# Generic pattern unification

***Abstract*—We provide a generic second-order unification algorithm for Miller's pattern fragment. The syntax with metavariables is parameterised by a notion of signature generalising binding signatures, covering ordered $\lambda$-calculus, or (intrinsic) polymorphic syntax such as system F. The correctness of the algorithm is stated using a categorical perspective, based on the observation that the most general unifier is an equaliser in a (non "free") multi-sorted Lawvere theory, thus generalising the case of first-order unification.**

***Index Terms*—Unification, Category theory, Syntax**

## I. Introduction

Unification consists in finding a *unifier* of two terms $t, u$, that is a (metavariable) substitution $\sigma$ such that $t[\sigma] = u[\sigma]$. Unification algorithms try to compute a most general unifier $\sigma$, in the sense that given any other unifier $\delta$, there exists a unique $\delta'$ such that $\delta = \sigma[\delta']$. First-order unification [21] is used in ML-style type inference systems and logic programming languages such as Prolog. For more advanced type systems, where variable binding is crucially involved, one needs second-order unification [15], which is undecidable [12]. However, Miller [17] 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, either computes the most general unifier.

Recent results in type inference, Dunfield-Krishnaswami [7], or Jinxu et al [25], include very large proofs: the former comes with a 190 page appendix, and the latter comes with a Coq proof is 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 is parameterised by an abstract notion of signature, covering simply-typed second-order syntax, ordered syntax, or (intrinsic) polymorphic syntax such as system F. We focused on Miller pattern unification, a decidable fragment of higher-order unification where metavariables can only take distinct variables as arguments. This is already a step beyond the above-cited works [25], [7] 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.

As an introduction, we start by presenting pattern unification in the case of pure $\lambda$-calculus in Section §II-A. The goal of this paper is to generalise it, by parameterising the algorithm by a signature specifying a syntax. Then, in Section §III-A, we motivate our general setting and provide categorical semantics of the algorithm, by revisiting pure $\lambda$-calculus.

### *Related work*

First-order unification has been explained from a lattice-theoretic point of view by Plotkin [5], and later categorically analysed in [22], [11], [4, Section 9.7] as coequalisers. However, there is little work on understanding pattern unification algebraically, with the notable exception of [24], working with normalised terms of simply-typed $\lambda$-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 [9] than to the nominal sets' [10] in that everything is explicitly scoped: terms come with their support, metavariables always appear with their scope of allowed variables.

Nominal unification [23] 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 $\alpha$-equivalence as an external relation on the syntax, and as a consequence deals with freshness problems in addition to unification problems.

### *Plan of the paper*

In section §II, we present our generic pattern unification algorithm, with our generalised notion of binding signature. We introduce categorical semantics of pattern unification in Section §III. We finally show correctness of the two phases of the algorithms in Section §IV and Section §V), before presenting some applications in Section §VI.

### *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 $f : A \to B$ and $g : A' \to B$, we denote the induced morphism $A + A' \to B$ by $f, g$. Coproduct injections $A_i \to \coprod_{i\in I} A_i$ are typically denoted by $in_i$. Let $T$ be a monad on a category $\mathscr{B}$. We denote its unit by $\eta$, 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 $A \to TB$ in $\mathscr{B}$. We denote the Kleisli composition of $f : A \to TB$ and $g : B \to TC$ by $f[g] : A \to TC$.

## II. Presentation of the algorithm

### A. An example: pure λ-calculus.

Consider the syntax of pure $\lambda$-calculus extended with metavariables satisfying the pattern restriction, encoded with De Bruijn levels, rather than De Bruijn indices [6]. More formally, the syntax is inductively generated by the following inductive rules, where $\Gamma$ is a metavariable context $M_1 : m_1, \dots, M_p : m_p$ specifying a metavariable symbol $M_i$ together with its number of arguments $m_i$.

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

$$\frac{M : m \in \Gamma \quad 1 \leq i_1, \dots, i_m \leq n \quad i_1, \dots i_m \text{ distinct}}{\Gamma; n \vdash M(i_1, \dots, i_m)}$$

Note that the De Bruijn level convention means that the variable bound in $\Gamma; n \vdash \lambda t$ is $n + 1$.

A *metavariable substitution* $\sigma : \Gamma \to \Gamma'$ assigns to each metavariable $M$ of arity $m$ in $\Gamma$ a term $\Gamma'; m \vdash \sigma_M$. This assignation extends (through a recursive definition) to any term $\Gamma; n \vdash t$, yielding a term $\Gamma'; n \vdash t[\sigma]$. The base case is

$$M(x_1, \dots, x_n)[\sigma] = \sigma_M\{i \mapsto x_i\}, \tag{1}$$

where $-\{i \mapsto x_i\}$ is variable renaming. 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 $\Gamma$. Composition of substitutions $\sigma : \Gamma_1 \to \Gamma_2$ and $\sigma' : \Gamma_2 \to \Gamma_3$ is then defined as $(\sigma[\sigma'])_M = \sigma_M[\sigma']$.

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, either fails when there is no unifier at all (for example, when unifying $t_1\ t_2$ with $\lambda u$). To handle the latter case, we add[1] a formal error metavariable context $\bot$ in which the only term (in any variable context) is a formal error term !, inducing a unique substitution $! : \Gamma \to \bot$, satisfying $t[!] = !$ for any term $t$. For example, we have $\Gamma \vdash t_1\ t_2 = \lambda u \Rightarrow ! \dashv \bot$.

[1]We will interpret $\bot$ as a freely added terminal object in Section §III-A.

We generalise the notation (and thus the input of the unification algorithm) to lists of terms $\vec{t} = (t_1, \dots, t_n)$ and $\vec{u} = (u_1, \dots, u_n)$ such that $\Gamma; n_i \vdash t_i, u_i$. Then, $\Gamma \vdash \vec{t} = \vec{u} \Rightarrow \sigma \dashv \Gamma'$ means that $\sigma$ unifies each pair $(t_i, u_i)$ and is the most general one, in the sense that it uniquely factors any other substitution that unifies each pair $(t_i, u_i)$. As a consequence, we get the *congruence* rule for application.

$$\frac{\Gamma \vdash t_1, t_2 = u_1, u_2 \Rightarrow \sigma \dashv \Delta}{\Gamma \vdash t_1\ t_2 = u_1\ u_2 \Rightarrow \sigma \dashv \Delta}$$

Unifying a list of term pairs $t_1, \vec{t_2} = u_1, \vec{u_2}$ can be performed sequentially by first computing the most general unifier $\sigma_1$ of $(t_1, u_1)$, then applying the substitution to $(\vec{t_2}, \vec{u_2})$, and finally computing the most general unifier of the resulting list of term pairs: this is precisely the rule U$\Lambda$-Split in Figure 1.

