A tutorial implementation of dynamic pattern unification

A dependently typed programming language implementation pearl

Adam Gundry Conor McBride

University of Strathclyde, Glasgow

{adam.gundry,conor.mcbride} @ strath.ac.uk

Abstract

A higher-order unification algorithm is an essential component of a dependently typed programming language implementation, and understanding its capabilities is important if dependently typed programmers are to become productive. Miller showed that, for simply typed λ-terms in the pattern fragment (where metavariables are applied to spines of distinct bound variables), unification is decidable and most general unifiers exist. We describe an algorithm for pattern unification in a full-spectrum dependent type theory with dependent pairs (Σ-types). The algorithm exploits heterogeneous equality and a novel concept of 'twin' free variables to handle dependency. Moreover, it supports dynamic management of constraints, postponing equations that fall outside the pattern fragment in case other equations make them simpler. We aim to make sense both to language implementors and users, and to this end present our algorithm as a Haskell program.

Categories and Subject Descriptors F.4.1 [*Mathematical Logic and Formal Languages*]: Mathematical Logic—Lambda calculus and related systems; F.3.3 [*Logics and Meanings of Programs*]: Studies of Program Constructs—Type structure

General Terms Algorithms, Languages, Theory

Keywords Dependent types, higher-order unification

1. Introduction

Higher-order unification is the problem of finding a substitution that unifies two λ-calculus terms. Dependently typed programming languages rely on it for elaborating (typechecking) source programs, in much the same way as typechecking ML-style languages with Algorithm W [Milner 1978] makes use of first-order unification [Robinson 1965]. Languages with a kernel type theory, such as Coq and Epigram [McBride and McKinna 2004], do not need unification in the kernel, but they depend on it to elaborate human-readable syntax. Agda uses higher-order unification for pattern matching and implicit argument synthesis [Norell 2007].

Programmers in a dependently typed language need to grasp the capabilities (and in particular, the incapabilities) of unification if they are to become productive users of the language. Knowing what to omit, because the machine can reconstruct it for you, is a crucial aspect of writing comprehensible programs. We thus have a twofold motivation: not only to detail an algorithm for higher-order unification, for the benefit of language implementors, but also to explain it to programmers who wish to understand the tools they use. This paper fills a gap between descriptions of typechecking a kernel theory [Löh et al. 2010] and elaborating a high-level dependently typed language [Brady 2012].

Higher-order unification extends first-order unification in that

- terms have a binding structure, so unifiers must respect variable scope: e.g. $\lambda x.\alpha \equiv \lambda x.x$ may not be solved by $[x/\alpha]$ if the metavariable α cannot depend on the bound variable x; and
- terms have a nontrivial equational theory, given by the β- and η-rules: for example, $\lambda f.f \equiv \lambda f.\lambda x.\alpha\, f\, x$ can be solved by $[\lambda g.g/\alpha]$ as $\lambda f.\lambda x.(\lambda g.g)\, f\, x =_\beta \lambda f.\lambda x.f\, x =_\eta \lambda f.f$.

Given these complications it is perhaps unsurprising that full higher-order unification is undecidable [Huet 1973]. Most general unifiers do not necessarily exist and terms may have infinite sets of unifiers, though they can be generated by a semidecision procedure [Huet 1975]. Miller [1992] observed that a useful subproblem, unification in the *pattern fragment*, is decidable and has unique most general unifiers if they exist at all. Here metavariables must be applied to spines of distinct bound variables, so the two previous examples are included but $\lambda x.\alpha\, x\, x \equiv \lambda x.x$ is not; observe that it has two incompatible solutions $[\lambda x.\lambda y.x/\alpha]$ and $[\lambda y.\lambda x.x/\alpha]$.

Languages with simple pairs or Σ-types (pairs in which the type of the second component may depend on the value of the first component) motivate extending the pattern fragment to projections. For example, consider $\alpha\,\text{HD}\,x \equiv x$ where postfix HD is first projection. This does not fall in the original pattern fragment but has most general solution $[(\lambda x.x\ ,\ \beta)/\alpha]$ where β is a fresh variable.

For many applications, the static pattern fragment is overly restrictive: we often have multiple constraints, some of which fall into it and some of which do not, but solving those which do may make others solvable. We therefore need *dynamic* pattern unification, which postpones constraints that lie outside the pattern fragment in case they later become solvable.

Dynamic treatment of constraints is necessary even in first-order problems, because there is no fixed positional order of constraint solving that will work in all cases. For example, consider the problem $(\alpha + \beta\ ,\ \alpha) \equiv (3\ ,\ 0)$ where α and β are natural number metavariables. If our algorithm always unifies the components of pairs from left to right, it gets stuck on the constraint $\alpha + \beta \equiv 3$. On the other hand, after solving $\alpha \equiv 0$, the first constraint computes to the much easier $\beta \equiv 3$. (Clearly, always unifying from right to left is no better, because it fails if we swap the components of the pairs). The Coq proof assistant, used as a dependently typed programming language, suffers from exactly this problem.

1.1 Related work

Since Huet's seminal work on higher-order unification for simply typed λ-calculus [Huet 1975], many people have sought to extend it to dependently typed calculi, in particular for the Edinburgh Logical Framework [Harper et al. 1993], also known as λ^{Π}-calculus. Elliott [1990] and Pym [1992] both demonstrated unification algorithms based on Huet's, using the fact that dependencies are erasable in the LF to give notions of 'type similarity' (in Pym's terminology) that relate the types of terms being unified. Brown [1996] studied the metatheory of a variant of λ^{Π}-calculus with type similarity, and used this to re-present unification as a system of reduction rules.

In contrast to Huet-style semidecision procedures, which generate a sequence of unifiers, much research has been done on Miller's pattern unification [Miller 1992], which finds most general unifiers when they exist. Duggan [1998] generalised the pattern condition to support System F_{ω} with simple product types. Reed [2009] described how to apply dynamic pattern unification to LF. He introduced 'typing modulo' (discussed in subsection 1.2) as a neat simplification of type similarity and similar invariants used to handle the complications of type dependency. Abel and Pientka [2011] extended Reed's algorithm to support $\lambda^{\Pi\Sigma}$-calculus (LF with Σ-types) and implemented it for the Beluga language.

Separately, languages based on Martin-Löf Type Theory, such as Agda [Norell 2007], or the Calculus of (Inductive) Constructions, such as Coq, have developed higher-order unification algorithms. Here, unlike in LF, we have *full-spectrum dependency*: metavariables may stand for types (rather than merely appearing in them), and dependencies are not erasable as types may be recursively defined and computed from terms (by large elimination). Thus the work on unification for LF is not immediately applicable. Pfenning [1991] extended pattern unification to the Calculus of Constructions, characterising exactly those terms that fall in the pattern fragment statically; hence he is able to consider only well typed terms of identical type, because the types can always be unified first.

The present paper builds on the work of Reed [2009] and Abel and Pientka [2011] to describe unification for a full-spectrum dependent type theory. Moreover, we seek to lessen the gap between theory and practice by exhibiting a Haskell implementation of our algorithm, in the spirit of Nipkow's Standard ML implementation of pattern unification for simply typed λ-calculus [Nipkow 1993].

1.2 Heterogeneous equality

We write $\Pi\, S\ T$ as a 'constructor form' presentation of the dependent function space, often written as $(x : S) \to T\, x$. Given the problem $\Pi\, A\ B \equiv \Pi\, S\ T$, a reasonable step to take is to simplify it to $A \equiv S, B \equiv T$. However, at this stage $B : A \to \mathbf{Set}$ and $T : S \to \mathbf{Set}$ have apparently different types, as the equation between A and S may not be solved immediately. This shows the need for a heterogeneous notion of equality. In general, we will need to formulate and solve equations between vectors of terms in a telescope, so heterogeneity is inevitable. It is also plausible, however, in the sense that unifying the first $n-1$ terms in the vectors will make the types of the n^{th} terms equal.

Reed [2009] elegantly dealt with heterogeneity using a weaker invariant on homogeneous equations, *typing modulo*, which states that the two sides are well typed up to the equational theory of the constraints yet to be solved. However, this means that if there are blocked constraints left when the algorithm has terminated, then some solved metavariables may be ill typed (up to definitional equality). This is problematic for elaboration of a full-spectrum dependently typed source language. Here typechecking is interleaved with unification, so if unification creates ill typed terms we will have problems. The algorithm we present avoids this difficulty by ensuring that all outputs are well typed, provided it is given well typed input.

1.3 Intensional vs. extensional equality

Recall that *definitional* equality is the relation of $\beta\eta$-convertibility, written $s =_{\beta\eta} t$ (or simply $s = t$). For a strongly normalising type theory, it is easy to test, by checking that s and t have the same normal form (up to α-equivalence). It is *intensional* in the sense that extensionally equal terms need not be definitionally equal: for example, $s = \lambda x.\mathbf{true}$ and $t = \lambda x.\mathbf{if}\ x\ \mathbf{then}\ x\ \mathbf{else}\ \mathbf{true}$ are equal on all boolean inputs, but $s \neq_{\beta\eta} t$. Extensional equality is undecidable in general (is the function that, given a description of a Turing machine M and a natural number n, returns whether M halts within n steps, equal to the constantly $\mathbf{false}$ function?).

The algorithm we will describe finds solutions up to the intensional definitional equality, not extensional equality. Finding solutions up to extensional equality involves proof search and most general solutions are not (intensionally) unique. For example, if $\alpha : \mathbf{Bool} \to \mathbf{Bool}$ is a metavariable and $x : \mathbf{Bool}$ is a variable, the problem $\alpha\, x \equiv \mathbf{true}$ has unique solution $\lambda x.\mathbf{true}$ up to definitional equality, but other solutions up to extensional equality include $\lambda x.\mathbf{if}\ x\ \mathbf{then}\ x\ \mathbf{else}\ \mathbf{true}$ and $\lambda x.x \vee \mathbf{not}\ x$.

Most type theories have some internal notion of *propositional* equality in which equations can be proved, such as the identity type in Martin-Löf Type Theory [Martin-Löf and Sambin 1984], which reflects the definitional equality as a type, or coercion types in System F_{C} [Sulzmann et al. 2007], where equality evidence is explicit but in a different syntactic category to terms. Given a type theory with a sufficiently expressive propositional equality, we could represent unification problems as types, and unification could deliver terms as evidence (in the form of equality proofs). However, in this paper we prefer to make fewer assumptions about the object type theory to emphasise that our work is more widely applicable.

