Generic pattern unification

We provide a generic second-order unification algorithm for Miller's pattern fragment, implemented in Agda. The syntax with metavariables is parameterised by a notion of signature generalising binding signatures, covering ordered λ-calculus, or (intrinsic) polymorphic syntax such as System F. The correctness of the algorithm is stated and proved on papers using a categorical perspective, based on the observation that the most general unifier is an equaliser in a multi-sorted Lawvere theory, thus generalising the case of first-order unification.

ACM Reference Format:
. 2024. Generic pattern unification. *Proc. ACM Program. Lang.* 1, POPL, Article 1 (January 2024), 9 pages.

1 INTRODUCTION

Unification consists in finding a *unifier* of two terms t, u, that is a (metavariable) substitution σ such that $t[\sigma] = u[\sigma]$. Unification algorithms try to compute a most general unifier σ, in the sense that given any other unifier δ, there exists a unique δ' such that $\delta = \sigma[\delta']$. First-order unification **?** is used in ML-style type inference systems and logic programming languages such as Prolog. More advanced type systems, where variable binding is crucially involved, requires second-order unification **?**, which is undecidable **?**. However, Miller **?** identified a decidable fragment: in so-called *pattern unification*, metavariables are allowed to take distinct variables as arguments. In this situation, we can write an algorithm that either fails in case there is no unifier, or computes the most general unifier.

Recent results in type inference, Dunfield-Krishnaswami **?**, or Jinxu et. al **?**, include very large proofs: the former comes with a 190 page appendix, and the latter comes with a Coq proof many thousands of lines long -- and both of these results are for tiny kernel calculi. If we ever hope to extend this kind of result to full programming languages like Haskell or OCaml, we must raise the abstraction level of these proofs, so that they are no longer linear (with a large constant) in the size of the calculus. A close examination of these proofs shows that a large part of the problem is that the type inference algorithms make use of unification, and the correctness proofs for type inference end up essentially re-establishing the entire theory of unification for each algorithm. The reason they do this is because algorithmic typing rules essentially give a first-order functional program with no abstractions over (for example) a signature for the unification algorithm to be defined over, or any axiomatic statement of the invariants the algorithmic typing rules had to maintain.

The present work is a first step towards a general solution to this problem. Our generic unification algorithm implemented in Agda is parameterised by a new notion of signature for syntax with metavariables, whose scope goes beyond the standard binding signatures. One important feature is that the notion of contexts is customisable, making it possible to cover simply-typed second-order syntax, ordered syntax, or (intrinsic) polymorphic syntax such as System F. We focused on Miller's pattern unification, as this is already a step beyond the above-cited works **??** that use plain first-order unification. Moreover, this is necessary for types with binders (e.g., fixed-point operators like $\mu a.A[a]$) as well as for rich type systems like dependent types.

Author's address:

2024. 2475-1421/2024/1-ART1 $15.00
https://doi.org/

Agda implementation

We use Agda as a programming language, not as a theorem prover. We leave for future work the task of mechanising the correctness proof of the algorithm, by investigating the formalisation of various concepts from category theory – a notorious challenge on its own – on which our proof relies on.

Since we focus on providing an effective implementation, the definitions of our data structures typically do not mention the properties. For example, our definition of categories in Agda does not include associativity of composition and neutrality of the identity:

```
record Category : Set where
  field
    Obj : Set
    _⇒_ : Obj → Obj → Set

    id : ∀ {A} → (A ⇒ A)
    _∘_ : ∀ {A B C} → (B ⇒ C) → (A ⇒ B) → (A ⇒ C)
```

Furthermore, we are not reluctant to using logically inconsistent features to make programming easier: the type hierachy is collapsed and the termination checker is disabled. We find that dependent types are still helpful in guiding the implementation. In comparison, we previously implemented an ocaml version where the code was much less constrained by the typing discipline, and thus more error-prone.

Related work

First-order unification has been explained from a lattice-theoretic point of view by Plotkin **?**, and later categorically analysed in **???**, Section 9.7 as coequalisers. However, there is little work on understanding pattern unification algebraically, with the notable exception of **?**, working with normalised terms of simply-typed λ-calculus. The present paper can be thought of as a generalisation of their work.

Although our notion of signature has a broader scope since we are not specifically focusing on syntax where variables can be substituted, our work is closer in spirit to the presheaf approach **?** to binding signatures than to the nominal approach **?** in that everything is explicitly scoped: terms come with their support, metavariables always appear with their scope of allowed variables.

Nominal unification **?** is an alternative to pattern unification where metavariables are not supplied with the list of allowed variables. Instead, substitution can capture variables. Nominal unification explicitly deals with α-equivalence as an external relation on the syntax, and as a consequence deals with freshness problems in addition to unification problems.