Thanks to this rule, we can focus on unification of a single term pair. The idea here is to recursively inspect the structure of the given terms, until reaching a metavariable application $M(x_1, \dots, x_n)$ at top level on either hand side of $\Gamma, M : n \vdash t, u$. Assume by symmetry $t = M(x_1, \dots, x_n)$, then three mutually exclusive situations must be considered:

1) $M$ occurs deeply in $u$;
2) $M$ occurs in $u$ at top level, i.e., $u = M(y_1, \dots, y_n)$;
3) $M$ does not occur in $u$.

In the first case, there is no unifier because the size of both hand sides can never match after substitution. This justifies the rule U$\Lambda$-Cyclic, where $M \in u$ means that $M$ occurs in $u$.

In the second case, we are unifying $M(\vec{x})$ with $M(\vec{y})$. The most general unifier $\sigma$ coincides with the identity substitution except for $\sigma_M = M'(\vec{z})$, where $\vec{z} = (z_1, \dots, z_p)$ is the family of common positions $i$ such that $x_i = y_i$, and $M'$ is fresh. We denote[2] such a situation by $n \vdash \vec{x} = \vec{y} \Rightarrow \vec{z} \dashv p$. We therefore get the rule U$\Lambda$-Flex.

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

Finally, the last case consists in unifying $M(\vec{x})$ with some $u$ such that $M$ does not occur in $u$, i.e., $u$ restricts to the smaller metavariable context $\Gamma$. We denote such a situation by $u_{|\Gamma} = \underline{u'}$, where $u'$ is essentially $u$ but considered in the smaller metavariable context $\Gamma$. The algorithm then enters a *non-cyclic phase*, which specifically addresses such non-cyclic unification problems. Let us introduce a specific notation: $\Gamma \vdash u' :> M(\vec{x}) \Rightarrow w; \sigma \dashv \Delta$ means that $u'$ is a term in a metavariable context $\Gamma$, while $M$ is a fresh metavariable with respect to $\Gamma$ and $\vec{x} = (x_1, \dots, x_m)$ are distinct variables in the (implicit) variable context of $u'$. The output is the most general unifier of $u'$ and $M(\vec{x})$, both considered in the extended metavariable context $\Gamma, M : m$. This substitution from $\Gamma, M : m$ to $\Delta$ is explicitly defined as the extension of a substitution $\sigma : \Gamma \to \Delta$ with a term $\Delta; m \vdash w$ for substituting $M$.

[2]The similarity with the above introduced notation is no coincidence: as we will see, both are coequalisers.

**Judgments**

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

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

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

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

**Unification Phase**

- Structural rules

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

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

- Rigid-rigid ($o, o'$ are applications, $\lambda$-abstractions, or variables)

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

- Flex-*, no cycle

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

- 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

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

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

- Rigid

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

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

- Flex

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

Fig. 1. Unification for pure $\lambda$-calculus (Section §II-A)

*Remark* 2. The symbol :> evokes the *pruning* involved in this phase. Indeed, one intuition behind the non-cyclic unification of $M(\vec{x})$ and $u$ consists in taking $u[x_i \mapsto i]$ as a definition for $M$. This only makes sense if the free variables of $u$ are among $\vec{x}$: if $u$ is a variable that does not occur in $\vec{x}$, then obviously there is no unifier. However, it is possible to remove the *outbound* variables in $u$ if they only occur in metavariable arguments, by restricting the arities of those metavariables. We accordingly call $\sigma : \Gamma \to \Delta$ the pruning substitution. As an example, if $u$ is a metavariable application $N(\vec{x}, \vec{y})$, then although the free variables are not all included in $\vec{x}$, there is still a most general unifier, and the corresponding pruning substitution essentially replaces $N$ with $M$, discarding the outbound variables $\vec{y}$.

The previous discussion justifies the rule UΛ-NoCycle.

The non-cyclic phase proceeds recursively by introducing fresh metavariables for each argument of the top-level operation. Let us look for example at the congruence rule PΛ-Lam for $\lambda$-abstraction. Note that a fresh variable $M'$ is introduced for the body of the $\lambda$-abstraction which is supplied the bound variable $n+1$ as an argument, as it should not be pruned. Keeping in mind the intuition that $M = \lambda M'$, if $M'$ is to be substituted with $w$, then $M$ should be substituted with $\lambda w$, as seen in the conclusion.

As before, the rule PΛ-App for application motivates generalising the non-cyclic phase to handle lists. More formally given a list $\vec{u} = (u_1, \dots, u_p)$ of terms in context $\Gamma; n_i \vdash u_i$, and lists of pruning patterns $(\vec{x}_1, \dots, \vec{x}_p)$ where each $\vec{x_i}$ is a choice of distinct variables in $n_i$, the judgement $\Gamma \vdash \vec{u}$ **:>** $M_1(\vec{x}_1), \dots, M_p(\vec{x}_p) \Rightarrow \vec{w}; \sigma \dashv \Delta$ means that the substitution $\sigma : \Gamma \to \Delta$ extended with $M_i \mapsto w_i$ is the most general unifier of $\vec{u}$ and $M_1(\vec{x}_1), \dots M_p(\vec{x}_p)$ in the extended metavariable context $\Gamma, M_1 : n_1, \dots, M_p : n_p$

The sequential rule UΛ-Split can be adapted to the non-cyclic phase as in the rule PΛ-Split in Figure 1. Thanks to this sequential rule, we can focus on pruning a single term. The variable case is straightforward (rules PΛ-VarOk and PΛ-VarFail).

The remaining case consists in unifying $N(\vec{x})$ and $M(\vec{y})$. Consider the family $z_1, \dots, z_p$ of common values in $x_1, \dots, x_n$ and $y_1, \dots, y_m$, so that $z_i = x_{l_i} = y_{r_i}$ for some lists $(l_1, \dots, l_p)$ and $(r_1, \dots, r_p)$ of distinct elements of $\{1, \dots, n\}$ and $\{1, \dots, m\}$ respectively. We denote[3] such a situation by $n \vdash \vec{x}$ **:>** $\vec{y} \Rightarrow \vec{l}; \vec{r} \dashv p$. Then, the most general unifier replaces $N$ with $P(\vec{r})$ for some fresh metavariable $P$ of arity $p$, while the metavariable $M$ is replaced with $P(\vec{l})$, as seen in the rule PΛ-Flex.

