbb{F}\langle a\rangle \to \mathbb{F}\langle b\rangle \qquad \text{subject to } \gamma \equiv \mathbb{F}\langle ab\rangle.$$

(Here γ is an unknown fresh type variable standing in for the type of x.) If c is a fresh group variable, the constraint decomposes into two simpler constraints $\gamma \equiv_{\mathrm{TY}} \mathbb{F}\langle c\rangle \ \wedge\ c \equiv_{\mathrm{GR}} ab$, which can be solved one at a time to give the context $c?, \gamma := \mathbb{F}\langle c\rangle, x : \gamma \fatsemi a?, b := ca^{-1}$. Generalising by 'skimming off' type variables in the locality (and substituting out the definition of b) gives the principal scheme

$$c?, \gamma := \mathbb{F}\langle c\rangle, x : \gamma \ \vdash\ d : \forall a. \mathbb{F}\langle a\rangle \to \mathbb{F}\langle ca^{-1}\rangle.$$

6 Discussion

I have shown how to combine abelian group unification with syntactic unification in such a way that generalisation is straightforward. Unlike the usual Damas-Milner approach to generalisation, the structured context discipline adopted here works well with the nontrivial equational theory. The algorithms presented here solve unification problems by making gradual refinements towards a solution, so it is comparatively easy to check each step is sound and most general.

A key point is that flex-rigid equations $\alpha \equiv \tau$ cannot always be solved by instantiating α to τ, in the presence of a nontrivial equational theory. Instead, τ decomposes into a 'rigid hull' (the outer structure that α must match exactly) and a collection of subexpressions (that must match in the equational theory).

I plan to apply this technique to other equational theories and more advanced type systems. In particular, I am interested in the computational equality of dependent types [10], and Miller's 'mixed prefix' unification [11], which can be used to implement arbitrary-rank polymorphism as available in modern Haskell.

In the beginning I mentioned types indexed by integers, which form an abelian group under addition, so type inference could be implemented using the algorithm described here. However, for many purposes natural numbers are needed, so I am exploring how to solve inequalities in this setting. There are also many other algebraic structures to consider, notably rings and semirings, though unification is not often unitary for their equational theories.

In this paper we have been following the trail that Kennedy blazed, both in the representation of units of measure using a free abelian group with constants, and the observation that unification has decidable most general unifiers in this case. In order to extend the technique to less convenient type systems, we will need to deal with problems that cannot necessarily be solved on the first try. Where will we store problems that we cannot yet solve? In the context, of course!

References

[1] Buckwalter, B.: Dimensional - statically checked physical dimensions for Haskell, `http://code.google.com/p/dimensional/`

[2] Bürckert, H.J., Herold, A., Schmidt-Schauß, M.: On equational theories, unification and decidability. In: RTA '87. pp. 204–215. Springer (1987)

[3] Chen, F., Roşu, G., Venkatesan, R.P.: Rule-based analysis of dimensional safety. In: RTA '03. pp. 197–207. Springer (2003)

[4] Gundry, A., McBride, C., McKinna, J.: Type inference in context. In: MSFP '10. pp. 43–54. ACM (2010)

[5] Karr, M., Loveman III, D.B.: Incorporation of units into programming languages. Comm. ACM 21, 385–391 (1978)

[6] Kennedy, A.: Programming Languages and Dimensions. Ph.D. thesis, University of Cambridge (1996)

[7] Kennedy, A.: Type inference and equational theories. Research Report LIX/RR/96/09, École Polytechnique (1996)

[8] Kennedy, A.: Types for units-of-measure: Theory and practice. In: CEFP '09, LNCS, vol. 6299, pp. 268–305. Springer (2010)

[9] Kuan, G., MacQueen, D.: Efficient ML type inference using ranked type variables. In: ML '07. pp. 3–14. ACM (2007)

[10] McBride, C.: Dependently Typed Functional Programs and their Proofs. Ph.D. thesis, University of Edinburgh (1999)