Cheney **?** shows that nominal unification and pattern unification problems are inter-translatable. As he notes, this result indirectly provides semantic foundations for pattern unification based on the nominal approach. In this respect, the present work provides a more direct semantic analysis of pattern unification, leading us to the generic algorithm we present, parameterised by a general notion of signature for the syntax.

Plan of the paper

In section §2, we present our generic pattern unification algorithm, parameterised by our generalised notion of binding signature. We introduce categorical semantics of pattern unification in Section §**??**. We show correctness of the two phases of the unification algorithm in Section §**??** and Section §**??**. Completeness is justified in Section §**??**. Finally, we present some examples of signatures in Section §**??**.

General notations

Given a list $\vec{x} = (x_1, \ldots, x_n)$ and a list of positions $\vec{p} = (p_1, \ldots, p_m)$ taken in $\{1, \ldots, n\}$, we denote $(x_{p_1}, \ldots, x_{p_m})$ by $x_{\vec{p}}$.

Given a category $\mathscr{B}$, we denote its opposite category by $\mathscr{B}^{op}$. If a and b are two objects of $\mathscr{B}$, we denote the set of morphisms between a and b by $\hom_{\mathscr{B}}(a, b)$. We denote the identity morphism at an object x by 1_x. We denote the coproduct of two objects A and B by $A + B$ and the coproduct of a family of objects $(A_i)_{i \in I}$ by $\coprod_{i \in I} A_i$, and similarly for morphisms. If $f : A \longrightarrow B$ and $g : A' \longrightarrow B$, we denote the induced morphism $A + A' \longrightarrow B$ by f, g. Coproduct injections $A_i \longrightarrow \coprod_{i \in I} A_i$ are typically denoted by in_i. Let T be a monad on a category $\mathscr{B}$. We denote its unit by η, and its Kleisli category by Kl_T: the objects are the same as those of $\mathscr{B}$, and a Kleisli morphism from A to B is a morphism $A \longrightarrow TB$ in $\mathscr{B}$. We denote the Kleisli composition of $f : A \longrightarrow TB$ and $g : B \longrightarrow TC$ by $f[g] : A \longrightarrow TC$.

2 PRESENTATION OF THE ALGORITHM

In this section, we start by describing a pattern unification algorithm for pure λ-calculus, summarised in Figure 1. Then we present our generic algorithm (Figure 2), and finally show that it indeed describes a terminating algorithm in Section §??. Soundness of the algorithm is justified in later sections.

2.1 An example: pure λ-calculus.

Consider the syntax of pure λ-calculus extended with metavariables satisfying the pattern restriction. We list the Agda code in Figure 3, together with more legible inductive rules generating the syntax. We write $\Gamma; n \vdash t$ to mean t is a wellformed λ-term in the context $\Gamma; n$, consisting of two parts:

(1) a metavariable context $\Gamma = (M_1 : m_1, \ldots, M_p : m_p)$, specifying metavariable symbols M_i together with their arities, i.e, their number of arguments m_i, and
(2) a variable context, which is a mere natural number indicating the highest possible free variable.

Free variables are indexed from 1 and we use the De Bruijn level convention: the variable bound in $\Gamma; n \vdash \lambda t$ is $n + 1$, not 0, as it would be using De Bruijn indices ?. In Agda, variables in the variable context n consist of elements of Fin n, the type of natural numbers between[1] 1 and n. We also use a nameless encoding of metavariable contexts: they are mere lists of metavariable arities, and metavariables are referred to by their index in the list (starting from 0). More concretely, let us focus on the last constructor building a metavariable application in the context $\Gamma; n$. The argument of type $m \in \Gamma$ is an index of any element m in the list Γ. This constructor also takes an argument of type $m \Rightarrow n$, which unfolds as Vec (Fin n) m: this is the type of lists of size m consisting of natural numbers between 1 and n. Note that contrary to our mathematical description, the Agda code does not explicitly enforce that the metavariable arguments are distinct. Anyway, our unification algorithm is only guaranteed to produce correct outputs if this constraint is satisfied in the inputs.

The Agda implementation of metavariable substitution is listed in Figure 5. A *metavariable substitution* $\sigma : \Gamma \longrightarrow \Delta$ assigns to each metavariable M of arity m in Γ a term $\Delta; m \vdash \sigma_M$. In Agda, the type of substitutions between Γ and Δ is defined as VecList (Tm Δ) Γ, where VecList.t X ℓ is (recursively) defined as the product type $X\ a_1 \times \cdots \times X\ a_n$ for any dependent type $X : A \longrightarrow$ Set and list $\ell = [a_1, \ldots, a_n]$ of elements of A.