**Example 3.** 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 $M \mapsto N', N \mapsto N'$.

[3]Again, the similarity with the notation for non-cyclic unification is no coincidence: both are pullbacks.

This ends our description of the unification algorithm, in the specific case of pure $\lambda$-calculus. The purpose of this work is to present a generalisation, by parameterising the algorithm by a signature specifying a syntax.

### B. Generalisation

Our algorithm is parameterised by a notion of signature that generalises binding signatures [18]. To recall, a binding signature $(O, \alpha)$ specifies for each natural number $n$ a set of $n$-ary operation symbols $O_n$ together with an arity $\alpha_o = (\overline{o}_1, \dots, \overline{o}_n)$ which is a list of natural numbers specifying how many arguments are bound in each argument. For example, pure $\lambda$-calculus is specified by $O_1 = \{lam\}$, $O_2 = \{app\}$, $O_n = \emptyset$ for any other natural number $n$, $\alpha_{app} = (0, 0)$ and $\alpha_{lam} = (1)$.

Our algorithm is parameterised by a *generalised binding signature*, or GB-signature, a notion we will formaly introduce in Definition 19. A GB-signature consists in a tuple $(\mathcal{A}, O, \alpha)$ consisting of

- a small category $\mathcal{A}$ whose objects are called *arities or* variable contexts, and morphisms are called renamings;
- for each arity $a$ and natural number $n$, a set of $n$-ary operation symbols $O_n(a)$;
- for each operation symbol $o \in O_n(a)$, a list of arities $\alpha_o = (\overline{o}_1, \dots, \overline{o}_n)$

such that $O$ and $\alpha$ are functorial in a suitable sense (see Remark 7 below for more details).

Intuitively, $O_n(a)$ is the set of operation symbols available in the variable context $a$.

**Definition 4.** The syntax specified by a GB-signature $(\mathcal{A}, O, \alpha)$ is inductively defined by the two following rules.

$$\frac{o \in O_n(a) \quad \Gamma; \overline{o}_1 \vdash t_1 \quad \dots \quad \Gamma; \overline{o}_n \vdash t_n}{\Gamma; a \vdash o(t_1, \dots, t_n)} \textsc{Rig}$$

$$\frac{M : m \in \Gamma \quad x \in \hom_{\mathcal{A}}(m, a)}{\Gamma; a \vdash M(x)} \textsc{Flex}$$

where a context $\Gamma; a$ consists of a variable context $a$ and a metavariable context $\Gamma$, that is, an metavariable arity function from a finite set of metavariable symbols to the set of objects of $\mathcal{A}$. We call a term *rigid* if it is of the shape $o(\dots)$, *flexible* if it is $M(\dots)$.

*Remark* 5. The syntax in the empty metavariable context does not depend on the morphisms in $\mathcal{A}$. In fact, by restricting the morphisms in the category of arities to identity morphisms, any GB-signature induces an indexed container [3] generating the same syntax without metavariables.

**Example 6.** Binding signatures can be compiled into GB-signatures. More specifically, a syntax specified by a binding signature $(O, \alpha)$ is also generated by the GB-signature $(\mathbb{F}_m, O', \alpha')$, where

- $\mathbb{F}_m$ is the category of finite cardinals and injections between them;
- $O_n(p) = \{v_1, \dots, v_p\} \sqcup \{o_n | o \in O_n\}$;

- $\alpha'_{v_i} = ()$ and $\alpha'_{o_n} = (n+\overline{o}_1, \ldots, n+\overline{o}_n)$ for any $o \in O$, $i, n \in \mathbb{N}$.

Note that variables $v_i$ are explicitly specified as nullary operations and thus do not require a separate formation rule, contrary to what happens with binding signatures. Moreover, the choice of renamings (i.e., morphisms in the category of arities) is motivated by the FLEX rule. Indeed, if $M$ has arity $m \in \mathbb{N}$, then a choice of arguments in the variable context $a \in \mathbb{N}$ consists of a list of distinct variables in the variable context $a$, or equivalently, an injection between the cardinal sets $m$ and $a$, that is, a morphism in $\mathbb{F}_m$ between $m$ and $a$.

GB-signatures also capture multi-sorted binding signatures such as simply-typed $\lambda$-calculus, or polymorphic syntaxes such as System F (see Section §VI).

*Remark* 7. In the notion of GB-signature, functoriality ensures that the generated syntax supports renaming: given a morphism $f : a \to b$ in $\mathcal{A}$ and a term $\Gamma; a \vdash t$, we can recursively define a term $\Gamma; b \vdash t\{f\}$. The case of metavariables is simple: $M(x)\{f\} = M(f \circ x)$. For an operation $o(t_1, \ldots, t_n)$, functoriality provides the following components:

- an operation symbol $o\{f\} \in O_n(b)$;
- a morphism $f^o_i : \overline{o}_i \to \overline{o\{f\}}_i$ for each $i \in \{1, \ldots, n\}$.

Then, $o(t_1, \ldots, t_n)\{f\}$ is defined as $o\{f\}(t_1\{f^o_1\}, \ldots, t_n\{f^o_n\})$.

Unification can be formulated by following the same route as the pure $\lambda$-calculus. A metavariable substitution $\sigma$ from $\Gamma = (M_1 : m_1, \ldots, M_p : m_p)$ to $\Delta$ is a list of terms $(\sigma_1, \ldots, \sigma_p)$ such that $\Delta; m_i \vdash \sigma_i$. Given a term $\Gamma; a \vdash t$, we can recursively define the substituted case $\Delta; a \vdash t[\sigma]$ by $o(\vec{t})[\sigma] = o(\vec{t[\sigma]})$ and $M_i(x)[\sigma] = \sigma_i\{x\}$.

Figure 2 summarises our generic algorithm, parameterised by a GB-signature, and a few more parameters: a solver for the equation

$$o = o'\{x\}, \tag{2}$$

where $o'$ is the unknown (see the rules P-RIG and P-FAIL, detailed below), and a construction of equalisers and pullbacks in $\mathcal{A}$, highlighted in blue. These are used to compute the most general unifier of two metavariable applications in the rules U-FLEX and P-FLEX. For example, for pure $\lambda$-calculus, they correspond to the rules UΛ-FLEX and PΛ-FLEX: indeed, the vector of common positions of $\vec{x}$ and $\vec{y}$ can be characterised as their equaliser in $\mathbb{F}_m$, when thinking of lists as functions from a finite domain, while the vectors of common value positions can be characterised as a pullback.