[11] Miller, D.: Unification under a mixed prefix. J. Symbolic Computation 14(4), 321–358 (1992)

[12] Rémy, D.: Extension of ML type system with a sorted equational theory on types. Research Report RR-1766, INRIA (1992)

[13] Rittri, M.: Dimension inference under polymorphic recursion. In: FPCA '95. pp. 147–159. ACM (1995)

[14] Schabel, M.C., Watanabe, S.: Boost.Units 1.1.0, `http://www.boost.org/doc/libs/1_46_1/doc/html/boost_units.html`

[15] Syme, D.: The F# 2.0 Language Specification. Microsoft (2010)

[16] Vytiniotis, D., Peyton Jones, S., Schrijvers, T.: Let should not be generalized. In: TLDI '10. pp. 39–50. ACM (2010)

Appendix

First I give a technical lemma that illustrates explicitly the reasoning used to show generality. Subsequent proofs will not be in quite this much detail.

Lemma 5. *Suppose d is a well-formed group expression in Γ, $\gamma : \Gamma \sqsubseteq \Gamma_0$ is invertible (i.e. there exists $\gamma^{-1} : \Gamma_0 \sqsubseteq \Gamma$ such that $\gamma \cdot \gamma^{-1} \equiv \iota$ and $\gamma^{-1} \cdot \gamma \equiv \iota$), $\delta : \Delta \sqsubseteq \Delta_0$ is invertible and $\iota \cdot \gamma \equiv \delta \cdot \iota$. Then the following rule is admissible:*

$$\frac{\Gamma_0 \mathrel{\widehat{\sqsubseteq}} \Delta_0 \vdash \gamma d \equiv 0}{\Gamma \mathrel{\widehat{\sqsubseteq}} \Delta \vdash d \equiv 0}.$$

Proof. We assume $\Gamma_0 \mathrel{\widehat{\sqsubseteq}} \Delta_0 \vdash \gamma d \equiv 0$. Now $\delta^{-1} \cdot \iota \cdot \gamma \equiv \iota : \Gamma \sqsubseteq \Delta$, and $\Delta_0 \vdash \gamma d \equiv 0$ so $\Delta \vdash (\delta^{-1} \cdot \iota)(\gamma d) \equiv 0$ and hence $\Delta \vdash d \equiv 0$. For generality, let $\theta : \Gamma \sqsubseteq \Theta$ be such that $\Theta \vdash \theta d \equiv 0$. Then $\theta \cdot \gamma^{-1} : \Gamma_0 \sqsubseteq \Theta$ and $\Theta \vdash (\theta \cdot \gamma^{-1})(\gamma d) \equiv 0$ so by the assumption of minimality, there is a substitution $\zeta : \Delta_0 \sqsubseteq \Theta$ such that $\theta \cdot \gamma^{-1} \equiv \zeta \cdot \iota$. Now $\zeta \cdot \delta \cdot \iota \equiv \zeta \cdot \iota \cdot \gamma \equiv \theta \cdot \gamma^{-1} \cdot \gamma \equiv \theta$, so $\zeta \cdot \delta : \Delta \sqsubseteq \Theta$ is the required substitution. □

Lemma 1 (Soundness and generality of abelian group unification). *If unification succeeds with $\Gamma, [\Psi] \twoheadrightarrow \Delta \vdash d$* **id***, then $\mathcal{V}_{\mathrm{TY}}(\Gamma, \Psi) = \mathcal{V}_{\mathrm{TY}}(\Delta)$, $\mathcal{V}_{\mathrm{GR}}(\Gamma, \Psi) \subseteq \mathcal{V}_{\mathrm{GR}}(\Delta)$ and it gives a most general solution $\Gamma, \Psi \mathrel{\widehat{\sqsubseteq}} \Delta \vdash d \equiv_{\mathrm{GR}} 0$.*

Proof. We proceed by structural induction on derivations. For soundness, it is easy to verify that $\Gamma, \Psi \sqsubseteq \Delta$ and $\Delta \vdash d \equiv_{\mathrm{GR}} 0$. Let us consider generality for each rule in Figure 3:

Trivial and Ignore are straightforward to check.

