Type Inference for Units of Measure

Research Paper

Adam Gundry

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

Abstract. Units of measure are an example of a type system extension involving a nontrivial equational theory. Type inference for such an extension requires equational unification. 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 this Haskell function, traditionally of type $\mathit{Float} \to \mathit{Float}$:

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

Kennedy [5–7] teaches us how to check units of measure: with velocity in $\mathbf{m} \cdot \mathbf{s}^{-1}$ and acceleration in $\mathbf{m} \cdot \mathbf{s}^{-2}$, the system could infer the type $\mathit{Float}\langle\mathbf{s}\rangle \to \mathit{Float}\langle\mathbf{m}\rangle$. Type inference relies on unification, but units need a more liberal equational theory than syntactic equality, as $\mathbf{m} \cdot \mathbf{s}^{-1} \cdot \mathbf{s}$ should mean the same thing as $\mathbf{m}$. Kennedy uses the theory of abelian groups. He has introduced units of measure with polymorphism into the functional programming language F# [15].

The Damas-Milner type inference algorithm relies on unification for variable specialisation, and on dependency analysis for let-generalisation. Using the occurs check to identify generalisable variables (those that are free in the type but not the typing environment) is problematic for the equational theory of abelian groups, as **variable occurrence does not imply variable dependency**. 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. This paper's contribution is an analysis of dependency in nontrivial equational theories, and the theory of abelian groups in particular, that exposes and resolves the difficulties with generalisation that they present.

Kennedy gives the example [7, p. 292] (in slightly different notation: here $\mathbb{F}\langle d\rangle$ is a type of numbers with units d, defined in Section 4.1)

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

If one adds constraint solving for units to Damas-Milner with the usual occurrence-based let-generalisation rule, the resulting algorithm fails to infer a type for this term, because polymorphism is lost: d is given the monotype $\mathbb{F}\langle a\rangle \to \mathbb{F}\langle c \cdot a^{-1}\rangle$ where a and c are unification variables, and a cannot unify with $\mathbf{kg}$ and $\mathbf{s}$. However, if d is given its principal type scheme $\forall a.\mathbb{F}\langle a\rangle \to \mathbb{F}\langle c \cdot a^{-1}\rangle$, then the term has type $\mathbb{F}\langle c\rangle \to \mathbb{F}\langle c \cdot \mathbf{kg}^{-1}\rangle \times \mathbb{F}\langle c \cdot \mathbf{s}^{-1}\rangle$, as described in Section 5.

The difficulty is that the algorithm fails to assign principal type schemes to open terms (even without a Haskell-style monomorphism restriction) because of the nontrivial equational theory on types. Type systems with different notions of polymorphism (such as higher-rank or intersection types) do not support the relevant equational theory or provide a direct solution.

One possible way around this difficulty is Kennedy's notion of *generaliser*, "a substitution that 'reveals' the polymorphism available under a given type environment" [6, 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 previous work [4], McBride, McKinna and I described a rationalisation of syntactic unification and Hindley-Milner type inference problem solving that provides a more refined account of dependency analysis. Term and type variables live in a dependency-ordered context. Problems are solved in small steps, each of which is most general (for unification) and involves minimal extra dependency. This makes let-generalisation particularly easy: we simply 'skim off' generalisable type variables from the end of the context, as nothing can depend on them.

In this paper I extend the unification algorithm (and hence type inference) to support the theory of abelian groups. Unification that does not handle dependencies will show up as the source of the difficulty described above, leading to a 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 the loss of polymorphism can be avoided 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 it can be reduced to linear integer constraint solving, standard constraint solving algorithms do not track dependencies.

This development is an example of the value of our approach to problem solving: 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; perhaps a better account of generalisation will help us decide.

In the sequel, I describe a framework for contextual problem solving algorithms (Section 2) and a new algorithm for abelian group unification (Section 3). Using this, I extend type unification to handle units of measure (Section 4). I sketch the type inference algorithm (Section 5), then conclude with related work and future directions (Section 6). A Haskell implementation of the algorithms is available online, along with a technical report containing full proofs [3].

Acknowledgments I thank Conor McBride and James McKinna for all their help and fruitful collaboration. Attendees at TFP 2011 provided valuable feedback, as did the anonymous reviewers of previous versions. This work was supported by the Microsoft Research PhD Scholarship Programme.