1.4 Outline

In this paper, we present an algorithm for dynamic pattern unification for a language with full-spectrum dependent types including Σ-types. This algorithm is implemented in Haskell.[1] Our contributions include:

- the use of heterogeneous equality constraints, instead of typing modulo, to maintain a typing discipline suitable for an intensional full-spectrum dependently typed language;
- a notion of 'twin variables' used to simplify problems such as $(\lambda x.s : \Pi\, A\ B) \equiv (\lambda x.t : \Pi\, S\ T)$ heterogeneously when A and S are intensionally distinct types;
- an account of the context structure suitable for managing dependency and tracking blocked unification problems; and
- the demonstration of a minimal-commitment strategy that makes it easy to deliver most general unifiers.

In Section 2, we describe the basic concepts we will use to implement pattern unification. We sketch the algorithm in Section 3, along with a high-level specification as rewrite rules on the metacontext. The actual implementation is in Section 4, and some concluding remarks form Section 5.

2. Setting the scene

In this section, we introduce the representations of terms and contexts, give the rules for the type theory in which we will be solving unification problems, discuss the use of 'twins' and recall some basic definitions about occurrences. These concepts will be used in section 3, where we specify the unification algorithm.

[1] Source code for everything presented here is available from `http://personal.cis.strath.ac.uk/adam.gundry/pattern-unify/`

$$
\begin{array}{rcl}
s,\ t,\ S,\ T & ::= & h\ \overline{e_i}^{\,i} \mid \lambda x.t \mid \mathbf{Set} \mid \Pi\ S\ T \mid \Sigma\ S\ T \mid (s,t) \\
h & ::= & x \mid \acute{x} \mid \grave{x} \mid \alpha \\
d,\ e & ::= & t \mid \textsc{hd} \mid \textsc{tl} \\
\Delta,\ \Theta & ::= & \cdot \mid \Delta, \alpha : T \mid \Delta, \alpha := t : T \mid \Delta, ?\,P \\
\Gamma,\ \Phi,\ \Psi & ::= & \cdot \mid \Gamma, x : T \mid \Gamma, \hat{x} : S \ddagger T \\
P & ::= & (s : S) \equiv (t : T) \mid \forall \Gamma\,.\,P
\end{array}
$$

Figure 1. Syntax

2.1 Term representation

The syntax of terms is given in Figure 1. Since this is a full-spectrum dependently typed language, types and terms live in a single syntactic category. Neutral terms are represented in spine form [Cervesato and Pfenning 2003], allowing easy access to the head h, which may be a variable x, y, z or a metavariable α, β, γ. The spine of eliminators $\overline{e_i}^{\,i}$ includes both applications to terms t and projections from Σ-types, written $\textsc{hd}$ for first projection and $\textsc{tl}$ for second projection. (The mysterious accents on variables will be discussed in subsection 2.4.)

A *telescope* $\Phi = \overline{x_i : T_i}^{\,i}$ is a vector of name bindings with corresponding types, where the type T_i may depend on the variables $x_0, \ldots, x_{i-1}$. We write $\Pi x : S.\ T$ for $\Pi\ S\ (\lambda x.T)$ and generalise this to binding a telescope $\Pi\Phi.\ T$. Similarly $h\ \Phi$ is the application of the head h to the variables bound in Φ. The non-dependent Π and Σ are $S \twoheadrightarrow T$ and $S \times T$ respectively.

In this representation, terms are always β-normal; this is possible thanks to hereditary substitution [Watkins et al. 2003], which evaluates redexes as soon as they are created. We will thus feel free to write the usual application $s\ t$.

Terms are represented using the data type Tm in Figure 2.[2] The Binders Unbound library of Weirich et al. [2011] defines the Bind type constructor and gives us a cheap locally nameless representation with operations including α-equivalence and substitution for first-order datatypes containing terms.

2.2 Contexts and unification problems

A *(meta)context* Δ is a list of entries, either metavariables α (carrying a type and possibly a definition) or unification problems P. Unification problems have a telescope of parameters Γ, but metavariables do not need one as they can simply have a Π-type instead. Scope is managed according to the invariant that each entry depends only on those that precede it, and in terms, metavariables are explicitly applied to all the variables they may depend upon.

Unification problems are heterogeneous equations under some universally quantified variables, written $\forall \Gamma\,.\,(s : S) \equiv (t : T)$. (We will sometimes omit the parameters and types, writing $s \equiv t$.) We have already remarked on the need for the terms being unified to have different types (subsection 1.2). Problems include a novel form of universal quantification in order to deal with heterogeneously equal hypotheses (see subsection 2.4).

For example,

$$\alpha : \mathbf{Set} \twoheadrightarrow \mathbf{Set}, \beta : \mathbf{Set},$$
$$?\,\forall X : \mathbf{Set} \twoheadrightarrow \mathbf{Set}\,.\,(\alpha\ X : \mathbf{Set}) \equiv (X\ \beta : \mathbf{Set})$$

is a valid metacontext, which declares metavariables α and β and has a single unification problem with parameter X.

Figure 3 shows the data types representing the metacontext, unification problems and parameters. The algorithm will move about in the context, keeping track of its position in traditional zipper fashion [Huet 1997], using a pair of lists for entries to the left and right of the cursor. The list to the right may also contain suspended substitutions, which will be used in subsection 4.8 to propagate information about updated metavariables. The $\mathsf{ProblemState}$ type will also be defined in subsection 4.8.

Problems have identifiers Id to record the dependency of one problem upon the solution of others. The specification will not make use of them, simply replacing problems with equivalent ones.

[2] Actually, in practice we use a single constructor for all canonical forms, in order to factor out common patterns in the typechecker. However, the unification algorithm is written as if Tm was defined in this way, thanks to pattern synonyms [Aitken and Reppy 1992; McBride 2010].

```
data Tm = N Head [Elim]          -- neutral application
        | L (Bind Nom Tm)        -- λ-abstraction
        | Set                    -- type of types
        | Π Tm Tm                -- function space
        | Σ Tm Tm                -- dependent sum
        | Pair Tm Tm             -- inhabits Σ

data Head = V Nom Twin | M Nom
data Twin = Only | TwinL | TwinR
data Elim = A Tm | Hd | Tl

(%%) :: Tm → Elim → Tm    -- elimination
($$) :: Tm → Tm → Tm      -- application

type Subs = [(Nom, Tm)]
substs     :: Subs → Tm → Tm      -- substitution
compSubs :: Subs → Subs → Subs    -- composition

type Type = Tm
```

Figure 2. Term representation and operations

```
data Dec       = HOLE | DEFN Tm
data Entry     = MV Nom Type Dec
               | Prob Id Problem ProblemState
type MContext = ([Entry], [Either Subs Entry])

data Problem  = All Param (Bind Nom Problem)
              | Unify Equation
data Equation = (Tm : Type) ≡ (Tm : Type)

data Param = P Type | Type‡Type
```

Figure 3. Context and problem representation

2.3 Typing rules

The typing rules are given in Figure 4. In the style of contextual type theory [Nanevski et al. 2008], we separate the metacontext Δ (which contains metavariables and unification problems) from the parameters list Γ (which binds variables). Unlike contextual type theory, however, we do not represent metavariable contexts explicitly: they simply have Π-types instead. Note that metavariables can depend only on variables they are applied to.

In the rules, $x \# \Gamma$ means x is fresh for Γ and similarly $\alpha \# \Delta$ means α is fresh for Δ. Whenever we have $\Delta \mid \Gamma \vdash t : T$ we take it as given that $\Delta \mid \Gamma \vdash T : \mathbf{Set}$. We adopt the inconsistent $\mathbf{Set} : \mathbf{Set}$ for simplicity of presentation, but the algorithm we describe can easily be adapted to a suitable type theory with Σ-types such as Luo's Extended Calculus of Constructions [1994]. For brevity, the typing rules for netural terms are not given in spine form, as this will be enforced by the syntax in any case.

$\boxed{\Delta \vdash \mathbf{mctx}}$ Δ is a valid metacontext

$$\frac{}{\cdot \vdash \mathbf{mctx}} \qquad \frac{\Delta \,|\, \Gamma \vdash s : S \quad \Delta \,|\, \Gamma \vdash t : T}{\Delta, ?\,\forall \Gamma\,.\,(s : S) \equiv (t : T) \vdash \mathbf{mctx}}$$

$$\frac{\alpha \# \Delta \quad \Delta \,|\, \cdot \vdash T : \mathbf{Set}}{\Delta, \alpha : T \vdash \mathbf{mctx}} \qquad \frac{\alpha \# \Delta \quad \Delta \,|\, \cdot \vdash t : T}{\Delta, \alpha := t : T \vdash \mathbf{mctx}}$$

$\boxed{\Delta \,|\, \Gamma \vdash \mathbf{ctx}}$ Γ is a valid parameter list in metacontext Δ

$$\frac{\Delta \vdash \mathbf{mctx}}{\Delta \,|\, \cdot \vdash \mathbf{ctx}} \qquad \frac{x \# \Gamma \quad \Delta \,|\, \Gamma \vdash T : \mathbf{Set}}{\Delta \,|\, \Gamma, x : T \vdash \mathbf{ctx}}$$

$$\frac{x \# \Gamma \quad \Delta \,|\, \Gamma \vdash S : \mathbf{Set} \quad \Delta \,|\, \Gamma \vdash T : \mathbf{Set}}{\Delta \,|\, \Gamma, \hat{x} : S \ddagger T \vdash \mathbf{ctx}}$$