[1]Fin n is actually defined in the standard library as an inductive type designed to be (canonically) isomorphic with $\{0, \ldots, n - 1\}$.

Judgments

$$\Gamma \vdash \vec{t} = \vec{u} \Rightarrow \sigma \dashv \Delta \iff \sigma : \Gamma \longrightarrow \Delta \text{ is the most general unifier of } \vec{t} \text{ and } \vec{u}$$

$$\Gamma \vdash \vec{u} \mathrel{:>} \overrightarrow{M(\vec{x})} \Rightarrow \vec{w}; \sigma \dashv \Delta \iff \sigma : \Gamma \longrightarrow \Delta \text{ extended with } M_i \mapsto w_i \text{ is the most general unifier of } \Gamma \vdash \vec{u}$$

$$m \vdash \vec{x} = \vec{y} \Rightarrow \vec{z} \dashv p \iff (z_1, \ldots, z_p) \text{ are the common positions of } (x_1, \ldots, x_m) \text{ and } (y_1, \ldots, y_m)$$

$$n \vdash \vec{x} \mathrel{:>} \vec{y} \Rightarrow \vec{l}; \vec{r} \dashv p \iff (l_1, \ldots, l_p) \text{ and } (r_1, \ldots, r_p) \text{ are the common value positions of } (x_1, \ldots, x$$

Unification Phase

- Structural rules

$$\frac{}{\Gamma \vdash () = () \Rightarrow 1_\Gamma \dashv \Gamma}\text{U}\Lambda\text{-Empty} \qquad \frac{}{\bot \vdash \vec{t} = \vec{u} \Rightarrow\ ! \dashv \bot}\text{U}\Lambda\text{-ExFalso}$$

$$\frac{\Gamma \vdash t_1 = u_1 \Rightarrow \sigma_1 \dashv \Delta_1 \qquad \Delta_1 \vdash t_2[\sigma_1] = u_2[\sigma_1] \Rightarrow \sigma_2 \dashv \Delta_2}{\Gamma \vdash t_1, t_2 = u_1, u_2 \Rightarrow \sigma_1[\sigma_2] \dashv \Delta_2}\text{U}\Lambda\text{-Split}$$

- Rigid-rigid (o, o' are applications, λ-abstractions, or variables)

$$\frac{\Gamma \vdash \vec{t} = \vec{u} \Rightarrow \sigma \dashv \Delta}{\Gamma \vdash o(\vec{t}) = o(\vec{u}) \Rightarrow \sigma \dashv \Delta}\text{U}\Lambda\text{-RigRig} \qquad \frac{o \neq o'}{\Gamma \vdash o(\vec{t}) = o'(\vec{u}) \Rightarrow\ ! \dashv \bot}\text{U}\Lambda\text{-Clash}$$

- Flex-*, no cycle

$$\frac{M \notin u \qquad \Gamma \vdash u \mathrel{:>} M(\vec{x}) \Rightarrow w; \sigma \dashv \Delta}{\Gamma, M : m \vdash M(\vec{x}) = u \Rightarrow \sigma, M \mapsto w \dashv \Delta}\text{U}\Lambda\text{-NoCycle} \quad + \text{ symmetric rule}$$

- Flex-Flex, same

$$\frac{m \vdash \vec{x} = \vec{y} \Rightarrow \vec{z} \dashv p}{\Gamma, M : m \vdash M(\vec{x}) = M(\vec{y}) \Rightarrow M \mapsto M'(\vec{z}) \dashv \Gamma, M' : p}\text{U}\Lambda\text{-Flex}$$

- Flex-Rigid, cyclic

$$\frac{M \in u \qquad u \neq M(\ldots)}{\Gamma, M : m \vdash M(\vec{x}) = u \Rightarrow\ ! \dashv \bot}\text{U}\Lambda\text{-Cyclic} \quad + \text{ symmetric rule}$$

Non-cyclic Phase

- Structural rules

$$\frac{}{\Gamma \vdash () \mathrel{:>} () \Rightarrow (); 1_\Gamma \dashv \Gamma} \qquad \frac{}{\bot \vdash \vec{t} \mathrel{:>} \overrightarrow{M(\vec{x})} \Rightarrow\ !;\ ! \dashv \bot}$$

$$\frac{\Gamma \vdash t_1 \mathrel{:>} M(\vec{x}_1) \Rightarrow u_1; \sigma_1 \dashv \Delta_1 \qquad \Delta_1 \vdash \vec{t_2}[\sigma_1] \mathrel{:>} \overrightarrow{M(\vec{x}_2)} \Rightarrow \vec{u_2}; \sigma_2 \dashv \Delta_2}{\Gamma \vdash t_1, \vec{t_2} \mathrel{:>} M_1(\vec{x}_1), \overrightarrow{M(\vec{x}_2)} \Rightarrow u_1[\sigma_2], \vec{u_2}; \sigma_1[\sigma_2] \dashv \Delta_2}\text{P}\Lambda\text{-Split}$$