The main differences with the example of pure $\lambda$-calculus presented above in Figure 1 is that the vector notation is dropped for arguments of metavariables, since they are now morphisms in a category. Moreover, the case for operations and variables are merged in the non-cyclic phase with the rules P-RIG and P-FAIL that both are handling non-cyclic unification of $M(x)$ with $o(\vec{t})$. If $o = o'\{x\}$ for some $o'$, then P-RIG applies; otherwise, P-FAIL applies. Let us explain how they specialise for a syntax specified by a binding signature as in Example 6. In this case, $x$ is a list of distinct variables $(x_1, \ldots, x_m)$. If $o$ is a variable, then the side condition $o = o'\{x\}$ that means that $o$ is $x_{o'}$ for some $o'$. Thus, those rules specialises to the rules PΛ-VAROK and PΛ-VARFAIL. On the other hand, if $o$ is actually $o_n$, i.e., an operation symbol in variable context $n$, then $o_n = o_m\{x\}$ and thus the rule P-RIG always applies with $o' = o_m$. Moreover, $x^o_i$ is in fact the morphism from $m + \alpha_i$ to $n + \alpha_i$ defined as $x + \alpha_i$, corresponding to the list $(\vec{x}, n+1, \ldots, n+\alpha_i)$. Thus, the rule P-RIG unfolds as

$$\frac{\Gamma \vdash \vec{t} :> \ldots, M_i(\vec{x}, \overbrace{n+1, \ldots, n+1+\alpha_i}^{\text{bound variables}}), \ldots \Rightarrow \vec{u}; \sigma \dashv \Delta}{\Gamma \vdash o(\vec{t}) :> M(\vec{x}) \Rightarrow o(\vec{u}); \sigma \dashv \Delta}.$$

The correctness of the algorithm relies on additionnal assumptions on the GB-signature that we will introduce later. In particular all morphisms in $\mathcal{A}$ must be monomorphic: this ensures that Equation (2) has at most one solution (see Property 21.(i) below).

### C. Progress and termination

Each inductive rule in Figure 2 provides an elementary step for the construction of the most general unifier. To ensure that this set of rules describes a terminating algorithm, we essentially need two properties: *progress*, i.e., there is always one rule that applies given some input data, and *termination*, i.e., there is no infinite sequence of rule applications. The former is straightforward to check, but we may an explicit argument in. In this section, we sketch the proof of the latter termination property, following a standard argument. Roughly, it consists in defining the size of an input and realising that it strictly decreases in the premises. This relies on the notion of the size $|\Gamma|$ of a metavariable context $\Gamma$, as the number of its declared metavariables. We extend this definition to the case where $\Gamma = \bot$, by taking $|\bot| = 0$. We also recursively define the size $||t||$ of a term $t$ by $||M(x)|| = 1$ and $||o(\vec{t})|| = 1 + ||\vec{t}||$, with $||\vec{t}|| = \sum_i ||t_i||$. Note that no term is of empty size.

Let us first quickly justify termination of the non-cyclic phase. We define the size of a judgment $\Gamma \vdash \vec{t} :> \overrightarrow{M(x)} \Rightarrow \vec{w}; \sigma \dashv \Delta$ as $||\vec{t}||$. It is straightforward to check that the sizes of the premises are strictly smaller than the size of the conclusion, for the two recursive rules P-SPLIT and P-RIG of the non-cyclic phase, thanks to the following lemmas

**Lemma 8.** *For any term $\Gamma; a \vdash t$ and substitution $\sigma : \Gamma \to \Delta$, if $\sigma$ is a* metavariable renaming*, i.e., $\sigma_M$ is a metavariable application for any $M : m \in \Gamma$, then $||t[\sigma]|| = ||t||$.*

**Lemma 9.** *If there is a finite derivation tree of $\Gamma \vdash \vec{t} :> \overrightarrow{M(x)} \Rightarrow \vec{w}; \sigma \dashv \Delta$ and $\Delta \neq \bot$, then $|\Gamma| = |\Delta|$ and $\sigma$ is a metavariable renaming.*

Now, we tackle termination for the unification phase. We define the size of a judgment $\Gamma \vdash t = u \Rightarrow \sigma \dashv \Delta$ to be the pair $(|\Gamma|, ||t||)$. The following lemma ensures that for the two recursive rules U-SPLIT and U-RIGRIG in the unification phase, the sizes of the premises are strictly smaller than the size of the conclusion, for the lexicographic order.

**Judgments**

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

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

$$m \vdash x = y \Rightarrow z \dashv p \iff p \xrightarrow{z} m \underset{y}{\overset{x}{\rightrightarrows}} \dots \text{ is an equaliser in } \mathcal{A}$$

$$n \vdash x :> y \Rightarrow l; r \dashv p \iff \begin{array}{ccc} p & \xrightarrow{l} & n \\ {\scriptstyle r}\downarrow & & \downarrow{\scriptstyle x} \\ \dots & \xrightarrow[y]{} & \dots \end{array} \text{ is a pullback in } \mathcal{A}$$

**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

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

**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

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

- 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 §II-B)

**Lemma 10.** *If there is a finite derivation tree of* $\Gamma \vdash \vec{t} = \vec{u} \Rightarrow \sigma \dashv \Delta$*, then* $|\Gamma| \geq |\Delta|$*, and moreover if* $|\Gamma| = |\Delta|$ *and* $\Delta \neq \bot$*, then* $\sigma$ *is a renaming.*

## III. Categorical semantics

It remains to be proven that each rule is sound, e.g., for the rule U-Split, if the output of the premises are most general unifiers, then so is the conclusion. In this section, we first introduce categorical semantics for pattern unification. In Section §III-A, we relate pattern unification to coequaliser construction, and in Section §III-B, we provide a formal definition of GB-signatures with Initial Algebra Semantics for the generated syntax.

### A. Pattern unification as a coequaliser construction

In this section, we assume given a GB-signature $S = (\mathcal{A}, O, \alpha)$ and explain how most general unifiers can be thought of as equalisers in a multi-sorted Lawvere theory, as is well-known in the first-order case [22], [4]. We further more provide a formal justification for the error metavariable context $\bot$.

**Lemma 11.** *Metavariable contexts and substitutions (with their composition) between them define a category* $\mathrm{MCon}(S)$.

This relies on functoriality of GB-signatures that we will spell out formally in the next section. There, we will see that this category fully faithfuly embeds in a Kleisli category for a monad generated by $S$ on $[\mathcal{A}, \mathrm{Set}]$.