2 Problem solving in context

First I describe a framework for unification and type inference problem-solving algorithms that makes it is easy to prove they deliver most general results. Problem solving means evolving the initial context into one in which the problem is solved, given an 'information increase' relation between contexts that captures legitimate steps towards a solution. A strategy based on minimal sound information increases is easily seen to be most general if all judgments are 'stable' (once a judgment holds in a context, subsequent information increases do not invalidate it). The setup here induces stability by construction.

The syntax of sorts, contexts and expressions is given in Figure 1. Variable *sorts* are TY for types, GR for groups and TM for terms. Informally, a well-formed context is a list of variable declarations in dependency order. Unlike a traditional typing environment, type variables appear in the context structure.

Sorts	$T ::= \mathrm{TY} \mid \mathrm{GR} \mid \mathrm{TM}$
Contexts	$\Gamma, \Delta, \Theta ::= \mathcal{E} \mid \Gamma, x\, D_T \mid \Gamma \fatsemi$
Variables (any sort)	$x \in \mathcal{V}_T$
Type variables	$\alpha, \beta, \gamma \in \mathcal{V}_{\mathrm{TY}}$
Group variables	$a, b, c \in \mathcal{V}_{\mathrm{GR}}$
Properties (sort TY or GR)	$D_{\mathrm{TY}}, D_{\mathrm{GR}} ::= \,? \mid (:= e)$
Expressions (sort TY or GR)	$d, e, f ::= \ldots$
Types (expressions of sort TY)	$\tau, \upsilon ::= \alpha \mid \tau \to \upsilon$

Fig. 1. Syntax

More precisely, a *context* is a list of *declarations* $x\,D_T$ (giving each variable $x \in \mathcal{V}_T$ a *property* D_T) and $\fatsemi$ markers (to be discussed in Section 2.1). The empty context is written $\mathcal{E}$ or omitted. A type or group variable α is either unknown ($\alpha ?$) or defined ($\alpha := e$). Expressions of sort GR are given in Section 3; sort TM is introduced in Section 5. As is conventional, I assume a countable supply of fresh variable names of each sort that can be generated as required.

Statements are assertions that can be judged in contexts. The judgment $\Gamma \vdash S$ means statement S holds in context Γ. For now, the syntax of statements is:

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

I regard $\Gamma \vdash \cdot \equiv_T \cdot$ as a partial equivalence relation, reflexive on *well-formed* expressions, so e **is** T means $e \equiv_T e$. 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.

Figure 2 gives rules to construct a valid context and interpret variables in the context. To avoid nonsensical contexts such as $\alpha := \alpha$, not all properties are valid extensions to a context. A *validity map* $\mathbf{ok}_T$ from properties to statements gives a statement that must hold in Γ for a property D_T to be a valid extension $\Gamma, x\,D_T$. Declarations are given meaning by an *interpretation map* $[\![\cdot]\!]_T$ from declarations of sort T to statements, used by the Lookup rule. Crucially, this is the only rule that uses a declaration in the context to prove a statement. Information increase preserves applicability of Lookup; stability then follows inductively. Figure 2 also gives rules for conjunctions and equivalence, omitting the (fixed) context.

The rules ensure that a valid context has no duplicated variables. I write $\mathcal{V}_T(\Gamma)$ for the set of variables **bound** in the context Γ, which is different from the set of **free** variables in a context suffix or expression X, written $\mathrm{FV}_T(X)$. Thus $\mathcal{V}_{\mathrm{TY}}(\Gamma_0) = \{\alpha, \beta\}$ and $\mathrm{FV}_{\mathrm{TY}}(\beta := \alpha) = \{\alpha\}$. A valid context has no free variables.

For type and group variables, the ? property is always valid, but definitions must be well-formed expressions. A declaration means that the corresponding variable

$$\boxed{\Gamma \vdash S}$$

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

$$\textsc{Lookup}\;\frac{x\,D_T \in \Gamma}{\Gamma \vdash [\![x\,D_T]\!]_T} \qquad \frac{S \quad S'}{S \wedge S'}$$

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

Fig. 2. Rules for context validity, lookup, conjunction and equivalence

is a well-formed expression and is equal to its definition (in any):

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

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.