Reduce, Collect and Expand satisfy the generality condition by Lemma 5.

For Define, suppose $\theta : \Gamma, a?, \Psi \sqsubseteq \Theta$ is such that $\Theta \vdash \theta(na + ne) \equiv 0$. Then $\Theta \vdash n(\theta(a + e)) \equiv 0$ and hence $\Theta \vdash \theta(a + e) \equiv 0$, since we are working in the **free** abelian group. Thus $\Theta \vdash \theta a \equiv \theta(-e)$ and so $\theta : \Gamma, \Psi, a {:=} {-e} \sqsubseteq \Theta$.

Finally, we consider Repossess. If Ψ is empty then the result is straightforward. Otherwise, it contains a single unknown variable β; let $d \equiv n\beta + e$. Suppose $\theta : \Gamma \fatsemi \beta? \sqsubseteq \Theta \fatsemi \Phi$ is such that $\Theta \fatsemi \Phi \vdash \theta(n\beta + e)$ **id**. Then $\Theta \fatsemi \Phi \vdash n(\theta\beta) \equiv -(\theta e)$ but θe is defined over Θ so $\theta\beta$ must be defined over Θ (by substituting out definitions in Φ if necessary). Thus $\theta : \Gamma, \beta? \sqsubseteq \Theta$ and the result follows by the inductive hypothesis. □

Lemma 2 (Completeness of abelian group unification).
*If d is a well-formed group expression in Γ, and there is some $\theta : \Gamma \sqsubseteq \Theta$ such that $\Theta \vdash \theta d \equiv_{\mathrm{GR}} 0$, then the algorithm produces Δ such that $\Gamma, [\mathcal{E}] \twoheadrightarrow \Delta \vdash d$ **id**.*

Proof. First, let us establish termination of the rules when viewed as an algorithm, where hypotheses correspond to recursive calls. Termination is by the lexicographic order on the total length of the context (including Ψ), the maximum coefficient of a variable in the expression being unified, and the length of the first part of the context (excluding Ψ). Only the Reduce and Collect rules do not decrease the total length on recursive calls; moreover, Reduce decreases the maximum coefficient of a variable (to n) and Collect decreases the length of the first part of the context. Note that the final result may be longer than the original context, due to Reduce.

Since the algorithm terminates, we are entitled to reason about completeness by induction on the call graph. By inspection of the rules, we observe that only two possible cases are not covered: either d is a non-zero constant expression, or d contains exactly one variable a, and the coefficient of a does not divide the coefficients of the constants. In either case, there are no possible solutions of the unification problem $d \equiv_{\mathrm{GR}} 0$.

Finally, we note that each rule preserves solutions: that is, if the initial problem (conclusion of the rule) has a solution then the rewritten problem (hypothesis of the rule) must also have a solution. Hence failure of the algorithm indicates that the original problem had no solutions. □

Lemma 3 (Soundness and generality of type unification).

(a) If type unification succeeds with $\Gamma \twoheadrightarrow \Delta \vdash \tau \equiv \upsilon$, then $\mathcal{V}_{\mathrm{TY}}(\Gamma) = \mathcal{V}_{\mathrm{TY}}(\Delta)$, $\mathcal{V}_{\mathrm{GR}}(\Gamma) \subseteq \mathcal{V}_{\mathrm{GR}}(\Delta)$ and it gives a most general solution $\Gamma \mathrel{\widehat{\sqsubseteq}} \Delta \vdash \tau \equiv \upsilon$.
(b) Correspondingly, if $\Gamma \,|\, \Xi \twoheadrightarrow \Delta \vdash \alpha \equiv \tau$, then $\mathcal{V}_{\mathrm{TY}}(\Gamma, \Xi) = \mathcal{V}_{\mathrm{TY}}(\Delta)$, $\mathcal{V}_{\mathrm{GR}}(\Gamma) \subseteq \mathcal{V}_{\mathrm{GR}}(\Delta)$ and $\Gamma, \Xi \mathrel{\widehat{\sqsubseteq}} \Delta \vdash \alpha \equiv \tau$.