$\boxed{\Delta \,|\, \Gamma \vdash t : T}$ term t has type T under Δ and Γ

$$\frac{\Delta \,|\, \Gamma \vdash \mathbf{ctx}}{\Delta \,|\, \Gamma \vdash \mathbf{Set} : \mathbf{Set}} \qquad \frac{\Delta \,|\, \Gamma \vdash S : \mathbf{Set} \quad \Delta \,|\, \Gamma \vdash T : S \to \mathbf{Set}}{\Delta \,|\, \Gamma \vdash \Pi\, S\ T : \mathbf{Set}}$$

$$\frac{\Delta \,|\, \Gamma \vdash S : \mathbf{Set} \quad \Delta \,|\, \Gamma \vdash T : S \to \mathbf{Set}}{\Delta \,|\, \Gamma \vdash \Sigma\, S\ T : \mathbf{Set}}$$

$$\frac{\Delta \,|\, \Gamma, x : S \vdash t : T\ x}{\Delta \,|\, \Gamma \vdash \lambda x.t : \Pi\, S\ T} \qquad \frac{\Delta \,|\, \Gamma \vdash s : S \quad \Delta \,|\, \Gamma \vdash t : T\ s}{\Delta \,|\, \Gamma \vdash (s\ ,\ t) : \Sigma\, S\ T}$$

$$\frac{\Delta \ni \alpha : T \quad \Delta \,|\, \Gamma \vdash \mathbf{ctx}}{\Delta \,|\, \Gamma \vdash \alpha : T} \qquad \frac{\Gamma \ni x : T \quad \Delta \,|\, \Gamma \vdash \mathbf{ctx}}{\Delta \,|\, \Gamma \vdash x : T}$$

$$\frac{\Gamma \ni \hat{x} : S \ddagger T \quad \Delta \,|\, \Gamma \vdash \mathbf{ctx}}{\Delta \,|\, \Gamma \vdash \acute{x} : S} \qquad \frac{\Gamma \ni \hat{x} : S \ddagger T \quad \Delta \,|\, \Gamma \vdash \mathbf{ctx}}{\Delta \,|\, \Gamma \vdash \grave{x} : T}$$

$$\frac{\Delta \,|\, \Gamma \vdash f : \Pi\, S\ T \quad \Delta \,|\, \Gamma \vdash s : S}{\Delta \,|\, \Gamma \vdash f\ s : T\ s} \qquad \frac{\Delta \,|\, \Gamma \vdash t : \Sigma\, S\ T}{\Delta \,|\, \Gamma \vdash t_{\,\mathrm{HD}} : S}$$

$$\frac{\Delta \,|\, \Gamma \vdash t : \Sigma\, S\ T}{\Delta \,|\, \Gamma \vdash t_{\,\mathrm{TL}} : T\ (t_{\,\mathrm{HD}})} \qquad \frac{\Delta \,|\, \Gamma \vdash t : S \quad \Delta \,|\, \Gamma \vdash S =_{\beta\eta} T}{\Delta \,|\, \Gamma \vdash t : T}$$

Figure 4. Typing rules

2.4 Twins

In order to implement unification, we will incrementally simplify unification problems. In a heterogeneous setting, an immediate question is how to simplify the problem

$$(f : \Pi\, A\ B) \equiv (g : \Pi\, S\ T).$$

It would not be type-correct (absent typing modulo) to produce

$$\forall x : A\,.\,(f\ x : B\ x) \equiv (g\ x : T\ x).$$

However, we can simplify it to

$$\forall \hat{x} : A \ddagger S\,.\,(f\ \acute{x} : B\ \acute{x}) \equiv (g\ \grave{x} : T\ \grave{x})$$

and we say $\acute{x}$ and $\grave{x}$ are *twin variables*, meaning that they represent the same variable at two different types. If the types later become definitionally equal, we can replace the twins with a single variable. On the other hand, the fact that they are different might not prevent the problem from being solved (if one of f and g is a constant function, for example). A single name is bound, but occurrences of the variable mark which twin they refer to. Thus they can be distinguished when typechecking but substitution must replace them with a single term (which will be type correct only if the types have been unified). Of course, twins are bound as parameters of unification problems, not in terms, so β-reduction never substitutes for twins.

If we were representing unification problems as types, twins would be modelled as distinct variables plus a proof of their equality, and replacing them with a single variable would exploit the elimination principle for equality.

Term equality is tested in the algorithm when typechecking a candidate solution for a metavariable, but it simply treats twins as distinct, and blocks until the types are equal. When calculating the free variables of a term, the twin annotations are ignored.

2.5 Occurrences

In the algorithm we must often distinguish between *flexible* occurrences, when a subterm occurs as an argument to a metavariable, from *rigid* occurrences, when it occurs otherwise. For example, in the term $\alpha\ X \to Y\ Z$, α, Y and Z occur rigidly while X occurs flexibly. Miller [1992] describes rigid occurrences as *permanent*, as opposed to *possible*, because flexible occurrences might be removed by substituting for metavariables but rigid occurrences cannot.

A rigid occurrence is *strong* if it is not an argument to a variable (so no substitution for variables can remove the occurrence). In the previous example, Y occurs strong rigidly but Z does not. When performing the occur check before solving a metavariable (to ensure the metavariable does not occur in its own candidate solution), the problem is unsolvable only if the metavariable occurs strong rigidly. For example, $\alpha \equiv \alpha \to \alpha$ is unsolvable but $\forall y : \mathbf{Bool} \to \mathbf{Bool}\,.\,\beta\ y \equiv y\ (\beta(\lambda x.x))$ is solvable (by $\lambda y.y\ \mathbf{true}$, amongst other things).

We write $\mathsf{fmv}(t)$ for the set of free metavariables of t, $\mathsf{fv}(t)$ for the free variables and $\mathsf{fv}^{\mathsf{rig}}(t)$ for the free variables that occur rigidly. The variables bound by a parameter list Γ are written $\mathsf{vars}(\Gamma)$.

3. Specification of the algorithm

Now that we can represent unification problems in context, we address the question of how to solve a unification problem. The idea is always to make small, local changes to the context, each of which is type-correct, makes the problem easier and preserves solutions (so that any solution we find is most general). Crucially, we make no unforced intensional choices: for example, given the equation $\alpha\,\mathbf{true} \equiv \mathbf{false}$, we may not define α by case analysis on its argument, in case a later equation demands $\alpha \equiv \lambda x.\mathbf{false}$ (which is intensionally distinct).

Those who prefer code over mathematical notation may wish to read section 4 in parallel with this section, where the algorithm described here is implemented.

The algorithm is represented in Figures 5–9 as a system of rewrite rules for transforming the metacontext. These rules are not deterministic as they permit arbitrary dependency-respecting permutation and working on problems in any order, but it is easy to make them deterministic by taking each active problem in turn, simplifying it by applying rules wherever possible and marking the problem as blocked when no more rules apply; blocked problems become active again when changed by substitution.

The rules should all be read as applying under an arbitrary context suffix, i.e. $\Delta, \Delta' \mapsto \Theta, \theta\Delta'$ if $\Delta \mapsto \Theta$ and θ is a substitution taking metavariables in Δ to their definitions in Θ (if any). The symbol $\bot$ is used to represent failure (an unsolvable constraint) and $\top$ is the empty conjunction (the trivial constraint). Any variables that appear on the right but not on the left are implicitly assumed to be freshly generated, so they do not conflict with any existing names. This can be achieved in the implementation by threading a name supply to generate fresh names as required.

The overall constraint solving steps (Figure 5) have rules for simplifying metavariables and problems, along with the main unification steps. We begin by describing unification, and will return to the simplification steps in subsection 3.5.

3.1 Unification

The heart of the algorithm is the procedure for solving individual unification problems. These split into four cases:

- problems that can be decomposed locally by η-expanding elements of Π or Σ types (Figure 6);
- rigid-rigid equations between two non-metavariables, which can also be decomposed locally (Figure 6);
- flex-rigid equations between a metavariable and a rigid term (Figure 7); and
- flex-flex equations between two metavariables (Figure 8).

η-expansion Given an equation between two functions, we have seen in subsection 2.4 that we can η-expand both sides by introducing twin variables, even if the domains are different. This is necessary to avoid losing solutions: for example, $\alpha\, y \equiv \lambda x.\alpha\, x\, x$ should not be rejected due to the occurrence of α on the right-hand side! Similarly, we can η-expand pairs by replacing

$$(u : \Sigma\, A\, B) \equiv (v : \Sigma\, S\, T) \quad \text{with}$$
$$(u\,\text{HD} : A) \equiv (v\,\text{HD} : S) \;\wedge\; (u\text{TL} : B(u\,\text{HD})) \equiv (v\text{TL} : T\,(v\,\text{HD})).$$

Rigid-rigid decomposition The simplest steps are those which decompose a rigid-rigid equation, leaving the context alone (Figure 6). Here neither side is an applied metavariable, so either the same symbol appears on both sides, or the equation is unsolvable. For example, we have seen that given $\Pi\, A\, B \equiv \Pi\, S\, T$ we can decompose it to $A \equiv S \;\wedge\; (B : A \twoheadrightarrow \mathbf{Set}) \equiv (T : S \twoheadrightarrow \mathbf{Set})$. Similarly, given $x\, \overline{d_i}^{\,i} \equiv x\, \overline{e_i}^{\,i}$ we can generate equations between the arguments d_i and e_i, insisting that projections match and unfolding the type of x to determine the types of the arguments. If we have twin variables at the head, say $\acute{x}\, \overline{d_i}^{\,i} \equiv \grave{x}\, \overline{e_i}^{\,i}$, we can proceed in the same way, with different types on each side. On the other hand, a mismatched equation like $\Pi\, A\, B \equiv \Sigma\, S\, T$ or $x \equiv y$ for distinct x and y can never be solved.

Flexible problems Once we have finished with rigid-rigid decomposition, we are left with a collection of flex-flex and flex-rigid problems (between two applied metavariables, or an applied metavariable and a rigid term, respectively). We first check whether there is any pruning to be done (see subsection 3.4), then try to solve the equation by transforming the context and instantiating metavariables (Figures 7 and 8). Our approach is always to look at the metavariable immediately left of the equation and ask "how do I instantiate this metavariable or move this problem past it?" For example, from the context $\Delta, \beta : S, ?\, P$ where β does not occur in P, we can simply move to $\Delta, ?\, P, \beta : S$ and continue.

With flex-rigid problems, we might encounter a metavariable that cannot be instantiated to solve the problem, but cannot be moved past either (as it occurs in the rigid term). For example, consider

$$\alpha : \mathbf{Set}, \beta : \mathbf{Set}, ?\,(\alpha : \mathbf{Set}) \equiv (\beta \twoheadrightarrow \beta : \mathbf{Set}).$$

Here we have no choice but to pick up β and move it left through the context until we find α. We represent this by

$$\alpha : \mathbf{Set}, [\beta : \mathbf{Set}]?\,(\alpha : \mathbf{Set}) \equiv (\beta \twoheadrightarrow \beta : \mathbf{Set}).$$

where the context entries in brackets are those that the problem depends on, and the entry to the left of the brackets is the next one to be examined by the algorithm. In this case, the problem is now trivially solvable, giving

$$\beta : \mathbf{Set}, \alpha := \beta \twoheadrightarrow \beta : \mathbf{Set}.$$

Blocking When moving an equation left in the context, we might find that we are unable to make further progress without making an intensional commitment or otherwise losing solutions, so no rule applies and we have to block instead. This means leaving the equation in the context, where hopefully other constraints will cause it to become easier to solve. For example,

$$\Delta, \alpha : T \twoheadrightarrow T \twoheadrightarrow T, ?\,\forall x : T\,.\,(\alpha\, x\, x : T) \equiv (x : T)$$

must block as the arguments of α are not linear, but if a later constraint demonstrates that α does not depend on its first argument, we get

$$\Delta, \beta : T \twoheadrightarrow T, \alpha := \lambda\_.\beta : T \twoheadrightarrow T \twoheadrightarrow T,$$
$$?\,\forall x : T\,.\,(\beta\, x : T) \equiv (x : T)$$

and we can resume solving the problem, which will instantiate β (after moving past α's definition).