2.1 Solving problems by increasing information

What does it mean to increase information in a context? 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 sorts $T \in \{\mathrm{TY}, \mathrm{GR}\}$. It 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. 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 $\forall T \in \{\mathrm{TY}, \mathrm{GR}\}.\ \forall x \in \mathcal{V}_T(\Gamma).\ \Delta \vdash \delta x \equiv_T \theta x$.

Substitutions describe moves from one context to another. For example, the identity function is a substitution from $\alpha ?, \beta := \alpha \to \alpha$ to $\alpha ?, \beta ?$. However, a legitimate solution step must also preserve information: the interpretation $[\![x D]\!]_T$ of a context entry $x\, D_T$ from the old context must hold in the new context (under the substitution). Thus the identity function is not an information increase between these contexts, as $\beta \equiv_{\mathrm{TY}} \alpha \to \alpha$ does not hold in the new context.

Information increases must also respect the dependency order in the context. *Localities* within the context are delimited 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 variable less local (moving it right to left of a separator) reduces the ability to generalise over it, so should be done only when essential for solving the problem. Making a variable more local (moving it left to right) is never permissible.

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

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

A substitution δ from Γ to Δ is an *information increase*, written $\delta : \Gamma \sqsubseteq \Delta$, if for all $n \in \mathbb{N}$ with $x\, D_T \in \Gamma \lfloor n$, we have that $\Delta \lfloor n$ is defined and $\Delta \lfloor n \vdash \delta [\![x D]\!]_T$. 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 declarations in a locality of Γ hold as equations in the corresponding locality of Δ. An example increase is $\alpha ? \fatsemi \beta ? \sqsubseteq \beta ?, \alpha := \beta \to \beta \fatsemi$, but on the other hand, $\beta ?, \alpha := \beta \to \beta \fatsemi \not\sqsubseteq \alpha ? \fatsemi \beta ?$ as the first locality of the new context does not support β **is** TY or $\alpha \equiv \beta \to \beta$.

The information increase relation can be used to define solutions to problems. 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 $\Delta \vdash \delta d \equiv_T \delta e$. This solution is *minimal* or *most general* if every other solution $\theta : \Gamma \sqsubseteq \Theta$ factors through it, i.e. there exists $\zeta : \Delta \sqsubseteq \Theta$ such that $\theta \equiv \zeta \circ \delta$ (with $\circ$ the usual composition of substitutions). 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 $\alpha \equiv_{\text{TY}} \beta \rightarrow \beta$ has $\alpha? \fatsemi \beta? \mathrel{\widehat{\sqsubseteq}} \beta?, \alpha := \beta \rightarrow \beta \fatsemi \vdash \alpha \equiv_{\text{TY}} \beta \rightarrow \beta$ a minimal solution. The algorithms will find solutions using the identity substitution, but the solutions are minimal with respect to arbitrary substitutions.

A statement S is *stable* if it is preserved by information increase, that is, if $\Gamma \vdash S$ and $\delta : \Gamma \sqsubseteq \Delta$ then $\Delta \vdash \delta S$. Since information increases preserve the interpretations of variable declarations, and only the LOOKUP rule is used to extract information from the context, stability holds by construction. Problems expressed as stable statements can be solved using a minimal commitment strategy (McBride's "optimistic optimisation" [4, 9]) to give minimal solutions.

3 Abelian group unification

I now consider abelian group unification problems in the framework of Section 2. A *group expression (with constants in K)* is an expression of sort GR given by

$$d, e, f \ ::= \ a \ | \ k \ | \ 1 \ | \ d \cdot d \ | \ d^{-1},$$

where $a \in \mathcal{V}_{\text{GR}}$ and $k \in K$. The rules for equivalence in Figure 3 extend those in Figure 2 by reflexivity and congruence (making group expressions well-formed), plus the four group axioms of commutativity, associativity, inverses and identity.

Let d^n mean d multiplied by itself n times and d^{-n} mean $(d^n)^{-1}$. Group expressions have a normal form $\prod_i {d_i}^{n_i}$ 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 $a \cdot a \cdot b \cdot 1 \cdot b \cdot a$ has normal form $a^3 \cdot b^2$ (if $a < b$ in the order).