- Rigid

$$\frac{\Gamma \vdash t \mathrel{:>} M'(\vec{x}, \overbrace{n+1}^{\text{bound variable}}) \Rightarrow w; \sigma \dashv \Delta}{\Gamma \vdash \lambda t \mathrel{:>} M(\vec{x}) \Rightarrow \lambda w; \sigma \dashv \Delta}\text{P}\Lambda\text{-Lam} \qquad \frac{\Gamma \vdash t, u \mathrel{:>} M_1(\vec{x}), M_2(\vec{x}) \Rightarrow w_1, w_2; \sigma \dashv \Delta}{\Gamma \vdash t\ u \mathrel{:>} M(\vec{x}) \Rightarrow w_1\ w_2; \sigma \dashv \Delta}\text{P}\Lambda\text{-App}$$

$$\frac{y = x_i}{\Gamma \vdash y \mathrel{:>} M(\vec{x}) \Rightarrow i; 1_\Gamma \dashv \Gamma}\text{P}\Lambda\text{-VarOk} \qquad \frac{y \notin \vec{x}}{\Gamma \vdash y \mathrel{:>} M(\vec{x}) \Rightarrow\ !;\ ! \dashv \bot}\text{P}\Lambda\text{-VarFail}$$

- Flex

$$\frac{n \vdash \vec{x} \mathrel{:>} \vec{y} \Rightarrow \vec{l}; \vec{r} \dashv p}{\Gamma, N : n \vdash N(\vec{x}) \mathrel{:>} M(\vec{y}) \Rightarrow P(\vec{l}); N \mapsto P(\vec{r}) \dashv \Gamma, P : p}\text{P}\Lambda\text{-Flex}$$

Fig. 1. Unification for pure λ-calculus (Section §2.1)

Judgments

$\Gamma \vdash \vec{t} = \vec{u} \Rightarrow \sigma \dashv \Delta \iff \sigma : \Gamma \longrightarrow \Delta$ is the most general unifier of $\vec{t}$ and $\vec{u}$

$\Gamma \vdash \vec{u} :> \overrightarrow{M(x)} \Rightarrow \vec{w}; \sigma \dashv \Delta \iff \sigma : \Gamma \longrightarrow \Delta$ extended with $M_i \mapsto w_i$ is the most general unifier of $\Gamma \vdash \vec{u}$

$m \vdash x = y \Rightarrow z \dashv p \iff$
$$\begin{tikzcd} p \arrow[r, "z"] & m \arrow[r, shift left, "x"] \arrow[r, shift right, "y"'] & \ldots \end{tikzcd}$$
is an equaliser in $\mathcal{A}$

$n \vdash x :> y \Rightarrow l; r \dashv p \iff$
$$\begin{tikzcd} p \arrow[r, "l"] \arrow[d, "r"'] & n \arrow[d, "x"] \\ \ldots \arrow[r, "y"'] & \ldots \end{tikzcd}$$
is a pullback in $\mathcal{A}$

Unification Phase

- Structural rules

$$\frac{}{\Gamma \vdash () = () \Rightarrow 1_\Gamma \dashv \Gamma} \qquad \frac{}{\bot \vdash \vec{t} = \vec{u} \Rightarrow ! \dashv \bot}$$

$$\frac{\Gamma \vdash t_1 = u_1 \Rightarrow \sigma_1 \dashv \Delta_1 \qquad \Delta_1 \vdash \vec{t_2}[\sigma_1] = \vec{u_2}[\sigma_1] \Rightarrow \sigma_2 \dashv \Delta_2}{\Gamma \vdash t_1, \vec{t_2} = u_1, \vec{u_2} \Rightarrow \sigma_1[\sigma_2] \dashv \Delta_2}\text{U-Split}$$

- Rigid-rigid

$$\frac{\Gamma \vdash \vec{t} = \vec{u} \Rightarrow \sigma \dashv \Delta}{\Gamma \vdash o(\vec{t}) = o(\vec{u}) \Rightarrow \sigma \dashv \Delta}\text{U-RigRig} \qquad \frac{o \neq o'}{\Gamma \vdash o(\vec{t}) = o'(\vec{u}) \Rightarrow ! \dashv \bot}\text{U-Clash}$$