3.2 Solving flex-rigid problems by inversion

In general, once we have moved a flex-rigid problem to its head metavariable, and rearranged its dependencies if necessary, we are in the situation

$$\Delta, \alpha : A, ?\,\forall \Gamma\,.\,(\alpha\, \overline{e_i}^{\,i} : S) \equiv (t : T)$$

where we hope to find a value for α that makes the equation hold. Miller observed that, if the arguments $\overline{e_i}^{\,i}$ are a list of distinct variables $\overline{x_i}^{\,i}$ containing all the free variables of t, and α does not occur in t, then this equation has the unique solution $\alpha \equiv \lambda\, \overline{x_i}^{\,i}\,.t$. For example, given the context

$$\Delta, \alpha : T \twoheadrightarrow T, ?\,\forall x : T\,.\,(\alpha\, x : T) \equiv (x : T)$$

we define α to give

$$\Delta, \alpha := \lambda x.x : T \twoheadrightarrow T, ?\,\forall x : T\,.\,(x : T) \equiv (x : T)$$

where the equation is now reflexive.

Occurs check First, we check whether α occurs strong rigidly in t (i.e. not as an argument to a variable or metavariable). If so, then the problem is unsolvable (for example, $\alpha \equiv \alpha \twoheadrightarrow \alpha$ has no solution).

Pattern condition Second, we check that $\overline{e_i}^{\,i}$ is (η-contractible to) a list of variables $\overline{x_i}^{\,i}$, and that it is linear on the variables that occur in t. If not, we have to block. For example, $\alpha\, x\, x \equiv x$ must block as we do not know whether α projects its first or second argument, but $\alpha\, x\, y\, x \equiv y$ can be solved unambiguously by $\lambda\_\ y\ \_.y$. We do not attempt to eliminate projections here, choosing instead to simplify metavariables and parameters before starting unification.

Typechecking Finally, we have a candidate solution $\lambda\, \overline{x_i}^{\,i}\,.t$ for $\alpha : A$, but we must check that it is scope-correct (in case α or any variables that are not in $\overline{x_i}^{\,i}$ occur) and type-correct (as heterogeneity means it might not be). Since we have already excluded the cases that are definitely unsolvable, the only thing we can do if not is block until the candidate solution becomes correct.

3.3 Solving flex-flex problems by inversion or intersection

The normal flex-flex case, with different metavariables on both sides, is solved in much the same way as the flex-rigid case. For example, given

$$\Delta, \alpha : T, ?\,\forall \Gamma\,.\,\alpha\, \overline{d_i}^{\,i} \equiv \beta\, \overline{e_i}^{\,i}$$

we perform the same checks as in subsection 3.2 to see if we can find a solution for α. If we cannot, we try moving α left in the context before the metavariable β, and try to solve for β.

In addition, we may have a problem of the form $\alpha\, \overline{d_i}^{\,i} \equiv \alpha\, \overline{e_i}^{\,i}$, with the same metavariable on both sides but potentially different spines of arguments. As usual, we must check that both spines are

Metavariable simplification

$$\begin{array}{rcl}
\Delta, \alpha : \Pi\Phi.\, \Sigma\, S\, T & \mapsto & \Delta, \alpha_0 : \Pi\Phi.\, S, \alpha_1 : \Pi\Phi.\, T\,(\alpha_0\,\Phi), \alpha := \lambda\Phi.(\alpha_0\,\Phi,\ \alpha_1\,\Phi) : \Pi\Phi.\, \Sigma\, S\, T \\
\Delta, \alpha : \Pi\Phi.\, \Pi x : (\Pi\Psi.\, \Sigma\, S\, T).\, U & \mapsto & \Delta, \beta : \Pi\Phi.\, \Pi y : (\Pi\Psi.\, S).\, \Pi z : (\Pi\Psi.\, T\,(y\,\Psi)).\, [(y\ ,\ z)/x]\, U, \\
 & & \quad \alpha := \lambda\Phi.\lambda x.\beta\,\Phi\,(\lambda\Psi.x\,\Psi_{\text{HD}})\,(\lambda\Psi.x\,\Psi_{\text{TL}}) : \Pi\Phi.\, \Pi x : (\Pi\Psi.\, \Sigma\, S\, T).\, U
\end{array}$$

Problem simplification

$$\begin{array}{rcll}
\Delta, ?\,\forall\Gamma\,.\,(t : T) \equiv (t' : T') & \mapsto & \Delta & \text{if } T =_{\beta\eta} T' \text{ and } t =_{\beta\eta} t' \\
\Delta, ?\,\forall\Gamma, x : T\,.\,P & \mapsto & \Delta, ?\,\forall\Gamma\,.\,P & \text{if } x \notin \mathsf{fv}(P) \\
\Delta, ?\,\forall\Gamma, \hat{x} : S \ddagger T\,.\,P & \mapsto & \Delta, ?\,\forall\Gamma\,.\,P & \text{if } x \notin \mathsf{fv}(P) \\
\Delta, ?\,\forall\Gamma, x : (\Pi\Phi.\, \Sigma\, S\, T)\,.\,P & \mapsto & \Delta, ?\,\forall\Gamma, y : (\Pi\Phi.\, S)\ ,\ z : (\Pi\Phi.\, T\,(y\,\Phi))\,.\,[(y\ ,\ z)/x]\,P & \\
\Delta, ?\,\forall\Gamma, \hat{x} : S \ddagger S'\,.\,P & \mapsto & \Delta, ?\,\forall\Gamma, x : S\,.\,[x/\hat{x}]\,P & \text{if } S =_{\beta\eta} S'
\end{array}$$

Unification

$$\begin{array}{rcll}
\Delta, ?\,P & \mapsto & \Delta, \overline{?\,P_i}^{\,i} & \text{if } P \mapsto_{\mathrm{d}} \bigwedge \overline{P_i}^{\,i} \text{ (see Figure 6)} \\
\Delta & \mapsto & \Delta' & \text{if } \Delta \mapsto_{\mathrm{p}} \Delta' \text{ (see subsection 3.4 and Figure 9)} \\
\Delta, ?\,\forall\Gamma\,.\,\alpha\,\overline{d_i}^{\,i} \equiv \beta\,\overline{e_j}^{\,j} & \mapsto & \Delta' & \text{if } \Delta, ?\,\forall\Gamma\,.\,\alpha\,\overline{d_i}^{\,i} \equiv \beta\,\overline{e_j}^{\,j} \mapsto_{\mathrm{ff}} \Delta' \text{ (see Figure 8)} \\
\Delta, ?\,\forall\Gamma\,.\,\alpha\,\overline{d_i}^{\,i} \equiv t & \mapsto & \Delta' & \text{if } t \text{ is not flexible and } \Delta, [\cdot]?\,\forall\Gamma\,.\,\alpha\,\overline{d_i}^{\,i} \equiv t \mapsto_{\mathrm{fr}} \Delta' \\
\Delta, ?\,\forall\Gamma\,.\,s \equiv t & \mapsto & \Delta' & \text{if } \Delta, ?\,\forall\Gamma\,.\,t \equiv s \mapsto \Delta'
\end{array}$$

Figure 5. Constraint solving steps

$$\begin{array}{rcl}
\forall\Gamma\,.\,(f : \Pi\, A\, B) \equiv (g : \Pi\, S\, T) & \mapsto_{\mathrm{d}} & \forall\Gamma, \hat{x} : A \ddagger S\,.\,(f\,\hat{x} : B\,\hat{x}) \equiv (g\,\grave{x} : T\,\grave{x}) \\
\forall\Gamma\,.\,(s : \Sigma\, A\, B) \equiv (t : \Sigma\, S\, T) & \mapsto_{\mathrm{d}} & \forall\Gamma\,.\,(s_{\text{HD}} : A) \equiv (t_{\text{HD}} : S)\ \wedge\ \forall\Gamma\,.\,(s_{\text{TL}} : B(s_{\text{HD}})) \equiv (t_{\text{TL}} : T\,(t_{\text{HD}})) \\
\forall\Gamma\,.\,(\mathbf{Set} : \mathbf{Set}) \equiv (\mathbf{Set} : \mathbf{Set}) & \mapsto_{\mathrm{d}} & \top \\
\forall\Gamma\,.\,(\Pi\, A\, B : \mathbf{Set}) \equiv (\Pi\, S\, T : \mathbf{Set}) & \mapsto_{\mathrm{d}} & \forall\Gamma\,.\,(A : \mathbf{Set}) \equiv (S : \mathbf{Set})\ \wedge\ \forall\Gamma\,.\,(B : A \twoheadrightarrow \mathbf{Set}) \equiv (T : S \twoheadrightarrow \mathbf{Set}) \\
\forall\Gamma\,.\,(\Sigma\, A\, B : \mathbf{Set}) \equiv (\Sigma\, S\, T : \mathbf{Set}) & \mapsto_{\mathrm{d}} & \forall\Gamma\,.\,(A : \mathbf{Set}) \equiv (S : \mathbf{Set})\ \wedge\ \forall\Gamma\,.\,(B : A \twoheadrightarrow \mathbf{Set}) \equiv (T : S \twoheadrightarrow \mathbf{Set}) \\
\forall\Gamma\,.\,x\,\overline{d_i}^{\,i} \equiv x\,\overline{e_i}^{\,i} & \mapsto_{\mathrm{d}} & \bigwedge \overline{\forall\Gamma\,.\,d_i \equiv e_i}^{\,i} \\
\forall\Gamma\,.\,\acute{x}\,\overline{d_i}^{\,i} \equiv \grave{x}\,\overline{e_i}^{\,i} & \mapsto_{\mathrm{d}} & \bigwedge \overline{\forall\Gamma\,.\,d_i \equiv e_i}^{\,i} \\
\forall\Gamma\,.\,s \equiv t & \mapsto_{\mathrm{d}} & \bot \text{ if } s \text{ and } t \text{ have different rigid head symbols}
\end{array}$$