*Remark* 12. $\mathrm{MCon}(S)$ is the opposite category of a multi-sorted Lawvere theory: the sorts are the objects of $\mathcal{A}$. This theory is not freely generated by operations unless $\mathcal{A}$ is discrete. Thus, even the GB-signature induced (as in Example 6) by an empty binding signature is not "free". When $\mathcal{A}$ is discrete, we recover (multi-sorted) first-order unification.

Since a substitution is precisely a list of terms sharing the same metavariable context $\Gamma$, a unification problem for two list of terms is equivalently given by a pair of parallel substitutions $\Gamma \xrightarrow[\sigma_2]{\sigma_1} \Delta$ .

**Lemma 13.** *The most general unifier of two lists of terms* $\Delta; n_i \vdash t_i, u_i$*, if it exists, is characterised as the coequaliser of* $\vec{t}$ *as* $\vec{u}$ *as substitutions from* $(N_1 : n_1, \dots)$ *to* $\Delta$.

This justifies a common interpretation as (co)equalisers of the two variants of the notation $- \vdash - = - \Rightarrow - \dashv -$ involved in Figure 2.

Pattern unification is often stated as the existence of a coequaliser on the condition that there is a unifier. It turns out that we can get rid of this condition by considering the category $\mathrm{MCon}(S)$ freely extended with a terminal object $\bot$, as we now explain.

**Definition 14.** Given a category $\mathscr{B}$, let $\mathscr{B}_\bot$ denote the category $\mathscr{B}$ extended freely with a terminal object $\bot$.

**Notation 15.** *We denote by* ! *any terminal morphism to* $\bot$ *in* $\mathscr{B}_\bot$.

Adding a terminal object results in adding a terminal cocone to all diagrams. As a consequence, we have the following lemma.

**Lemma 16.** *Let* $J$ *be a diagram in a category* $\mathscr{B}$*. The following are equivalent:*

1) $J$ *has a colimit as long as there exists a cocone;*
2) $J$ *has a colimit in* $\mathscr{B}_\bot$.

The following result is also useful.

**Lemma 17.** *Given a category* $\mathscr{B}$*, the canonical embedding functor* $\mathscr{B} \to \mathscr{B}_\bot$ *creates colimits.*

As a consequence,

1) whenever the colimit in $\mathrm{MCon}_\bot(S)$ is not $\bot$, it is also a colimit in $\mathrm{MCon}(S)$;
2) existing colimits in $\mathrm{MCon}(S)$ are also colimits in $\mathrm{MCon}_\bot(S)$;
3) in particular, coproducts in $\mathrm{MCon}(S)$, which are computed as union of metavariable contexts, are also coproducts in $\mathrm{MCon}_\bot(S)$.

Categorically speaking, our pattern unification algorithm provides an explicit proof of the following statment.

**Theorem 18.** *Given any pattern-friendly signature* $S$*, the category* $\mathrm{MCon}_\bot(S)$ *has coequalisers.*

### B. Generalised binding signatures

**Definition 19.** A *generalised binding signature*, or *GB-signature,* is a tuple $(\mathcal{A}, O, \alpha)$ consisting of

- a small category $\mathcal{A}$ of arities and renamings between them;
- a functor $O_-(-) : \mathbb{N} \times \mathcal{A} \to \mathrm{Set}$ of operation symbols;
- a functor $\alpha : \int J \to \mathcal{A}$

where $\int J$ denotes the category of elements of $J : \mathbb{N} \times \mathcal{A} \to \mathrm{Set}$ mapping $(n, a)$ to $O_n(a) \times \{1, \dots, n\}$, defined as follows:

- objects are tuples $(n, a, o, i)$ such that $o \in O_n(a)$ and $i \in \{1, \dots, n\}$;
- a morphism between $(n, a, o, i)$ and $(n', a', o', i')$ is a morphism $f : a \to a'$ such that $n = n'$, $i = i'$ and $o\{f\} = o'$ .

We now introduce our conditions for the correctness criterion.

**Definition 20.** A GB-signature $S = (\mathcal{A}, O, \alpha)$ is said *pattern-friendly* if

1) $\mathcal{A}$ has finite connected limits,
2) all morphisms in $\mathcal{A}$ are monomorphic,
3) Each $O_n(-) : \mathcal{A} \to \mathrm{Set}$ preserves finite connected limits,
4) $\alpha$ preserves finite limits.

These conditions ensure the following two properties.

**Property 21.** *The following properties hold.*

*(i) The action of* $O_n : \mathcal{A} \to \mathrm{Set}$ *on any renaming is an injection: given any* $o \in O_n(b)$ *and renaming* $f : a \to b$*, there is at most one* $o' \in O_n(a)$ *such that* $o = o'\{f\}$.

*(ii) Let $\mathcal{L}$ be the functor $\mathcal{A}^{op} \to \mathrm{MCon}(S)$ mapping a morphism $x \in \hom_{\mathcal{A}}(m,a)$ to the substitution $(A : a) \to (M : m)$ selecting (by the Yoneda Lemma) the term $M(x)$. Then, $\mathcal{L}$ preserves finite connected colimits: it maps pullbacks and equalisers in $\mathcal{A}$ to pushouts and coequalisers in $\mathrm{MCon}(S)$.*

*Proof.* (i) Since $O_n$ preserves finite connected limits, it preserves monomorphisms because a morphism $f : a \to b$ is monomorphic if and only if the following square is a pullback [16, Exercise III.4.4].

$$\begin{array}{ccc} A & = & A \\ \| & & \downarrow f \\ A & \xrightarrow[f]{} & B \end{array}$$

(ii) The proof is deferred to the end of this section. □

The first property is used for soundness of the rules P-RIG and P-FAIL. The second one is used to justify unification of two metavariables applications as pullbacks and equalisers in $\mathcal{A}$, in the rules U-FLEXFLEX and P-FLEX.

*Remark* 22. A metavariable application $\Gamma; a \vdash M(x)$ corresponds to the composition $\mathcal{L}x[in_M]$, where $in_M$ is the embedding of $M : m$ into $\Gamma$.

The rest of this section can be safely skipped at first reading: we provide Initial Algebra Semantics for the generated syntax that we exploit to prove Property 21.(ii).