- Flex-*, no cycle

$$\frac{M \notin u \qquad \Gamma \vdash u :> M(x) \Rightarrow w; \sigma \dashv \Delta}{\Gamma, M : m \vdash M(x) = u \Rightarrow \sigma, M \mapsto w \dashv \Delta}\text{U-NoCycle} \quad + \text{ symmetric rule}$$

- Flex-Flex, same

$$\frac{m \vdash x = y \Rightarrow z \dashv p}{\Gamma, M : m \vdash M(x) = M(y) \Rightarrow M \mapsto M'(z) \dashv \Gamma, M' : p}\text{U-Flex}$$

- Flex-Rigid, cyclic

$$\frac{M \in u \qquad u \neq M(\ldots)}{\Gamma, M : m \vdash M(x) = u \Rightarrow ! \dashv \bot}\text{U-Cyclic} \quad + \text{ symmetric rule}$$

Non-cyclic Phase

- Structural rules

$$\frac{}{\Gamma \vdash () :> () \Rightarrow (); 1_\Gamma \dashv \Gamma} \qquad \frac{}{\bot \vdash \vec{t} :> \overrightarrow{M(x)} \Rightarrow !; ! \dashv \bot}$$

$$\frac{\Gamma \vdash t_1 :> M_1(x_1) \Rightarrow u_1; \sigma_1 \dashv \Delta_1 \qquad \Delta_1 \vdash \vec{t_2}[\sigma_1] :> \overrightarrow{M_2(x_2)} \Rightarrow \vec{u_2}; \sigma_2 \dashv \Delta_2}{\Gamma \vdash t_1, \vec{t_2} :> M_1(x_1), \overrightarrow{M(x_2)} \Rightarrow u_1[\sigma_2], \vec{u_2}; \sigma_1[\sigma_2] \dashv \Delta_2}\text{P-Split}$$

- Rigid

$$\frac{\Gamma \vdash \vec{t} :> M_1(x_1^{o'}), \ldots, M_n(x_n^{o'}) \Rightarrow \vec{u}; \sigma \dashv \Delta \qquad o = o'\{x\}}{\Gamma \vdash o(\vec{t}) :> M(x) \Rightarrow o'(\vec{u}); \sigma \dashv \Delta}\text{P-Rig} \qquad \frac{o \neq \ldots\{x\}}{\Gamma \vdash o(\vec{t}) :> M(x) \Rightarrow !; ! \dashv \bot}\text{P-Fail}$$

- Flex

$$\frac{n \vdash x :> y \Rightarrow l; r \dashv p}{\Gamma, N : n \vdash N(x) :> M(y) \Rightarrow P(l); N \mapsto P(r) \dashv \Gamma, P : p}\text{P-Flex}$$

Fig. 2. Generic pattern unification algorithm (Section §??)

```
data _∈_ {A : Set} (a : A) : List A → Set where
  0 : ∀ {ℓ} → a ∈ (a :: ℓ)
  1+ : ∀ {x ℓ} → a ∈ ℓ → a ∈ (x :: ℓ)

MetaContext : Set
MetaContext = List ℕ

_⇒_ : ℕ → ℕ → Set
m ⇒ n = Vec (Fin n) m

data Tm (Γ : MetaContext) (n : ℕ) : Set where
  Var : Fin n → Tm Γ n
  App : Tm Γ n → Tm Γ n → Tm Γ n
  Lam : Tm Γ (1 + n) → Tm Γ n
  _(_) : ∀ {m} → m ∈ Γ → m ⇒ n → Tm Γ n
```

$$\frac{}{a \in (a, \dots)} \qquad \frac{a \in \ell}{a \in (x, \ell)}$$

$$\frac{1 \le i \le n}{\Gamma; n \vdash \underline{i}} \qquad \frac{\Gamma; n \vdash t \quad \Gamma; n \vdash u}{\Gamma; n \vdash t\ u} \qquad \frac{\Gamma; n+1 \vdash t}{\Gamma; n \vdash \lambda t}$$

$$\frac{M : m \in \Gamma \qquad \overbrace{x : m \Rightarrow n}^{x_1,\dots,x_m \in \{1,\dots,n\} \text{ distinct}}}{\Gamma; n \vdash M(x_1, \dots, x_m)}$$