Figure 6. η-expansion and rigid-rigid decomposition steps

$$\begin{array}{rcll}
\Delta, \alpha : T, [\Xi]?\,\forall\Gamma\,.\,\alpha\,\overline{x_i}^{\,i} \equiv t & \mapsto_{\mathrm{fr}} & \Delta, \Xi, \alpha := \lambda\,\overline{x_i}^{\,i}\,.t : T & \text{if } \overline{x_i}^{\,i} \text{ is linear on } \mathsf{fv}(t),\ \alpha \notin \mathsf{fmv}(\Xi, t) \text{ and } \Delta, \Xi \mid \cdot \vdash \lambda\,\overline{x_i}^{\,i}\,.t : T \\
\Delta, \alpha : T, [\Xi]?\,\forall\Gamma\,.\,\alpha\,\overline{x_i}^{\,i} \equiv t & \mapsto_{\mathrm{fr}} & \bot & \text{if } \alpha \text{ occurs strong rigidly in } t \\
\Delta, \beta : U, [\Xi]?\,P & \mapsto_{\mathrm{fr}} & \Delta, [\beta : U, \Xi]?\,P & \text{if } \beta \in \mathsf{fmv}(\Xi, P) \text{ and } \beta \text{ is not the head metavariable of } P \\
\Delta, \beta : U, [\Xi]?\,P & \mapsto_{\mathrm{fr}} & \Delta, [\Xi]?\,P, \beta : U & \text{if } \beta \notin \mathsf{fmv}(\Xi, P) \\
\Delta, ?\,P', [\Xi]?\,P & \mapsto_{\mathrm{fr}} & \Delta, [\Xi]?\,P, ?\,P' &
\end{array}$$

Figure 7. Flex-rigid unification steps

$$\begin{array}{rcll}
\Delta, \alpha : \Pi\Phi.\, T, ?\,\forall\Gamma\,.\,\alpha\,\overline{x_i}^{\,i} \equiv \alpha\,\overline{y_i}^{\,i} & \mapsto_{\mathrm{ff}} & \Delta, \beta : \Pi\Psi.\, T, & \text{if } \overline{x_i}^{\,i} \text{ agrees with } \overline{y_i}^{\,i} \text{ on } \Psi \subset \Phi \text{ and } \mathsf{fv}(T) \subset \mathsf{vars}(\Psi) \\
 & & \quad \alpha := \lambda\Phi.\beta\,\Psi : \Pi\Phi.\, T & \\
\Delta, \alpha : T, ?\,\forall\Gamma\,.\,\alpha\,\overline{x_i}^{\,i} \equiv \beta\,\overline{e_j}^{\,j} & \mapsto_{\mathrm{ff}} & \Delta, \alpha := \lambda\,\overline{x_i}^{\,i}\,.\beta\,\overline{e_j}^{\,j} : T & \text{if } \overline{x_i}^{\,i} \text{ linear on } \mathsf{fv}(\overline{e_j}^{\,j}),\ \alpha \notin \mathsf{fmv}(\beta\,\overline{e_j}^{\,j}) \\
 & & & \text{and } \Delta \mid \cdot \vdash \lambda\,\overline{x_i}^{\,i}\,.\beta\,\overline{e_j}^{\,j} : T \\
\Delta, \alpha : T, ?\,\forall\Gamma\,.\,\alpha\,\overline{d_i}^{\,i} \equiv \beta\,\overline{e_j}^{\,j} & \mapsto_{\mathrm{ff}} & \Delta' & \text{if } \Delta, [\alpha : T]?\,\forall\Gamma\,.\,\beta\,\overline{e_j}^{\,j} \equiv \alpha\,\overline{d_i}^{\,i} \mapsto_{\mathrm{fr}} \Delta' \\
\Delta, \beta : U, ?\,P & \mapsto_{\mathrm{ff}} & \Delta, ?\,P, \beta : U & \text{if } \beta \notin \mathsf{fmv}(P) \\
\Delta, ?\,P', ?\,P & \mapsto_{\mathrm{ff}} & \Delta, ?\,P, ?\,P' &
\end{array}$$

Figure 8. Flex-flex unification steps

$$\Delta, \beta : T, \Theta, ?\,\forall\Gamma\,.\,\alpha\,\overline{d_i}^{\,i} \equiv t \quad \mapsto_{\mathrm{p}} \quad \Delta, \gamma : U, \beta := u\,\gamma : T, [u\,\gamma/\beta]\Theta, ?\,[u\,\gamma/\beta]\,(\forall\Gamma\,.\,\alpha\,\overline{d_i}^{\,i} \equiv t)$$
$$\text{if } \beta\,\overline{e_j}^{\,j} \text{ occurs rigidly in } t \text{ and } \mathsf{pruneSpine}\ \cdot\ \cdot\,(\mathsf{vars}(\Gamma) \setminus \mathsf{fv}(\overline{d_i}^{\,i}))\ T\ \ \overline{e_j}^{\,j} \mapsto (U\ ,\ u)$$

$$\Delta, ?\,\forall\Gamma\,.\,\alpha\,\overline{d_i}^{\,i} \equiv t \quad \mapsto_{\mathrm{p}} \quad \bot \text{ if } \mathsf{fv}^{\mathsf{rig}}(t) \not\subset \mathsf{fv}(\overline{d_i}^{\,i})$$

$$\begin{array}{rcll}
\mathsf{pruneSpine}\,\Phi\,\Psi\ V\ T\ \cdot & \mapsto & (\Pi\Psi.\ T,\ \lambda y.\lambda\Phi.y\ \Psi) & \text{if } \mathsf{fv}(T) \subset \mathsf{vars}(\Psi) \text{ and } \Psi \neq \Phi \\
\mathsf{pruneSpine}\,\Phi\,\Psi\ V\,(\Pi\,S\ T)\,(x\,\overline{e_i}^{\,i}) & \mapsto & \mathsf{pruneSpine}\,(\Phi, x:S)\,(\Psi, x:S)\ V\,(T\,x)\ \ \overline{e_i}^{\,i} & \text{if } x \notin V \text{ and } \mathsf{fv}(S) \subset \mathsf{vars}(\Psi) \\
\mathsf{pruneSpine}\,\Phi\,\Psi\ V\,(\Pi\,S\ T)\ \ (s\,\overline{e_i}^{\,i}) & \mapsto & \mathsf{pruneSpine}\,(\Phi, y:S)\,\Psi\ V\,(T\,y)\ \ \overline{e_i}^{\,i} & \text{if } \mathsf{fv}^{\mathsf{rig}}(s) \cap V \neq \emptyset \text{ or } \mathsf{fv}^{\mathsf{rig}}(S) \not\subset \mathsf{vars}(\Psi)
\end{array}$$

Figure 9. Pruning

lists of variables. If so, we can restrict α to those arguments on which they agree. For example, a solution of

$$\Delta, \alpha : T \twoheadrightarrow T \twoheadrightarrow T, ?\,\forall x : T\,.\,\forall y : T\,.\,\alpha\,x\,x \equiv \alpha\,y\,x$$

is possible only if α does not depend on its first argument, so we can define

$$\Delta, \beta : T \twoheadrightarrow T, \alpha := \lambda\_.\beta : T \twoheadrightarrow T \twoheadrightarrow T,\ ?\,\forall x : T\,.\,(\beta\,x : T) \equiv (\beta\,x : T)$$

making the equation reflexive.

In LF, one can define intersection for meta-ground spines as well, but we are not free to do so. For example, $\alpha\,\mathbf{true}\,x \equiv \alpha\,\mathbf{true}\,y$ does not imply that α is independent of its second argument, as it might be defined by case analysis on its first argument. (It might not even have a second argument in the **false** case!)

3.4 Pruning

Given a problem of the form $\forall\Gamma\,.\,\alpha\,\overline{d_i}^{\,i} \equiv t$, we must check whether all the variables of t (which are bound by Γ) occur in $\overline{d_i}^{\,i}$, and hence are in scope for solutions of α. If any inaccessible variables occur rigidly, then the equation cannot be solved. On the other hand, if they occur only flexibly, we have to block, but we might be able to prune some metavariables to remove the occurrences of inaccessible variables. For example,

$$\alpha : \mathbf{Set}, ?\,\forall X : \mathbf{Set}\,.\,(\alpha : \mathbf{Set}) \equiv (X \twoheadrightarrow X : \mathbf{Set})$$

is unsolvable but

$$\beta : \mathbf{Set} \twoheadrightarrow \mathbf{Set}, \alpha : \mathbf{Set},\ ?\,\forall X : \mathbf{Set}\,.\,(\alpha : \mathbf{Set}) \equiv (\beta\,X \twoheadrightarrow \beta\,X : \mathbf{Set})$$

can be solved observing that β cannot depend on its argument, so we obtain

$$\gamma : \mathbf{Set}, \beta := \lambda\_.\gamma : \mathbf{Set} \twoheadrightarrow \mathbf{Set}, \alpha := \gamma \twoheadrightarrow \gamma : \mathbf{Set}.$$

The pruning steps are shown in Figure 9. Before moving the constraint left in the context with flex-flex or flex-rigid steps, we traverse t looking for free variables that are not in $\mathsf{fv}(\overline{d_i}^{\,i})$. When we encounter a metavariable, we call $\mathsf{pruneSpine}\,\Phi\,\Psi\ V\ T\ \ \overline{e_j}^{\,j}$ to prune the set of variables V from the spine $\overline{e_j}^{\,j}$, where T is the type of the metavariable. This accumulates telescopes of the original parameters (Φ) and those remaining after pruning (Ψ). If it finds a variable $x \notin V$ whose type depends only on remaining parameters, it leaves the argument alone, whereas if it finds a term containing variables in V or whose type depends on parameters that have been removed, it removes it. When it reaches the end of the spine, and provided the type depends only on remaining parameters, it returns the restricted type for the metavariable and a solution for the metavariable given a value of the new type.

Note that this is a partial operation: for example, pruning

$$\Delta, ?\,\forall x : T\,.\,\alpha \equiv \beta(\gamma\,x)$$

gets stuck because while α cannot depend on x, it is not obvious whether β or γ projects it away. (In particular, pruning gets stuck because the pruned variable x occurs flexibly in an argument to β.) If pruning gets stuck, we simply apply the other rules as normal, so the constraint will block due to the occurrence of x.

Pruning allows arguments to be retained only if they are variables, getting stuck otherwise, because the metavariable being pruned might be defined by case analysis on the argument (and hence removing other arguments might lose solutions). For example, knowing $\beta\,\mathbf{true}\,x$ cannot depend on x does not mean that β cannot depend on its second argument. Once again, Miller's pattern condition appears: a constraint captures the entire behaviour of a metavariable only if the metavariable is applied to a list of variables.

3.5 Metavariable and problem simplification

The simplification steps in Figure 5 are intended to be applied before and during unification, in order to eliminate projections from Σ-types, reflexive equations and redundant twins.