Any GB-signature $S = (\mathcal{A}, O, \alpha)$, generates an endofunctor $F_S$ on $[\mathcal{A}, \mathrm{Set}]$, that we denote by just $F$ when the context is clear, defined by

$$F_S(X)_a = \coprod_{n \in \mathbb{N}} \coprod_{o \in O_n(a)} X_{\overline{o}_1} \times \cdots \times X_{\overline{o}_n}.$$

**Lemma 23.** *$F$ is finitary and generates a free monad that restricts to a monad $T$. Moreover, $TX$ is the initial algebra of $Z \mapsto X + FZ$.*

*Proof.* $F$ is finitary because filtered colimits commute wi. In this setting, we proveth finite limits [16, Theorem IX.2.1] and colimits. The free monad construction is due to [20]. □

**Lemma 24.** *The syntax generated by a GB-signature (see Definition 4) is recovered as free algebras for $F$. More precisely, given a metavariable context $\Gamma = (M_1 : m_1, \ldots, M_p : m_p)$,*

$$T(\underline{\Gamma})_a \cong \{t | \Gamma; a \vdash t\}$$

where $\underline{\Gamma} : \mathcal{A} \to \mathrm{Set}$ is defined as the coproduct of representable functors $\coprod_i ym_i$, mapping $a$ to $\coprod_i \hom_{\mathcal{A}}(m_i, a)$.

**Notation 25.** *Given a metavariable context $\Gamma$. We sometimes denote $\underline{\Gamma}$ just by $\Gamma$.*

If $\Gamma = (M_1 : m_1, ..., M_p : m_p)$ and $\Delta$ are metavariable contexts, a Kleisli morphism $\sigma : \Gamma \to T\Delta$ is equivalently given (by the Yoneda Lemma and the universal property of coproducts) by a metavariable substitution from $\Gamma$ to $\Delta$. Moreover, Kleisli composition corresponds to composition of substitutions. This provides a formal link between the category of metavariable contexts $\mathrm{MCon}(S)$ and the Kleisli category of $T$

**Lemma 26.** *The category $\mathrm{MCon}(S)$ is equivalent to the full subcategory of $Kl_T$ spanned by coproducts of representable functors.*

We will exploit this characterisation to prove various properties of this category when the signature is *pattern-friendly*.

**Lemma 27.** *Given a GB-signature $S = (\mathcal{A}, O, \alpha)$ such that $\mathcal{A}$ has finite connected limits, $F_S$ restricts as an endofunctor on the full subcategory $\mathscr{C}$ of $[\mathcal{A}, \mathrm{Set}]$ consisting of functors preserving finite connected limits if and only if the last two conditions of Definition 20 holds.*

*Proof.* See Appendix §A. □

We now assume given a pattern-friendly signature $S = (\mathcal{A}, O, \alpha)$.

**Lemma 28.** *$\mathscr{C}$ is closed under limits, coproducts, and filtered colimits. Moreover, it is cocomplete.*

*Proof.* Cocompleteness follows from [1, Remark 1.56], since $\mathscr{C}$ is the category of models of a limit sketch, and is thus locally presentable, by [1, Proposition 1.51].

For the claimed closure property, all we have to check is that limits, coproducts, and filtered colimits of functors preserving finite connected limits still preserve finite connected limits. The case of limits is clear, since limits commute with limits. The same argument applies for coproducts and filtered colimits: they commute with finite connected limits [2, Example 1.3.(vi)]. □

**Corollary 29.** *$T$ restricts as a monad on $\mathscr{C}$ freely generated by the restriction of $F$ as an endofunctor on $\mathscr{C}$ (Lemma 43).*

*Proof.* The result follows from the construction of $T$ using colimits of initial chains, thanks to the closure properties of $\mathscr{C}$. □

We now turn to the proof of Property 21.(ii). It relies on the following lemma.

By right continuity of the homset bifunctor, any representable functor is in $\mathscr{C}$ and thus the embedding $\mathscr{C} \to [\mathcal{A}, \mathrm{Set}]$ factors the Yoneda embedding $\mathcal{A}^{op} \to [\mathcal{A}, \mathrm{Set}]$.

**Lemma 30.** *Let $\mathscr{D}$ denote the opposite category of $\mathcal{A}$ and $K : \mathscr{D} \to \mathscr{C}$ the factorisation of $\mathscr{C} \to [\mathcal{A}, \mathrm{Set}]$ by the Yoneda embedding.*

**Lemma 31.** *$K : \mathscr{D} \to \mathscr{C}$ preserves finite connected colimits.*

*Proof.* Let $y : \mathcal{A}^{op} \to [\mathcal{A}, \mathrm{Set}]$ denote the Yoneda embedding and $J : \mathscr{C} \to [\mathcal{A}, \mathrm{Set}]$ denote the canonical embedding, so that

$$y = J \circ K. \tag{3}$$

Now consider a finite connected limit $\lim F$ in $\mathcal{A}$. Then,

$$\begin{aligned}
\mathscr{C}(K \lim F, X) &\cong [\mathcal{A}, \mathrm{Set}](JK \lim F, JX) \\
&\qquad (J \text{ is fully faithful}) \\
&\cong [\mathcal{A}, \mathrm{Set}](y \lim F, JX) \qquad \text{(By (3))} \\
&\cong JX(\lim F) \qquad \text{(By the Yoneda Lemma.)} \\
&\cong \lim(JX \circ F) \\
&\qquad (X \text{ preserves finite connected limits}) \\
&\cong \lim([\mathcal{A}, \mathrm{Set}](yF-, JX)] \\
&\qquad \text{(By the Yoneda Lemma)} \\
&\cong \lim([\mathcal{A}, \mathrm{Set}](JKF-, JX)] \qquad \text{(By (3))} \\
&\cong \lim \mathscr{C}(KF-, X) \quad (J \text{ is full and faithful}) \\
&\cong \mathscr{C}(\mathrm{colim}\ KF, X) \\
&\qquad \text{(By left continuity of the hom-set bifunctor)}
\end{aligned}$$

Thus, $K \lim F \cong \mathrm{colim}\ KF$. □

*Remark* 32. Unification of two metavariables as pullbacks or equalisers in $\mathcal{A}$ crucially relies on Lemma 31, which holds because we restrict to functors preserving finite connected limits.

*Proof of Property 21.(ii).* Let $T_{|\mathscr{C}}$ be the monad $T$ restricted to $\mathscr{C}$, following Corollary 29. Since $K : \mathscr{D} \to \mathscr{C}$ preserves finite connected colimits (Lemma 31), composing it with the left adjoint $\mathscr{C} \to Kl_{T_{|\mathscr{C}}}$ yields a functor $\mathscr{D} \to Kl_{T_{|\mathscr{C}}}$ also preserving those colimits. Since it factors as $\mathscr{D} \xrightarrow{\mathcal{L}} \mathrm{MCon}(S) \hookrightarrow Kl_{T_{|\mathscr{C}}}$, where the right functor is full and faithful, $\mathcal{L}$ also preserves finite connected colimits. □

## IV. Soundness of the unification phase

In this section, we discuss soundness of some of the rules of the main unification phase in Figure 2, which computes a coequaliser in $\mathrm{MCon}_\perp(S)$. More specifically, we discuss the rule sequential rule U-Split (Section §IV-A), the rule U-FlexFlex unifying metavariable with itself (Section §IV-B), and the failing rule U-Cyclic for cyclic unification of a metavariable with a term which includes it deeply (Section §IV-C).