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

$$\frac{}{1 \equiv_{\text{GR}} 1} \qquad \frac{}{k \equiv_{\text{GR}} k}\, k \in K \qquad \frac{d \equiv_{\text{GR}} e}{d^{-1} \equiv_{\text{GR}} e^{-1}} \qquad \frac{d_0 \equiv_{\text{GR}} e_0 \quad d_1 \equiv_{\text{GR}} e_1}{d_0 \cdot d_1 \equiv_{\text{GR}} e_0 \cdot e_1}$$

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

$$\frac{d \ \textbf{is}\ \text{GR}}{d \cdot (d^{-1}) \equiv_{\text{GR}} 1} \qquad \frac{d \ \textbf{is}\ \text{GR}}{d \cdot 1 \equiv_{\text{GR}} d}$$

Fig. 3. Declarative rules for group expression equivalence

Consider the equation $a^3 \cdot b^2 \equiv 1$ in the context $a?, b?$. As 2 does not divide 3, b cannot be defined to solve this equation, but the problem can be simplified by $b := c \cdot a^{-1}$ where c is a fresh variable. This leaves $a \cdot c^2 \equiv 1$ in the context $a?, c?$, which is solved by rearranging and taking $a := c^{-2}$ to give $c?, a := c^{-2}, b := c \cdot a^{-1}$.

More generally, when solving such an equation, one can ask whether a variable has the largest power, and if not, reduce the other powers by it to simplify the problem. Some notation is in order. Suppose $d \equiv \prod_i {d_i}^{n_i}$ and define:

$$\begin{array}{rll} \text{maxpow(d)} = \max\{|n_i| \mid d_i \text{ is a variable}\}, & \text{highest absolute variable power;} \\ Q_n(d) = \prod_i {d_i}^{(n_i \text{ quot } n)}, & \text{quotient by } n \text{ of every power;} \\ R_n(d) = \prod_i {d_i}^{(n_i \text{ rem } n)}, & \text{remainder by } n \text{ of every power;} \end{array}$$

where $\cdot \text{quot} \cdot$ is integer division truncated towards zero, and $\cdot \text{rem} \cdot$ is the corresponding remainder. The important point is that $d \equiv_{\text{GR}} (Q_n(d))^n \cdot R_n(d)$.

3.1 The abelian group unification algorithm

In this section, I give a new algorithm for unification problems $d \equiv_{\text{GR}} e$. The inverse operation means it suffices to solve problems $d' \equiv_{\text{GR}} 1$, written d' **id**.

Figure 4 shows the algorithm presented as a collection of inference rules. Given as input a context Γ, Ψ and a group expression d, the judgment $\Gamma, [\Psi] \twoheadrightarrow \Delta \vdash d\ \textbf{id}$ means that the algorithm outputs the context Δ such that $\Delta \vdash d\ \textbf{id}$. 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.

So how does the algorithm work? If the problem is 1 **id**, then it is TRIVIAL. Otherwise, it moves 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 a defined variable is encountered, it is substituted 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 power 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 power.

The interesting cases arise when an unknown variable a, that occurs in the problem, in reached. This is written $(a^n \cdot e)$ **id**, always meaning that $a \notin \text{FV}_{\text{GR}}(e)$. Suppose the normal form of e is $\prod_i {e_i}^{n_i}$. There are four possibilities, either:

(1) n divides n_i for all i;
(2) e has at least one variable and $|n| \leq$ maxpow(e) but case (1) does not apply;
(3) e has at least one variable and $|n| >$ maxpow(e); or
(4) e has no variables.

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

$$\textsc{Trivial}\ \frac{}{\Gamma, [\mathcal{E}] \twoheadrightarrow \Gamma \vdash 1 \ \mathbf{id}} \qquad \textsc{Repossess}\ \frac{\Gamma, [\Psi] \twoheadrightarrow \Delta \vdash d \ \mathbf{id}}{\Gamma {\fatsemi}, [\Psi] \twoheadrightarrow \Delta {\fatsemi} \vdash d \ \mathbf{id}}$$