Metavariable simplification How can we solve the constraint $\alpha\,\text{HD} \equiv s$ where $\alpha : \Sigma\,S\ T$? One option is to extend the pattern fragment to cover projections, as Duggan [1998] does for System F_ω, but we take the simpler option of aggressively lowering metavariables to eliminate projections. In this case, replacing α with the pair $(\beta\ ,\ \gamma)$ of fresh metavariables $\beta : S, \gamma : T\,\beta$ simplifies the constraint to $\beta \equiv s$ which is easily solved.

Similarly, a metavariable $\alpha : \Pi x : \Sigma\,S\ T.\ U$ can be replaced with $\beta : \Pi x_0 : S.\,\Pi x_1 : T\,x_0.\,[(x_0\ ,\ x_1)/x]\ U$, which will transform the non-pattern constraint $\alpha(y\ ,\ z) \equiv t$ into the pattern constraint $\alpha\,y\,z \equiv t$.

These transformations maintain the same set of solutions thanks to the η-rule for Σ-types, otherwise known as surjective pairing, $(t\,\text{HD}\ ,\ t\,\text{TL}) =_\eta t$. Both of the rules are slightly complicated by the need to apply under any parameters Φ.

Problem simplification Unsurprisingly, reflexive equations can simply be removed from the context immediately, rather than invoking the unification algorithm. Moreover, parameters to problems that do not occur in the problem can be dropped.

Similarly to metavariable simplification, if a parameter has a Σ-type it can be replaced with two parameters in order to eliminate projections from equations. For example,

$$\Delta, ?\,\forall x : \Sigma\,S\ T\,.\,\alpha(x\,\text{TL}) \equiv t \quad \text{becomes}$$
$$\Delta, ?\,\forall x_0 : S\ ,\ x_1 : T\,x_0\,.\,\alpha\,x_1 \equiv [(x_0\ ,\ x_1)/x]\,t.$$

Finally, given a pair of twins whose types are definitionally equal, they can be replaced with a single variable, potentially allowing the further progress. Thus $\forall\hat{x} : S \ddagger S\,.\,P$ becomes $\forall x : S\,.\,[x/\hat{x}]\,P$. The substitution $[x/\hat{x}]$ is necessary to replace occurrences of $\hat{x}$ and $\grave{x}$ with the unannotated x.

4. Implementation

Having specified the unification algorithm in the previous section, we now implement it as a Haskell program. First we describe the domain-specific language in which we will write the program (subsection 4.1), then we show how to tackle the different kinds of unification problems step-by-step. Finally, we implement a simple way to manage constraint solving by working on the leftmost active constraint (subsection 4.8).

4.1 The Contextual monad: a DSL for problem solving

We work in a monad, Contextual, which stores the current context and parameters, generates fresh names when required for going under binders, and allows errors to be thrown when a unification problem is unsolvable. It supports the operations shown in Figure 10.

With these operations, we can write a bidirectional typechecker for the rules in Figure 4, as described by Löh et al. [2010]. This is based on a typed definitional equality test that η-expands its arguments and compares the normal forms up to α-equivalence.

There are also commands for recording inactive problems in the context, and for creating and solving metavariables. When making a hole of a given type, we supply a continuation that will be given a metavariable of that type and run with it in scope. We can also define an unknown metavariable by giving it a value. These both take a telescope of parameters under which to work.

4.2 Unification

As we have seen, unification splits into four main cases: η-expanding elements of Π or Σ types, rigid-rigid equations between non-metavariable terms (Figure 6), flex-rigid equations between a metavariable and a rigid term (Figure 7), and flex-flex equations between two metavariables (Figure 8). When the equation is between two functions, we use twins (see subsection 2.4) to decompose it heterogeneously. If it is in fact homogeneous, the twins will be simplified away later.

We will take slight liberties, here and elsewhere, with the Haskell syntax. In particular, we will use italicised capital letters (e.g. A) for Haskell variables representing types in the object language, while sans-serif letters (e.g. A) will continue to stand for data constructors.

```
unify :: Id → Equation → Contextual ()
unify n q@((f : Π A B) ≡ (g : Π S T)) = do
  x ← freshNom
  let (ẋ, ẋ) = (N (V x TwinL) [], N (V x TwinR) [])
  simplify n (Unify q) [Unify ((A : Set) ≡ (S : Set)),
    ∀x : A‡S. Unify ((f $$ ẋ : B $$ ẋ) ≡ (g $$ ẋ : T $$ ẋ))]
unify n q@((t : Σ A B) ≡ (w : Σ C D)) =
  simplify n (Unify q) [Unify ((a : A) ≡ (c : C))
                       , Unify ((b : B $$ a) ≡ (d : D $$ c))]
  where (a, b) = (t %% Hd, t %% Tl)
        (c, d) = (w %% Hd, w %% Tl)
unify n q@((N (M _) _ : _) ≡ (N (M _) _ : _)) =
  tryPrune n q $ tryPrune n (sym q) $ flexFlex n q
unify n q@((N (M _) _ : _) ≡ (_ : _)) =
  tryPrune n q $ flexRigid [] n q
unify n q@((_ : _) ≡ (N (M _) _ : _)) =
  tryPrune n (sym q) $ flexRigid [] n (sym q)
unify n q = rigidRigid q >>=
              simplify n (Unify q) ∘ map Unify
```

Here sym swaps the two sides of an equation:

```
sym :: Equation → Equation
sym ((s : S) ≡ (t : T)) = (t : T) ≡ (s : S)
```

Context manipulation:

```
popL       :: Contextual Entry
popR       :: Contextual (Maybe (Either Subs Entry))
pushL      :: Entry → Contextual ()
pushR      :: Either Subs Entry → Contextual ()
lookupMeta :: Nom → Contextual Type
```

Parameters:

```
ask         :: Contextual [(Nom, Param)]
localParams :: ([(Nom, Param)] → [(Nom, Param)]) →
                 Contextual a → Contextual a
lookupVar   :: Nom → Twin → Contextual Type
```

Fresh name generation and error handling:

```
freshNom   :: Contextual Nom
throwError :: String → Contextual a
catchError :: Contextual a → (String → Contextual a) →
                Contextual a
```

Type and equality checking:

```
equal     :: Type → Tm → Tm → Contextual Bool
typecheck :: Type → Tm → Contextual Bool
```

Stashing problems in the context:

```
active   :: Id → Equation → Contextual ()
block    :: Id → Equation → Contextual ()
failed   :: Id → Equation → String → Contextual ()
solved   :: Id → Equation → Contextual ()
simplify :: Id → Problem → [Problem] → Contextual ()
```

Creating and solving metavariables:

```
hole   :: [(Nom, Type)] → Type →
            (Tm → Contextual a) → Contextual a
define :: [(Nom, Type)] → Nom → Type → Tm →
            Contextual ()
```

Figure 10. Operations on Contextual monad

4.3 Rigid-rigid decomposition

A rigid-rigid equation (between two non-metavariable terms) can either be decomposed into simpler equations or it is impossible to solve. For example, $\Pi\ A\ B \equiv \Pi\ S\ T$ splits into $A \equiv S, B \equiv T$, but $\Pi\ A\ B \equiv \Sigma\ S\ T$ cannot be solved.

```
rigidRigid :: Equation → Contextual [Equation]
rigidRigid ((Set : Set) ≡ (Set : Set)) = return []
rigidRigid ((Π A B : Set) ≡ (Π S T : Set)) =
    return [(A : Set) ≡ (S : Set)
           , (B : A ⇀ Set) ≡ (T : S ⇀ Set)]
rigidRigid ((Σ A B : Set) ≡ (Σ S T : Set)) =
    return [(A : Set) ≡ (S : Set)
           , (B : A ⇀ Set) ≡ (T : S ⇀ Set)]
rigidRigid ((N (V x w) ds : S) ≡ (N (V y w′) es : T))
  | x ≐ y = do
    X ← lookupVar x w
    Y ← lookupVar y w′
    ((X : Set) ≡ (Y : Set):) <$>
      matchSpine X (N (V x w) []) ds
                   Y (N (V y w′) []) es
rigidRigid _ = throwError "Rigid-rigid mismatch"
```

When we have the same rigid variable (or twins) at the head on both sides, we proceed down the spine, demanding that projections are identical and unifying terms in applications. Note that matchSpine heterogeneously unfolds the types of the terms being applied to determine the types of the arguments. For example, if $x : \Pi a : A.\, B\ a \twoheadrightarrow C$ then the constraint $x\ s\ t \equiv x\ u\ v$ will decompose into $(s : A) \equiv (u : A) \wedge (t : B\ s) \equiv (v : B\ u)$.

```
matchSpine :: Type → Tm → [Elim] →
              Type → Tm → [Elim] →
              Contextual [Equation]
matchSpine (Π A B) u (A a : ds)
           (Π S T) v (A s : es) =
  ((a : A) ≡ (s : S):) <$>
    matchSpine (B $$ a) (u $$ a) ds (T $$ s) (v $$ s) es
matchSpine (Σ A B) u (Hd : ds) (Σ S T) v (Hd : es) =
  matchSpine A (u %% Hd) ds S (v %% Hd) es
matchSpine (Σ A B) u (Tl : ds) (Σ S T) v (Tl : es) =
  matchSpine (B $$ a) b ds (T $$ s) t es
  where (a, b) = (u %% Hd, u %% Tl)
        (s, t) = (v %% Hd, v %% Tl)
matchSpine _ _ [] _ _ [] = return []
matchSpine _ _ _ _ _ _ = throwError "spine mismatch"
```

4.4 Flex-rigid equations

A flex-rigid unification problem is one where one side is an applied metavariable and the other is a non-metavariable term. We move left in the context, accumulating a list of metavariables that the term depends on (the bracketed list of entries Ξ in the rules). Once we reach the target metavariable, we attempt to find a solution by inversion. This implements the steps in Figure 7, as described in subsection 3.2.

```
flexRigid :: [Entry] → Id → Equation →
             Contextual ()
flexRigid Ξ n q@((N (M α) _ : _) ≡ (_ : _)) = do
  Γ ← ask
  popL >>= \e → case e of
    MV β T HOLE
      | α ≐ β ∧ α ∈ fmv(Ξ) → do pushL e
                                mapM_ pushL Ξ
                                block n q
      | α ≐ β              → do mapM_ pushL Ξ
                                tryInvert n q T
                                  (block n q >>
                                   pushL e)
      | β ∈ fmv(Γ, Ξ, q)   → flexRigid (e : Ξ) n q
    _                      → do pushR (Right e)
                                flexRigid Ξ n q
```

Given a flex-rigid or flex-flex equation whose head metavariable has just been found in the context, tryInvert calls invert to seek a solution to the equation. If it finds one, it defines the metavariable and leaves the equation in the context (so the definition will be substituted out and the equation found to be reflexive by the constraint solver). If invert cannot find a solution, it runs the continuation.

```
tryInvert :: Id → Equation → Type → Contextual () →
             Contextual ()
tryInvert n q@((N (M α) es : _) ≡ (s : _)) T k =
  invert α T es s >>= \m → case m of
    Nothing → k
    Just v  → do active n q
                 define [] α T v
```

Given a metavariable α of type T, spine $\overline{e_i}^{\,i}$ and term t, invert attempts to find a value for α that solves the equation $\alpha\ \overline{e_i}^{\,i} \equiv t$. It may also throw an error if the problem is unsolvable due to an impossible (strong rigid) occurrence.

```
invert :: Nom → Type → [Elim] → Tm →
          Contextual (Maybe Tm)
invert α T es t = do
  let o = occurrence [α] t
  when (isStrongRigid o) $ throwError "occurrence"
  case toVars es of
    Just xs | o ≐ Nothing ∧ linearOn t xs → do
      b ← localParams (const []) (typecheck T (λxs. t))
      return $ if b then Just (λxs. t) else Nothing
    _ → return Nothing
```

Here toVars :: [Elim] → Maybe [Nom] tries to convert a spine to a list of variables, and linearOn :: Tm → [Nom] → Bool determines if a list of variables is linear on the free variables of a term. Note that we typecheck the solution $\lambda xs.\, t$ under no parameters, so typechecking will fail if an out-of-scope variable is used.