Fig. 3. Syntax of λ-calculus (Section §2.1)

```
id : ∀{n} → n ⇒ n
id {n} = Vec.allFin n

_∘_ : ∀ {p q r} → (q ⇒ r) → (p ⇒ q) → (p ⇒ r)
xs ∘ [] = []
xs ∘ (y :: ys) = Vec.lookup xs y :: (xs ∘ ys)

_↑ : ∀ {p q} → p ⇒ q → (1 + p) ⇒ (1 + q)
_↑ {p}{q} x = Vec.insert (Vec.map Fin.inject₁ x)
                  (Fin.fromℕ p) (Fin.fromℕ q)

_{_} : ∀ {Γ n p} → Tm Γ n → n ⇒ p → Tm Γ p
App t u { f } = App (t { f }) (u { f })
Lam t { f } = Lam (t { f ↑ })
Var i { f } = Var (i { f })
M ( x ) { f } = M ( f ∘ x )
```

$$\underbrace{id}_{(1,2,\dots,n)} : n \Rightarrow n$$

$$\frac{x : q \Rightarrow r \qquad y : p \Rightarrow q}{\underbrace{x \circ y}_{(x_{y_1}, \dots, x_{y_p})} : p \Rightarrow r}$$

$$\frac{x : p \Rightarrow q}{\underbrace{x \uparrow}_{(x_1,\dots,x_p,q+1)} : p+1 \Rightarrow q+1}$$

$$\frac{\Gamma; n \vdash t \qquad x : n \Rightarrow p}{\Gamma; p \vdash t\{x\}}$$

Fig. 4. Renaming for λ-calculus (Section §2.1)

This assignation extends (through a recursive definition) to any term $\Gamma; n \vdash t$, yielding a term $\Delta; n \vdash t[\sigma]$. The base case is

$$M(x_1, \dots, x_m)[\sigma] = \sigma_M\{x\},$$

where $-\{x\}$ is variable renaming (see Figure 4). For example, the identity substitution $1_\Gamma : \Gamma \to \Gamma$ is defined by the term $M(1, \dots, m)$ for each metavariable declaration $M : m \in \Gamma$. The composition $\sigma[\sigma'] : \Gamma_1 \to \Gamma_3$ of two substitutions $\sigma : \Gamma_1 \to \Gamma_2$ and $\sigma' : \Gamma_2 \to \Gamma_3$ is defined as $M \mapsto \sigma_M[\sigma']$.

```
_⟶_ : MetaContext → MetaContext → Set
Γ ⟶ Δ = VecList (Tm Δ) Γ
```

```
_[_]t : ∀ {Γ n} → Tm Γ n → ∀ {Δ} → (Γ ⟶ Δ) → Tm Δ n
App t u [ σ ]t = App (t [ σ ]t) (u [ σ ]t)
Lam t [ σ ]t = Lam (t [ σ ]t)
Var i [ σ ]t = Var i
M ( x ) [ σ ]t = VecList.nth M σ { x }
```

$$\boxed{\dfrac{\Gamma; n \vdash t \qquad \sigma : \Gamma \to \Delta}{\Delta; n \vdash t[\sigma]}}$$

```
_[_]s : ∀ {Γ₁ Γ₂ Γ₃} → (Γ₁ ⟶ Γ₂) → (Γ₂ ⟶ Γ₃) → (Γ₁ ⟶ Γ₃)
δ [ σ ]s = VecList.map (λ _ t → t [ σ ]t) δ
```

$$\boxed{\dfrac{\sigma : \Gamma_1 \to \Gamma_2 \qquad \delta : \Gamma_2 \to \Gamma_3}{\underbrace{\sigma[\delta]}_{M\mapsto\sigma_M[\delta]} \ : \Gamma_1 \to \Gamma_3}}$$

Fig. 5. Metavariable substitution for λ-calculus (Section §2.1)

A *unifier* of two terms $\Gamma; n \vdash t, u$ is a substitution $\sigma : \Gamma \to \Gamma'$ such that $t[\sigma] = u[\sigma]$. A *most general unifier* of t and u is a unifier $\sigma : \Gamma \to \Gamma'$ that uniquely factors any other unifier $\delta : \Gamma \to \Delta$, in the sense that there exists a unique $\delta' : \Gamma' \to \Delta$ such that $\delta = \sigma[\delta']$. We denote this situation by $\Gamma \vdash t = u \Rightarrow \sigma \dashv \Gamma'$, leaving the variable context n implicit. Intuitively, the symbol $\Rightarrow$ separates the input and the output of the unification algorithm, which either returns a most general unifier, or fails when there is no unifier at all (for example, when unifying $t_1\ t_2$ with λu). The type signature of our unification algorithm is thus

```
data _⟶? (Γ : MetaContext) : Set where
  _◄_ : ∀ Δ → (Γ ⟶ Δ) → Γ ⟶?
```

```
unify : ∀ {Γ n} → Tm Γ n → Tm Γ n → Maybe (Γ ⟶?)
```

where Maybe X is an inductive type with an error constructor $\perp$ and a success constructor $\lfloor - \rfloor$ taking as argument an element of type X.