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

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

$$\textsc{Define}\ \frac{}{\Gamma, a?, [\Psi] \twoheadrightarrow \Gamma, \Psi, a {:=} f^{-1} \vdash a^n \cdot f^n \ \mathbf{id}}\ n \neq 0$$

$$\textsc{Reduce}\ \frac{\Gamma, \Psi, [b?] \twoheadrightarrow \Delta \vdash b^n \cdot R_n(e) \ \mathbf{id}}{\Gamma, a?, [\Psi] \twoheadrightarrow \Delta, a {:=} b \cdot Q_n(e)^{-1} \vdash a^n \cdot e \ \mathbf{id}}\ |n| \leqslant \mathrm{maxpow(e)}, b \text{ fresh}$$

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

Fig. 4. Algorithmic rules for abelian group unification

Case (1). If n divides n_i for all i, then there is some f such that $e \equiv f^n$. The rule Define applies and sets $a := f^{-1}$ to give $a^n \cdot e \equiv a^n \cdot f^n \equiv f^{-n} \cdot f^n \equiv 1$. This is clearly a solution, and it is most general for the free abelian group.

Case (2). If not, and $|n| \leq \mathrm{maxpow(e)}$, then the Reduce rule applies and simplifies the problem by reducing the powers modulo n. Recall that we have $e \equiv Q_n(e)^n \cdot R_n(e)$ where $Q_n(e)$ takes the quotient by n of the powers in e. Hence, generating a fresh variable b and defining $a := b \cdot Q_n(e)^{-1}$ gives

$$a^n \cdot e \equiv \left(b \cdot Q_n(e)^{-1}\right)^n \cdot e \equiv b^n \cdot (e \cdot Q_n(e)^{-n}) \equiv b^n \cdot R_n(e).$$

Case (3). Suppose $|n| > \mathrm{maxpow(e)}$, so neither of the two previous cases apply, but there is at least one variable in e. Now n is the largest power of a variable, so reducing the powers modulo n would leave them unchanged. Instead, the Collect rule moves a further back in the context. This rule maintains the invariant that Ψ contains only the variable with the largest power, if any; the invariant also guarantees that Ψ will be empty when the rule applies.

Case (4). If e has no variables and n does not divide the powers of the constants in e, then $a^n \cdot e \equiv_{\mathrm{GR}} 1$ has no solution in the free abelian group.

3.2 Correctness of the abelian group unification algorithm

I only sketch correctness proofs here; more details are in the technical report [3].

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 $\Gamma, \Psi \;\widehat{\sqsubseteq}\; \Delta \vdash d \equiv_{\mathrm{GR}} 1$ is a most general solution.*

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 power, 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 $\Theta \vdash \theta d \equiv_{\mathrm{GR}} 1$, then the algorithm produces Δ such that $\Gamma, [\mathcal{E}] \twoheadrightarrow \Delta \vdash d$* **id**.

Proof (Sketch). A suitable metric shows that the algorithm terminates, so reasoning by structural induction on the call graph is justified. Completeness is by the fact that the rules cover all solvable cases and preserve solutions: if no rule applies then the original problem can have had no solutions. This occurs if a non-unit constant is equated to 1 or there is only one variable and its power does not divide the power of one of the constants (e.g. $(a^2 \cdot k)$ **id**). □

4 Unification for types with units of measure

Having developed a unification algorithm for abelian groups, I now extend type unification to support units of measure, calling group unification from Section 3 as a subroutine to solve constraints on units. The unification algorithm from my previous work [4] is shown in Figure 5. Again the rules are syntax directed and lead directly to an implementation. There are two kinds of rules:

– '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 $\Gamma \,|\, \Xi \twoheadrightarrow \Delta \vdash \alpha \equiv \tau$ 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.

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

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. 5. Original algorithmic rules for type unification

The algorithm starts by applying the Decompose rule to split up rigid-rigid problems into subproblems and solving them sequentially. If a flex-flex problem $\alpha \equiv \beta$ is reached, the context is searched for α and β, moving past other entries with Ignore or Skip. When a variable is found, the problem is either ignored by Idle if trivial, solved by Define, or simplified by Expand.

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) is accumulated. These must be moved back in the context until DefineS (solve α with τ) or ExpandS (substitute out α) applies. Note the occurs 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 $\alpha \equiv \beta \rightarrow \gamma$. This is a flex-rigid problem so 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 $\beta?, \gamma?, \alpha := \beta \rightarrow \gamma \fatsemi$.