4.5 Flex-flex equations

A flex-flex unification problem is one where both sides are applied metavariables. As in the flex-rigid case, we proceed leftwards through the context, looking for one of the metavariables so we can try to solve it with the other. This implements the steps in Figure 8, as described in subsection 3.3.

```
flexFlex :: Id → Equation → Contextual ()
flexFlex n q@((N (M α) ds : _) ≡ (N (M β) es : _)) = do
  Γ ← ask
  popL >>= \e → case e of
    MV γ T HOLE
      | γ ≐ α ∧ γ ≐ β → do block n q
                           tryIntersect α T ds es
      | γ ≐ α         → tryInvert n q T
                           (flexRigid [e] n (sym q))
      | γ ≐ β         → tryInvert n (sym q) T
                           (flexRigid [e] n q)
      | γ ∈ fmv(Γ, q) → do pushL e
                           block n q
    _                 → do pushR (Right e)
                           flexFlex n q
```

When we have a flex-flex equation with the same metavariable on both sides, $\alpha\ \overline{x_i}^{\,i} \equiv \alpha\ \overline{y_i}^{\,i}$, where $\overline{x_i}^{\,i}$ and $\overline{y_i}^{\,i}$ are both lists of variables, we can solve the equation by restricting α to the arguments on which $\overline{x_i}^{\,i}$ and $\overline{y_i}^{\,i}$ agree (i.e. creating a new metavariable β and using it to solve α). The tryIntersect function tests if both spines are lists of variables, then calls intersect to generate a restricted type for the metavariable. If this succeeds, it creates a new metavariable and solves the old one. Otherwise, it leaves the old metavariable in the context.

```
tryIntersect :: Nom → Type → [Elim] → [Elim] →
                Contextual ()
tryIntersect α T ds es = case (toVars ds, toVars es) of
  (Just xs, Just ys) → intersect [] [] T xs ys >>= \m →
    case m of
      Just (U, f) → hole [] U $ \β →
                    define [] α T (f β)
      Nothing     → pushL (MV α T HOLE)
  _               → pushL (MV α T HOLE)
```

Given the type of α and the two spines, intersect produces a type for β and a term with which to solve α given β. It accumulates lists of the original and retained parameters (Φ and Ψ respectively).

```
intersect :: Fresh m ⇒ [(Nom, Type)] → [(Nom, Type)] →
                        Type → [Nom] → [Nom] →
                        m (Maybe (Type, Tm → Tm))
intersect Φ Ψ S [] []
  | fv(S) ⊂ vars(Ψ) = return $ Just
                        (ΠΨ. S, \β → λΦ. β $*$ Ψ)
  | otherwise       = return Nothing
intersect Φ Ψ (Π A B) (x : xs) (y : ys) = do
  z ← freshNom
  let Ψ' = Ψ ++ if x ≐ y then [(z, A)] else []
  intersect (Φ ++ [(z, A)]) Ψ' (B $$ var z) xs ys
```

Note that we have to generate fresh names in case the renamings are not linear. Also note that the resulting type is well-formed: if the domain of a Π depends on a previous variable that was removed, then the renamings will not agree, so it will be removed as well.

4.6 Pruning

When we have a flex-rigid or flex-flex equation, we might be able to make some progress by pruning the metavariables contained within it, as described in Figure 9 and subsection 3.4. The tryPrune function calls prune, and if it learns anything from pruning, leaves the current problem where it is and instantiates the pruned metavariable. If not, it runs the continuation.

```
tryPrune :: Id → Equation →
            Contextual () → Contextual ()
tryPrune n q@((N (M _) ds : _) ≡ (t : _)) k = do
  Γ ← ask
  u ← prune (vars(Γ) \ fv(ds)) t
  case u of
    d : _ → active n q ≫ instantiate d
    []    → k
```

Pruning a term requires traversing it looking for occurrences of forbidden variables. If any occur rigidly, the corresponding constraint is impossible. On the other hand, if we encounter a metavariable, we observe that it cannot depend on any arguments that contain rigid occurrences of forbidden variables. This can be implemented by replacing it with a fresh variable of restricted type. The prune function generates a list of triples (β, U, f) where β is a metavariable, U is a type for a new metavariable γ and $f\ \gamma$ is a solution for β. We maintain the invariant that U and $f\ \gamma$ depend only on metavariables defined prior to β in the context.

```
prune :: [Nom] → Tm →
         Contextual [(Nom, Type, Tm → Tm)]
prune xs Set       = return []
prune xs (Π S T)   = (++) <$> prune xs S <*> prune xs T
prune xs (Σ S T)   = (++) <$> prune xs S <*> prune xs T
prune xs (Pair s t) = (++) <$> prune xs s <*> prune xs t
prune xs (L b)     = prune xs =<< (snd <$> unbind b)
prune xs (N (V z _) es)
    | z ∈ xs       = throwError "pruning error"
    | otherwise    = concat <$> mapM pruneElim es
  where pruneElim (A a) = prune xs a
        pruneElim _     = return []
prune xs (N (M β) es) = do
  T ← lookupMeta β
  maybe [] (\(U, f) → [(β, U, f)]) <$>
    pruneSpine [] [] xs T es
```

Once a metavariable has been found, pruneSpine unfolds its type and inspects its arguments, generating lists of unpruned and pruned arguments (Φ and Ψ). If an argument contains a rigid occurrence of a forbidden variable, or its type rigidly depends on a previously removed argument, then it is removed. Ultimately, it generates a simplified type for the metavariable if the codomain type does not depend on a pruned argument.

```
pruneSpine :: [(Nom, Type)] → [(Nom, Type)] →
              [Nom] → Type → [Elim] →
              Contextual (Maybe (Type, Tm → Tm))
pruneSpine Φ Ψ xs (Π A B) (A a : es)
    | ¬ stuck = do
        z ← freshNom
        let Ψ' = Ψ ++ if pruned then [] else [(z, A)]
        pruneSpine (Φ ++ [(z, A)]) Ψ' xs (B $$ var z) es
    | otherwise = return Nothing
  where
    o      = occurrence xs a
    o'     = occurrence (vars(Φ) \ vars(Ψ)) A
    pruned = isRigid o ∨ isRigid o'
    stuck  = isFlexible o ∨ (isNothing o ∧ isFlexible o')
             ∨ (¬ pruned ∧ ¬ (isVar a))
pruneSpine Φ Ψ _ T [] | fv(T) ⊂ vars(Ψ) ∧ Φ ≱ Ψ =
    return $ Just (ΠΨ. T, \v → λΦ. v $*$ Ψ)
pruneSpine _ _ _ _ _ = return Nothing
```

After pruning, we can instantiate a pruned metavariable by moving left through the context until we find the relevant metavariable, then creating a new metavariable and solving the old one.

```
instantiate :: (Nom, Type, Tm → Tm) → Contextual ()
instantiate d@(α, T, f) = popL ≫= \e → case e of
  MV β U HOLE | α ≐ β → hole [] T $ \t →
                          define [] β U (f t)
  _                   → do pushR (Right e)
                           instantiate d
```

4.7 Metavariable and problem simplification

Given a problem, the solver simplifies it according to the rules in Figure 5, introduces parameters and calls unify from subsection 4.2. In particular, it removes Σ-types from parameters, potentially eliminating projections, and replaces twins whose types are definitionally equal with a normal parameter.

```
solver :: Id → Problem → Contextual ()
solver n (Unify q) = isReflexive q ≫= \b →
  if b then solved n q
       else unify n q `catchError` failed n q
solver n (All p b) = do
  (x, q) ← unbind b
  case p of
    _ | x ∉ fv(q) → simplify n (All p b) [q]
    P S  → splitSig [] x S ≫= \m → case m of
      Just (y, A, z, B, s, _) →
          solver n (∀y : A. ∀z : B. subst x s q)
      Nothing → localParams (++[(x, P S)]) $ solver n q
    S ‡ T → equal Set S T ≫= \c →
      if c then solver n (∀x : S. subst x (var x) q)
           else localParams (++[(x, S ‡ T)]) $ solver n q
```

Given the name and type of a metavariable, lower attempts to simplify it by removing Σ-types, according to the metavariable simplification rules in Figure 5. If it cannot be simplified, it appends it to the (left) context.

```
lower :: [(Nom, Type)] → Nom → Type → Contextual ()
lower Φ α (Σ S T) = hole Φ S $ \s →
                    hole Φ (T $$ s) $ \t →
                    define Φ α (Σ S T) (Pair s t)
lower Φ α (Π S T) = do
  x ← freshNom
  splitSig [] x S >>= maybe
    (lower (Φ ++ [(x, S)]) α (T $$ var x))
    (\(y, A, z, B, s, (u, v)) →
      hole Φ (Πy : A. Πz : B. T $$ s) $ \w →
        define Φ α (Π S T) (λx. w $$ u $$ v))
lower Φ α T = pushL (MV α (ΠΦ. T) HOLE)
```

Both solver and lower above need to split Σ-types (possibly underneath a bunch of parameters) into their components. For example, $y : \Pi x : X. \Sigma\, S\, T$ splits into $y_0 : \Pi x : X. S$ and $y_1 : \Pi x : X. T\,(y_0\, x)$. Given the name of a variable and its type, splitSig attempts to split it, returning fresh variables for the two components of the Σ-type, an inhabitant of the original type in terms of the new variables and inhabitants of the new types by projecting the original variable.

```
splitSig :: Fresh m ⇒ [(Nom, Type)] → Nom → Type →
            m (Maybe (Nom, Type, Nom, Type,
                      Tm, (Tm, Tm)))
splitSig Φ x (Σ S T) = do
  y ← freshNom
  z ← freshNom
  return $ Just (y, ΠΦ. S, z, ΠΦ. (T $$ var y),
                 λΦ. Pair (var y $*$ Φ) (var z $*$ Φ),
                 (λΦ. var x $*$ Φ %% Hd,
                  λΦ. var x $*$ Φ %% Tl))
splitSig Φ x (Π A B) = do
  a ← freshNom
  splitSig (Φ ++ [(a, A)]) x (B $$ var a)
splitSig _ _ _ = return Nothing
```

4.8 Solvitur ambulando

We organise constraint solving via an automaton that lazily propagates a substitution rightwards through the metacontext, making progress on active problems and maintaining the invariant that the entries to the left include no active problems. This is not the only possible strategy: indeed, it is crucial for guaranteeing most general solutions that solving the constraints in any order would produce the same result.