### A. Sequential unification (rule U-Split)

The rule U-Split follows from a stepwise construction of coequalisers valid in any category, as noted by [22, Theorem 9] when studying first-order unification: if the first two diagrams below are coequalisers, then the last one as well.

$$A_1 \underset{u_1}{\overset{t_1}{\rightrightarrows}} \Gamma \dashrightarrow^{\sigma_1} \Delta_1 \qquad \begin{array}{ccccc} & & \Gamma & & \\ & {\scriptstyle t_2}\nearrow & & \searrow{\scriptstyle \sigma_1} & \\ A_2 & & & & \Delta_1 \dashrightarrow^{\sigma_2} \Delta_2 \\ & {\scriptstyle u_2}\searrow & & \nearrow{\scriptstyle \sigma_1} & \\ & & \Gamma & & \end{array}$$

$$A_1 + A_2 \underset{u_1, u_2}{\overset{t_1, t_2}{\rightrightarrows}} \Gamma \dashrightarrow^{\sigma_2 \circ \sigma_1} \Delta_2$$

### B. Flex-Flex, same metavariable (rule U-FlexFlex)

Here we detail unification of $M(x)$ amd $M(y)$, for $x, y \in \hom_{\mathcal{A}}(m, a)$. By Remark 22, $M(x) = \mathcal{L}x[in_M]$ and $M(y) = \mathcal{L}y[in_M]$. We exploit the following lemma with $u = \mathcal{L}x$ and $v = \mathcal{L}y$.

**Lemma 33.** *In any category, denoting morphism composition $g \circ f$ by $f[g]$, the following rule applies:*

$$\frac{B \vdash u = v \Rightarrow h \dashv C}{B + D \dashv u[in_B] = v[in_B] \Rightarrow h + 1_D \dashv C + D}$$

*In other words, if the below left diagram is a coequaliser, then so is the below right diagram.*

$$A \underset{v}{\overset{u}{\rightrightarrows}} B \overset{h}{\dashrightarrow} C \qquad \begin{array}{ccccc} & {\scriptstyle u}\nearrow & B & \searrow{\scriptstyle in_B} & \\ A & & & & B + D \overset{h+1_D}{\dashrightarrow} C + D \\ & {\scriptstyle v}\searrow & B & \nearrow{\scriptstyle in_B} & \end{array}$$

It follows that it is enough to compute the coequaliser of $\mathcal{L}x$ and $\mathcal{L}y$. Furthermore, by Property 21.(ii), it is the image by $\mathcal{L}$ of the equaliser of $x$ and $y$, thus justifying the rule U-FlexFlex.

### C. Flex-rigid, cyclic (rule U-Cyclic)

The rule U-Cyclic handles unification of $M(x)$ and a term $u$ such that $u$ is rigid and $M$ occurs in $u$. In this section, we show in this section that indeed there is no unifier. More precisely, we prove in Corollary 38 below that if there is a unifier, for a term $u$ in and a metavariable application $M(x)$, then either $M$ occurs at top-level in $u$, or it does not occur at all. The argument follows the basic intuition that $t = u[M \mapsto t]$ is impossible if $M$ occurs deeply in $u$ because the sizes of both hand sides can never match. To make this statement precise, we need some recursive definitions and properties of size.

**Definition 34.** The size[4] $|t| \in \mathbb{N}$ of a term $t$ is recursively defined by $|M(x)| = 0$ and $|o(\vec{t})| = 1 + |\vec{t}|$, with $|\vec{t}| = \sum_i t_i$.

We will also need to count the occurrences of a metavariables in a term.

**Definition 35.** For any term $t$ we define $|t|_M$ recursively by $|M(x)|_M = 1$, $|N(x)|_M = 0$ if $N \neq M$, and $|o(\vec{t})|_M = |\vec{t}|_M$ with the sum convention as above for $|\vec{t}|_M$.

**Lemma 36.** *For any term $\Gamma, M : m; a \vdash t$, if $|t|_M = 0$, then factors $\Gamma; a \vdash t$. Moreover, for any $\Gamma = (M_1 : m_1, \ldots, M_n : am_n)$, and well-formed term $t$ in context $\Gamma; a$, and substitution $\sigma : \Gamma \to \Delta$, we have $|t[\sigma]| = |t| + \sum_i |t|_{M_i} \times |\sigma_i|$.*

**Corollary 37.** *For any term $t$ in context $\Gamma, M : m; a$, substitution $\sigma : \Gamma \to \Delta$, morphism $x \in \hom_{\mathcal{A}}(m, a)$ and $u$ in context $\Delta; u$, we have $|t[\sigma, M \mapsto u]| \geq |t| + |u| \times |t|_M$ and $|M(x)[u]| = |u|$.*

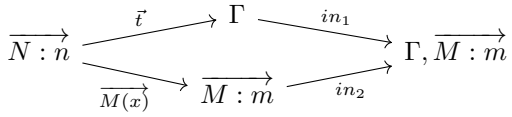
[4]The difference with the notion of size introduced in section §II-C is that metavariables are of size 0.

**Corollary 38.** *Let $t$ be a term in context $\Gamma, M : m; a$ and $x \in \hom_{\mathcal{A}}(m, a)$. Then, either $t = M(y)$ for some $y \in \hom_{\mathcal{A}}(m, a)$, or $\Gamma; a \vdash t$.*

*Proof.* Since $t[\sigma, u] = M(x)[u]$, we have $|t[\sigma, u]| = |M(x)[u]|$. Corollary 37 implies $|u| \geq |t| + |u| \times |t|_M$. Therefore, either $|t|_M = 0$ and we conclude by Lemma 36, or $|t|_M = 1$ and $|t| = 0$ and so $t$ is $M(y)$ for some $y$. □

## V. Soundness of the non-cyclic phase

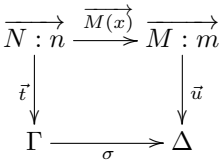
The non-cyclic phase is concerned with unifying a list of terms $\Gamma; n_i \vdash t_i$ with a list of fresh metavariable applications $M_1(x_1), \ldots, M_p(x_p)$, in the extended metavariable context $\Gamma, M_1 : m_1, \ldots, M_p : x_p$. Categorically speaking, we are looking at a coequalising diagram in $\mathrm{MCon}(S)$.