4.1 Units of measure as an abelian group

A *unit (of measure)* is a group expression with constants in a set of base units. In practice, the programmer could choose the base units and write conversion functions between them, often just as multiplicative constants, e.g. $2.2\langle\mathbf{lbs}\,/\,\mathbf{kg}\rangle$. The language of types is extended 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 type unification rule that invokes abelian group unification:

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

Now suppose the algorithm is used to solve $\mathbb{F}\langle b \cdot c\rangle \to \alpha \equiv \mathbb{F}\langle b\rangle \to \mathbb{F}\langle c\rangle$ in the context $b?, \alpha?, c?$. First the constraint $\mathbb{F}\langle b \cdot c\rangle \equiv \mathbb{F}\langle b\rangle$ is reduced to $b \cdot c \equiv_{\text{GR}} b$ 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 $b?, c := 1, \alpha := \mathbb{F}\langle c\rangle$.

Does Figure 5 extended with the Unit rule give a correct unification algorithm for the extended type system? It should be sound and complete, as the new algorithmic rule corresponds directly to the declarative rule, but generality fails.

4.2 Loss of generality and how to retain it

Suppose we seek $\alpha \equiv \mathbb{F}\langle b_0 \cdot b_1\rangle$ 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 $b_0?, b_1?, \alpha := \mathbb{F}\langle b_0 \cdot b_1\rangle \fatsemi$. However, a more general solution exists, namely $c?, \alpha := \mathbb{F}\langle c\rangle \fatsemi b_0?, b_1 := c \cdot {b_0}^{-1}$, where c is a fresh group variable and b_0 is still local. Why did the algorithm fail to find this?

The trouble is that, when solving a flex-rigid constraint, the variable need not be **syntactically** equal to the type: units need be equal only up to the theory of abelian groups. The property that equivalent expressions have identical free variables[2] holds for the syntactic theory and some other useful theories [12] but does not hold for groups. For example, the equation $a \cdot a^{-1} \equiv_{\text{GR}} 1$ has a free on the left but not the right. Thus variable occurrence does not imply dependency. The occurs check performed by the unification algorithm is overly syntactic.

To solve this, a flex-rigid constraint can be decomposed into a constraint on types, 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. An anonymous reviewer observes that similar techniques are used for type inference in annotated type systems [11, §5.3.2].

[2] This property is sometimes called *regularity* in the literature, but I 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.

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

The rules from Figure 5 can be modified 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 versions. (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 $\rho\langle \vec{a} \rangle = \mathbb{F}\langle a_0 \rangle \to \mathbb{F}\langle a_1 \rangle$.)

Solve′

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

DependS′

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

DependS″

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

Vectors of equations are solved one at a time, threading the context:

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

4.3 Correctness of type unification

With the above refinement, type unification gives most general results. As before, details of proofs are in the technical report [3].

Lemma 3 (Soundness and generality of type unification).

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

(b) Correspondingly, if $\Gamma \mid \Xi \twoheadrightarrow \Delta \vdash \alpha \equiv \tau$, *then* $\mathcal{V}_{\text{TY}}(\Gamma, \Xi) = \mathcal{V}_{\text{TY}}(\Delta)$, $\mathcal{V}_{\text{GR}}(\Gamma) \subseteq \mathcal{V}_{\text{GR}}(\Delta)$ *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:

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

$\Gamma, \Xi \vdash \tau$ ***is*** TY, *Ξ 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, first 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 $\upsilon \to \tau$ with $\mathbb{F}\langle d \rangle$) and occurs-check failures (such as unifying α with $\alpha \to \alpha$), neither of which have any solutions. $\square$

5 Type inference

I have given a unification algorithm for types, and this extends to a type inference algorithm for the corresponding type system. The extension is detailed in my previous paper [4] so for space reasons I will only sketch it here. Besides the new types, new unification algorithm and the ability to quantify over group variables, no changes to the core type inference algorithm are required.