A problem may be in any of five possible states: Active and ready to be worked on; Blocked and unable to make progress in its current state; Pending the solution of some other problems in order to become solved itself; Solved as it has become reflexive; or Failed because it is unsolvable. The specification simply removes constraints that are pending or solved, and represents failed constraints as failure of the whole process, but in practice it is often useful to have a more fine-grained representation.

```
data ProblemState = Active | Blocked | Pending [Id]
                  | Solved | Failed String
```

In the interests of simplicity, Blocked problems do not store any information about when they may be resumed, and applying a substitution that modifies them in any way makes them Active. A useful optimisation would be to track the conditions under which they should become active, typically when particular metavariables are solved or types become definitionally equal.

The ambulando automaton carries a list of problems that have been solved, for updating the state of subsequent problems, and a substitution with definitions for metavariables.

```
ambulando :: [Id] → Subs → Contextual ()
ambulando ns θ = popR >>= \x → case x of
                 -- if right context is empty, stop
  Nothing        → return ()
                 -- compose suspended substitutions
  Just (Left θ') → ambulando ns (compSubs θ θ')
                 -- process entries
  Just (Right e) → case update ns θ e of
    Prob n p Active → do pushR (Left θ)
                         solver n p
                         ambulando ns []
    Prob n p Solved → do pushL (Prob n p Solved)
                         ambulando (n : ns) θ
    MV α T HOLE     → do lower [] α T
                         ambulando ns θ
    e'              → do pushL e'
                         ambulando ns θ
```

Given a list of solved problems, a substitution and an entry, update returns a modified entry with the substitution applied and the problem state changed if appropriate.

```
update :: [Id] → Subs → Entry → Entry
update _ θ (Prob n p Blocked) = Prob n p' k
  where p' = substs θ p
        k  = if p ≐ p' then Blocked else Active
update ns θ (Prob n p (Pending ys))
  | null rs     = Prob n p' Solved
  | otherwise = Prob n p' (Pending rs)
  where rs = ys \ ns
        p' = substs θ p
update _ _ e'@(Prob _ _ Solved)     = e'
update _ _ e'@(Prob _ _ (Failed _)) = e'
update _ θ e'                       = substs θ e'
```

5. Conclusion

We conclude with brief remarks on the correctness properties of the algorithm, followed by a discussion of its key features.

Termination Intuitively, the algorithm terminates because each key step makes the metacontext simpler: either decomposing a unification problem into smaller components, solving a metavariable or replacing a metavariable with a metavariable of smaller type. (Strictly speaking, this depends on strong normalisation, which does not hold for a $\mathbf{Set} : \mathbf{Set}$ theory, but would for a consistent theory with minimal changes to the algorithm.)

Soundness Since we work in small steps, it is easy to verify that each is sound. All permutations of the metacontext respect dependency. Whenever the algorithm instantiates a metavariable, it does so with a term of the appropriate type. Moreover, every unification problem is replaced with an equivalent conjunction of unification problems. Thus the resulting metacontext is valid.

Generality We have carefully maintained the invariant that the algorithm makes no unforced intensional choices: that is, metavariables are solved only if the solution is unique (up to definitional equality). This corresponds to finding most general unifiers.

Completeness As we observed in the introduction, full higher-order unification is undecidable, so our algorithm is incomplete in general. We conjecture that it is complete for the Miller pattern fragment (where all metavariables are applied to spines of distinct bound variables). It goes beyond the pattern fragment in handling Σ-types. We believe that it handles a sufficiently broad class of problems to be useful for elaboration of a dependently typed language.

Performance and complexity The algorithm presented here is optimised for clarity rather than performance, and we have not considered its algorithmic complexity. A real implementation would probably need to use a representation of terms with more control over depth of evaluation, and could record much more precisely the conditions under which a blocked problem should be resumed (rather than completely reassessing it whenever it is changed by a substitution). Similarly, rather than repeatedly checking to see if the types of metavariables can be simplified, it could eliminate projections as they arise in unification problems.

5.1 Discussion

We have presented an implementation of an algorithm for higher-order dynamic pattern unification in a full-spectrum dependent type theory. Our approach to problem solving, based on representing metavariables and problems in an ordered context, allows careful control over dependency and makes it easy to suspend work on one problem while we try to solve another. Solving problems in small steps makes it easy to see that the algorithm is sound and most general.

The particular strategy for tackling constraints (here leftmost active constraint first) is inessential: the order in which constraints are solved should not make a difference to the result, and implementations are free to make alternative choices, provided all active constraints are eventually dealt with. Of course, since vectors of equations arise from telescopes, it will usually make sense to solve the leftmost equations first so that later equations become homogeneous.

Crucially, the algorithm uses heterogeneous equality to make it easy to represent the telescopes of equations that arise from dependent arguments, potentially allowing progress on some equations even if the equation that makes their types equal is initially blocked. Despite this, and unlike typing modulo, every solution is well typed up to the definitional equality, making the algorithm useful when mixing typechecking with elaboration. The algorithm makes use of *twins*, a novel representation of universally quantified variables in unification problems which have two intensionally distinct types (until the equation between the types is solved). It demonstrates the practicality of this concept as an implementation technique; we leave further consideration of twins' metatheoretic properties as future work.

Acknowledgments

The first author was supported by the Microsoft Research PhD Scholarship Programme. Our thanks also go to the authors of the `Ott` and `lhs2TeX` tools used in the production of this paper.

References

A. Abel and B. Pientka. Higher-order dynamic pattern unification for dependent types and records. In *Typed Lambda Calculi and Applications*, TLCA '11, pages 10–26. Springer, 2011.

W. E. Aitken and J. H. Reppy. Abstract value constructors. Tech. Rep. 92-1290, Department of Computer Science, Cornell University, 1992.

E. Brady. Implementing general purpose dependently typed languages. Submitted, 2012. URL `http://www.cs.st-andrews.ac.uk/~eb/drafts/impldtp.pdf`.

J. J. Brown. *Presentations of Unification in a Logical Framework*. PhD thesis, University of Oxford, 1996.

I. Cervesato and F. Pfenning. A linear spine calculus. *Journal of Logic and Computation*, 13(5):639–688, 2003.

D. Duggan. Unification with extended patterns. *Theoretical Computer Science*, 206(12):1 – 50, 1998.

C. Elliott. *Extensions and Applications of Higher-Order Unification*. PhD thesis, School of Computer Science, Carnegie Mellon University, 1990.

R. Harper, F. Honsell, and G. Plotkin. A framework for defining logics. *Journal of the ACM*, 40(1):143–184, 1993.

G. Huet. The undecidability of unification in third order logic. *Information and Control*, 22(3):257–267, 1973.

G. Huet. A unification algorithm for typed lambda-calculus. *Theoretical Computer Science*, 1(1):27–57, 1975.

G. Huet. The Zipper. *Journal of Functional Programming*, 7(5):549–554, 1997.

A. Löh, C. McBride, and W. Swierstra. A tutorial implementation of a dependently typed lambda calculus. *Fundamenta Informaticæ*, 102(2): 177–207, 2010.

Z. Luo. *Computation and Reasoning: A Type Theory for Computer Science*. Clarendon Press, 1994.

P. Martin-Löf and G. Sambin. *Intuitionistic Type Theory*. Bibliopolis, 1984.

C. McBride. Strathclyde Haskell Enhancement, 2010. URL `http://personal.cis.strath.ac.uk/conor.mcbride/pub/she/`.

C. McBride and J. McKinna. The view from the left. *Journal of Functional Programming*, 14(1):69–111, 2004.

D. Miller. Unification under a mixed prefix. *Journal of Symbolic Computation*, 14(4):321–358, 1992.

R. Milner. A theory of type polymorphism in programming. *Journal of Computer and System Sciences*, 17(3):348–375, 1978.

A. Nanevski, F. Pfenning, and B. Pientka. Contextual modal type theory. *ACM Transactions on Computational Logic*, 9(3):23:1–23:49, 2008.

T. Nipkow. Functional unification of higher-order patterns. In *Logic in Computer Science*, LICS '93, pages 64–74. IEEE, 1993.

U. Norell. *Towards a practical programming language based on dependent type theory*. PhD thesis, Chalmers University of Technology, 2007.

F. Pfenning. Unification and anti-unification in the calculus of constructions. In *Logic in Computer Science*, LICS '91, pages 74–85. IEEE, 1991.

D. Pym. A unification algorithm for the $\lambda\pi$-calculus. *International Journal of Foundations of Computer Science*, 3(3):333–378, 1992.

J. Reed. Higher-order constraint simplification in dependent type theory. In *Logical Frameworks and Meta-Languages: Theory and Practice*, LFMTP '09, pages 49–56. ACM, 2009.

J. A. Robinson. A machine-oriented logic based on the resolution principle. *Journal of the ACM*, 12(1):23–41, 1965.

M. Sulzmann, M. M. T. Chakravarty, S. Peyton Jones, and K. Donnelly. System F with type equality coercions. In *Types in Language Design and Implementation*, TLDI '07, pages 53–66. ACM, 2007.

K. Watkins, I. Cervesato, F. Pfenning, and D. Walker. A concurrent logical framework I: Judgments and properties. Tech. Rep. CMU-CS-02-101, School of Computer Science, Carnegie Mellon University, 2003.

S. Weirich, B. A. Yorgey, and T. Sheard. Binders unbound. In *International Conference on Functional Programming*, ICFP '11, pages 333–345. ACM, 2011.