The unification algorithm recursively inspects the structure of the given terms until reaching a metavariable at the top-level, as seen in Figure 7. In the implementation, we exploit the *do* notation to propagate failure. For example, in the application case, the program fails if unify $t\ t'$ does, otherwise it continues with the success return values Δ_1, σ_1.

From a mathematical point of view, it is more convenient to handle failure by considering[2] a formal error metavariable context $\perp$ in which the only term (in any variable context) is a formal error term !, inducing a unique substitution $! : \Gamma \to \perp$, satisfying $t[!] = !$ for any term t, as demonstrated in the last case when unifying two different *rigid* term constructors (application, λ-abstraction, or variables). With this extended meaning, the inductive rule for application remains sound, in a sense that will be clarified in Section §**??**.

Unification of two metavariables applications $M(x_1, \ldots, x_m)$ and $M'(y_1, \ldots, y_{m'})$ is detailed in Figure 6. The algorithm starts by testing whether M' is in $\Gamma \backslash M$, which denotes the context Γ without M. Note that this doesn't hold precisely when $M = M'$. In this case, we need to consider the *vector of common positions* of x and y, that is, the maximal vector of (distinct) positions $(z_1, \ldots, z_p)$ such that

[2]In Section §**??**, we interpret $\perp$ as a terminal object freely added to the category of metavariable contexts and substitutions between them.

Fig. 6. Unification of two metavariables for λ-calculus (Section §2.1)

```
unify-flex-flex : ∀ {Γ m m' n} → m ∈ Γ → m ⇒ n
                                → m' ∈ Γ → m' ⇒ n → Γ ⟶?
unify-flex-flex {Γ} M x M' y with M' \? M

... | ⊥ =
  let p , z = commonPositions x y in
  Γ [ M :  p ] ◂ M ↦-( z )

... | ⌊ M' ⌋ =
  let p , l , r = commonValues x y in
  Γ \ M [ M' :  p ] ◂ M ↦ (M' :  p) ( l )
                        , M' ↦-( r )
```

$$\boxed{\frac{M : m \in \Gamma \qquad m \vdash x = y \Rightarrow z \dashv p}{\Gamma \vdash M(x) = M(y) \Rightarrow M \mapsto M(z) \dashv \Gamma[M : p]}}$$

$$\boxed{\frac{\begin{array}{c} M : m, M' : m' \in \Gamma \qquad M \neq M' \\ m \vdash x :> y \Rightarrow l; r \dashv p \end{array}}{\Gamma \vdash M(x) = M'(y) \Rightarrow \begin{pmatrix} M \mapsto M'(l) \\ M' \mapsto M'(r) \end{pmatrix} \dashv \Gamma \backslash M[M' : p]}}$$

$x_{\vec{z}} = y_{\vec{z}}$. We denote[3] such a situation by $m \vdash x = y \Rightarrow z \dashv p$. The most general unifier σ coincides with the identity substitution except that $M : m$ is replaced by a fresh metavariable $P : p$ in the context, and σ maps M to $P(z)$. In fact, instead of deleting M from the context and inserting a fresh metavariable P, the implementation keeps with M and change its arity to p resulting in a context denoted by $\Gamma[M : p]$.

Example 2.1. Let x, y, z be three distinct variables, and let us consider unification of $M(x, y)$ and $M(z, x)$. Given a unifier σ, since $M(x, y)[\sigma] = \sigma_M\{1 \mapsto x, 2 \mapsto y\}$ and $M(z, x)[\sigma] = \sigma_M\{1 \mapsto z, 2 \mapsto x\}$ must be equal, σ_M cannot depend on the variables $1, 2$. It follows that the most general unifier is $M \mapsto M'$, replacing M with a fresh constant metavariable M'. A similar argument shows that the most general unifier of $M(x, y)$ and $M(z, y)$ is $M \mapsto M'(2)$.

If $M \neq M'$, consider the vector of common values positions $(l_1, \dots, l_p)$ and $(r_1, \dots, r_p)$ between $x_1, \dots, x_m$ and $y_1, \dots, y_{m'}$, i.e., the maximal pair of lists $(\vec{l}, \vec{r})$ of distinct positions such that $x_{\vec{l}} = y_{\vec{r}}$. We denote such a situation by $m \vdash x :> y \Rightarrow l; r \dashv p$. Then, the most general unifier σ coincides with the identity substitution except that the metavariables M and M' are removed from the context and replaced by a single metavariable declaration $P : p$. Then, σ maps M to $P(l)$ and M' to $P(r)$.