Proof. We proceed by induction on the structure of derivations, as discussed in the previous paper [4, Lemma 5]. There are four new rules:

For the Unit rule, the result follows from Lemma 1.

For DependS′, the required property is identical to the inductive hypothesis.

For the Solve′ and DependS″ rules, we use the Optimist's lemma [4, Lemma 4], which states (more formally) that the minimal solution to a conjunction of problems is found by 'optimistically' solving the first problem in the original context, then solving the second problem in the resulting context. These rules fit the pattern as solutions to $\alpha \equiv \tau\langle\vec{e}\rangle$ are the same as solutions to $\alpha \equiv_{\mathrm{TY}} \tau\langle\vec{\beta}\rangle \wedge \vec{\beta} \equiv_{\mathrm{GR}} \vec{e}$ up to the equational theory.

Apart from the new rules, the argument for generality of the Repossess rule is now more subtle, as group variables may appear in the context suffix Ξ

being moved into the previous locality. However, the invariant we established in Section 4.2 means that Ξ contains only type variables and unknown group variables that appear on their own in types. Any solution to the flex-rigid unification problem must move the entirety of Ξ past the marker, because all the group variables are genuine dependencies. $\square$

Lemma 4 (Completeness of type unification).

(a) If the types υ and τ are well-formed in Γ and there is some $\theta : \Gamma \sqsubseteq \Theta$ such that $\Theta \vdash \theta\upsilon \equiv \theta\tau$, then unification produces Δ such that $\Gamma \twoheadrightarrow \Delta \vdash \upsilon \equiv \tau$.

(b) Moreover, if $\theta : \Gamma, \Xi \sqsubseteq \Theta$ is such that $\Theta \vdash \theta\alpha \equiv \theta\tau$ and the following conditions are satisfied:

$\alpha \in \mathcal{V}_{\mathrm{TY}}(\Gamma)$, $\quad\tau$ is not a variable,

*$\Gamma, \Xi \vdash \tau$ **is** TY, $\quad\Xi$ contains only type or group variable declarations*

$\beta \in \mathcal{V}_{T}(\Xi) \Rightarrow \beta \in \mathrm{FV}_{T}(\tau, \Xi)$;

then there is some context Δ such that $\Gamma \mid \Xi \twoheadrightarrow \Delta \vdash \alpha \equiv \tau$.

Proof. First we establish that the system terminates, if viewed as an algorithm with inputs Γ (and Ξ), υ (or α) and τ, giving output Δ. The 'unify' judgments terminate because each recursive call removes a type variable from the context, decomposes the types or removes a group variable. The 'solve' judgments either shorten the whole context or the part of the context before the bar. Note that the SOLVE′ and DEPENDS″ rules may add group variables, but at least one type variable will be removed from the context before EXPANDS calls 'unify' again. Only the DECOMPOSE rule makes more than one recursive call to type unification, and it decomposes types so it does not matter that the intermediate context may have more group variables.

Now we proceed by structural induction on the call graph, observing that each rule in turn preserves solutions, and that all (potentially solvable) cases are covered. The only cases not covered are rigid-rigid mismatches (e.g. unifying $\upsilon \rightarrow \tau$ with $\mathbb{F}\langle d\rangle$) and the flex-rigid problem $\alpha \equiv \tau$ in context $\Gamma, \alpha D \mid \Xi$ where $\alpha \in \mathrm{FV}_{\mathrm{TY}}(\tau, \Xi)$. The latter has no solutions because the occur-check fails (if α is in Ξ then the conditions of the lemma ensure τ depends on it). For more details, see the previous paper [4, Lemma 7]. The algorithm may also fail in abelian group unification, for which completeness is by Lemma 2. $\square$