A *type scheme* $\sigma ::= \forall \Xi . \tau$ is a type τ quantified over by a list Ξ of variable declarations 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, while defined variables represent abbreviations 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, so is $\Xi \succ S$, with $\Gamma \vdash (\Xi \succ S)$ if $\Gamma, \Xi \vdash S$ (omitting some details). I introduce a new statement form $t : \tau$ for type assignment, where t is a term and τ is a well-formed type, then define a new statement for scheme assignment by $t :: \forall \Xi . \tau \;\mapsto\; \Xi \succ (t : \tau)$.

Now term variables $y \in \mathcal{V}_{\mathrm{TM}}$ can be assigned properties of the form $D_{\mathrm{TM}} ::= (:: \sigma)$. Extending the definitions in Section 2, the validity map says a scheme can be assigned if it is well-formed, and the interpretation map says a declaration in the context makes the corresponding scheme assignment statement hold:

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

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 6, adapted from the usual Hindley-Milner rules.

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

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

Fig. 6. Declarative rules for type assignment

These rules can be converted into an algorithm that is structurally recursive on terms, increasing information in the context along the way:

- For a term variable y, look up its type scheme in the context and specialise the scheme with fresh variables to produce a type.
- For a lambda abstraction $\lambda y.t$, create a fresh unknown type variable β, add it with $y :: \beta$ 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 $\text{let } y := s \text{ in } w$ 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 $y :: \sigma$ and 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. The initial context can be extended 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:

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

At the crucial point when the type of d is being inferred, the situation is

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

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

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

which is principal. Type inference for the whole term succeeds, giving the type

$$\mathbb{F}\langle c\rangle \to \mathbb{F}\langle c \cdot \mathbf{kg}^{-1}\rangle \times \mathbb{F}\langle c \cdot \mathbf{s}^{-1}\rangle.$$

6 Discussion

I have shown how to combine abelian group unification with syntactic unification while carefully tracking dependencies in a structured context, so generalisation is straightforward. The algorithms presented here solve unification problems by making gradual steps towards a solution, and it is comparatively easy to check that 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, given a nontrivial equational theory. Instead, τ decomposes into a 'hull' (the outer structure that α must match exactly) and a collection of constraints in the equational theory.

6.1 Related work

Many authors have proposed designs for systems of units of measure. I have followed Kennedy's design, using integer powers, so units form an abelian group. Some authors use rational powers (giving a vector space), including Rittri [13], who discusses the merits of both approaches. Chen et al. [2] give a useful overview of work on units, 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 [8]) 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.

6.2 Future directions

This technique can be applied to other equational theories and more advanced type systems. In particular, Miller's 'mixed prefix' unification [10] works well in this setting, as does the computational equality of dependent types [9].

Types indexed by integers form an abelian group under addition, so type inference could be implemented as described here. However, for many purposes inequalities are needed, so I am exploring how to solve them in this setting. There are also many other algebraic structures to consider, notably rings and semirings, though unification for their equational theories is often harder.

In this paper I 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. To extend the technique to less convenient type systems, I will need to deal with problems that cannot necessarily be solved on the first attempt. The contextual discipline described here provides a good foundation for developing suitable algorithms: progress through the problem is represented using the context structure, which can be extended to record postponed problems.

References

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

[2] Chen, F., Roşu, G., Venkatesan, R.P.: Rule-based analysis of dimensional safety. In: Nieuwenhuis, R. (ed.) Rewriting Techniques and Applications (RTA '03). LNCS, vol. 2706, pp. 197–207. Springer (2003)

[3] Gundry, A.: Type inference for units of measure. Technical report (2011), `http://personal.cis.strath.ac.uk/~adam/units-of-measure/`

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

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

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

[7] Kennedy, A.: Types for units-of-measure: Theory and practice. In: Horváth, Z., Plasmeijer, R., Zsók, V. (eds.) Central European Functional Programming (CEFP '09), LNCS, vol. 6299, pp. 268–305. Springer (2010)

[8] Kuan, G., MacQueen, D.: Efficient ML type inference using ranked type variables. In: Russo, C.V., Dreyer, D. (eds.) ML '07. pp. 3–14. ACM (2007)

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

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

[11] Nielson, F., Nielson, H.R., Hankin, C.: Principles of Program Analysis. Springer (1999)

[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: Functional Programming and Computer Architecture (FPCA '95). pp. 147–159. ACM

[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), `http://research.microsoft.com/apps/pubs/default.aspx?id=79948`

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