Example 2.2. Let x, y, z be three distinct variables. The most general unifier of $M(x, y)$ and $N(z, x)$ is $M \mapsto N'(1), N \mapsto N'(2)$. The most general unifier of $M(x, y)$ and $N(z)$ is $M \mapsto N', N \mapsto N'$.

Again, instead of removing M and M' from the context and inserting a fresh metavariable P, the implementation removes M but keeps M' and changes its arity to p.

Unification of a metavariable application $M(x_1, \dots, x_m)$ with a term u is detailed in Figure 8.

When reaching a metavariable application $M(x_1, \dots, x_m)$ at the top-level of either term, denoting by u the other term, three situations must be considered:

(1) u is a metavariable application $N(y_1, \dots, y_n)$;
(2) u is not a metavariable application and M occurs deeply in u
(3) M does not occur in u.

Note that the first

[3]The similarity with the above introduced notation is no coincidence: as we will see (Remark ??), both are (co)equalisers.

```
unify u (M ( x )) = unify-flex-* M x u
unify (M ( x )) u = unify-flex-* M x u

unify (Lam t) (Lam t') = unify t t'

unify (App t u) (App t' u') = do
  Δ₁ ◄ σ₁ ← unify t t'
  Δ₂ ◄ σ₂ ← unify (u [ σ₁ ]t) (u' [ σ₁ ]t)
  ⌊ Δ₂ ◄ σ₁ [ σ₂ ]s ⌋

unify {Γ} (Var i) (Var j) with i Fin.≟ j
... | no _ = ⊥
... | yes _ = ⌊ Γ ◄ idₛ ⌋

unify _ _ = ⊥
```

$$\boxed{\text{See Figure 8.}}$$

$$\frac{\Gamma \vdash t = t' \Rightarrow \sigma \dashv \Delta}{\Gamma \vdash \lambda t = \lambda t' \Rightarrow \sigma \dashv \Delta}$$

$$\frac{\Gamma \vdash t = t' \Rightarrow \sigma_1 \dashv \Delta_1 \qquad \Gamma \vdash u[\sigma_1] = u'[\sigma_2] \Rightarrow \sigma_2 \dashv \Delta_2}{\Gamma \vdash t\ u = t'\ u' \Rightarrow \sigma_1[\sigma_2] \dashv \Delta_2}$$

$$\frac{i \neq j}{\Gamma \vdash \underline{i} = \underline{j} \Rightarrow\ ! \dashv \bot} \qquad \frac{}{\Gamma \vdash \underline{i} = \underline{i} \Rightarrow 1_\Gamma \dashv \Gamma}$$

$$\frac{o \neq o' \ (\textit{rigid}\ \text{term constructors})}{\Gamma \vdash o(\vec{t}) = o'(\vec{t'}) \Rightarrow\ ! \dashv \bot}$$

Fig. 7. Unification for λ-calculus (Section §2.1): main phase

Fig. 8. Unification with a metavariable application for λ-calculus (Section §2.1)

$$\frac{M \notin u \qquad \Gamma \vdash u :> M(\vec{x}) \Rightarrow w; \sigma \dashv \Delta}{\Gamma, M : m \vdash M(\vec{x}) = u \Rightarrow \sigma, M \mapsto w \dashv \Delta}\text{U}\Lambda\text{-NoCycle} \quad + \text{ symmetric rule}$$

```
unify-flex-* : ∀ {Γ m n} → m ∈ Γ → m ⇒ n → Tm Γ n → Maybe (Γ ⟶?)
unify-flex-* M x (N ( y )) = ⌊ unify-flex-flex M x N y ⌋
unify-flex-* M x u = do
  u' ← u \?ₜ M
  Δ ◄ t , σ ← unify-no-cycle u' x
  ⌊ Δ ◄ M ↦ t , σ ⌋
```

In the first case, there is no unifier because the size of both hand sides can never match after substitution. This justifies the rule UΛ-Cyclic, where $M \in u$ means that M occurs in u. In the second case, we want to unify $M(\vec{x})$ with $M(\vec{y})$. The most general unifier σ coincides with the identity substitution except for $\sigma_M = M'(\vec{z})$, where M' is fresh and $\vec{z} = (z_1, \ldots, z_p)$ is the vector of common positions, that is, a maximal vector of (distinct) positions $\vec{z}$ such that $x_{\vec{z}} = y_{\vec{z}}$. We denote[4] such a situation by $n \vdash \vec{x} = \vec{y} \Rightarrow \vec{z} \dashv p$. We therefore get the rule UΛ-Flex.

[4]The similarity with the above introduced notation is no coincidence: as we will see (Remark ??), both are (co)equalisers.