The P-Split rule is a straightforward adaption of the U-Split rule specialised to those specific coequaliser diagrams.

*Remark* 39. A unifier $\Gamma, \overrightarrow{M : m} \to \Delta$ splits into two components: a substitution $\sigma : \Gamma \to \Delta$ and a substitution $\vec{u}$ from $\overrightarrow{N : n}$ to $\overrightarrow{M : m}$ such that $t_i[\sigma] = u_i\{x_i\}$ for each $i \in \{1, \ldots, p\}$

The coequaliser $\sigma, \vec{u} : (\Gamma, \overrightarrow{M : m}) \to \Delta$ is equivalently characterised as a pushout




This justifies a common interpretation as pushouts of the two variants of the notation $- \vdash - :> - \Rightarrow -; -$ involved in Figure 2, in $\mathcal{A}^{op}$ and $\mathrm{MCon}(S)$.

In the following sections, we detail soundness of the rules for the rigid case (Section §V-A) and then for the flex case (Section §V-B).

### A. Rigid (rules P-Rig and P-Fail)

In this section, we describe soundness for the rules P-Rig and P-Fail which handle the non-cyclic unification of $M(x)$ with $\Gamma; a \vdash o(\vec{t})$ in the metavariable context $\Gamma, M : m$ for some $o \in O_n(a)$. By Remark 39, a unifier is a substitution $\sigma : \Gamma \to \Delta$ and a term $u$ such that

$$o(\vec{t}[\sigma]) = u\{x\}. \tag{4}$$

Now, $u$ is either some $M(y)$ or $o'(\vec{v})$. But in the first case, $u\{x\} = M(y)\{x\} = M(x \circ y)$, contradicting Equation (4). Therefore, $u = o'(\vec{v})$ for some $o' \in O_n(m)$ and $\vec{v} = (v_1, \ldots, v_n)$ such that $\Delta; \overline{o'}_i \vdash v_i$. Then, $u\{x\} = (o'\{x\})(v_1\{x_1^{o'}\}, \ldots, )$. It follows from Equation (4) that $o = o'\{x\}$, and $t_i[\sigma] = v_i\{x_i^{o'}\}$.

Note that there is at most one $o'$ such that $o = o'\{x\}$, by Property 21.(i). In this case, a unifier is equivalently given by a substitution $\sigma : \Gamma \to \Delta$ and a list of terms $\vec{v} = (v_1, \ldots, v_n)$ such that $\Delta; \overline{o'}_i \vdash v_i$ and $t_i[\sigma] = v_i\{x_i^{o'}\}$. But, by Remark 39, this is precisely the data for a unifier of $\vec{t}$ and $M_1(x_i^{o'}), \ldots, M_n(x_n^{o'})$. This actually induces an isomorphism between the two categories of unifiers, thus justifying the rules P-Rig and P-Fail.

### B. Flex (rule P-Flex)

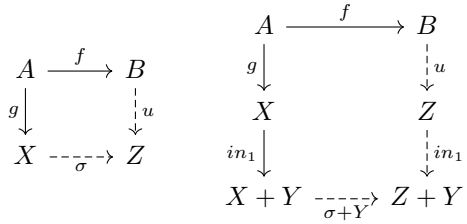
The rule P-Flex handles unification of $\Gamma, N : n; a \vdash N(x)$ and $M(y)$ where $M$ is fresh.

Note that $M(y) = \mathcal{L}y$ as a substitution $(A : a) \to (M : m)$ while $N(x) = \mathcal{L}x[in_N]$, by Remark 22. Thanks to the following lemma, it is actually enough to compute the pushout of $\mathcal{L}x$ and $\mathcal{L}y$.

**Lemma 40.** *In any category, denoting morphism composition by $f \circ g = g[f]$, the following rule applies*

$$\frac{X \vdash g :> f \Rightarrow u; \sigma \dashv Z}{X + Y \vdash g[in_1] :> f \Rightarrow u[in_1]; \sigma + Y \dashv Z + Y}$$

*In other words, if the diagram below left is a pushout, then so is the right one.*




By Property 21.(ii), the pushout of $\mathcal{L}x$ and $\mathcal{L}y$ is the image by $\mathcal{L}$ of the pullback of $x$ and $y$ in $\mathcal{A}$, thus justifying the rule P-Flex.

## VI. Applications

In this section, we present various examples of pattern-friendly signatures. We start in Section §VI-A with a variant of pure $\lambda$-calculus where metavariable arguments are sets rather than lists. Then, in Section §VI-B, we present simply-typed $\lambda$-calculus, as an example of syntax specified by a multi-sorted binding signature. Next, we introduce an example of unification for ordered syntax in Section §VI-C, and finally we present an example of polymorphic such as System F, in Section §VI-D.

### A. Arguments as sets

If we think of the arguments of a metavariable as specifying the available variables, then it makes sense to assemble them in a set rather than in a list. This motivates considering the category $\mathcal{A} = \mathbb{I}$ whose objects are natural numbers and a morphism $n \to p$ is a subset of $\{0, \ldots, p-1\}$ of cardinal $n$. For instance, $\mathbb{I}$ can be taken as subcategory of $\mathbb{F}_m$ consisting of strictly increasing injections, or as the subcategory of the

augmented simplex category consisting of injective functions. Then, a metavariable takes as argument a set of variables, rather than a list of distinct variables. In this approach, unifying two metavariables (see the rules U-FLEXFLEX and P-FLEX) amount to computing a set intersection.

### B. Simply-typed $\lambda$-calculus

In this section, we present the example of simply-typed $\lambda$-calculus. Our treatment generalises to any multi-sorted binding signature [8].

The set $T$ denote the set of simple types generated by a set of atomic types and a binary arrow type construction $- \Rightarrow -$.

Let us now describe the category $\mathcal{A}$ of arities, or variable contexts, and renamings between them. An arity $\vec{\sigma} \to \tau$ consists of a list of input types $\vec{\sigma}$ and an output type $\tau$. A term $t$ in $\vec{\sigma} \to \tau$ considered as a variable context is intuitively a well-typed term $\vec{\sigma} \vdash t : \tau$. A valid choice of arguments for a metavariable $M : (\vec{\sigma} \to \tau)$ in variable context $\vec{\sigma}' \to \tau'$ thus first requires $\tau = \tau'$, and consists of an injective renaming $\vec{r}$ between $\vec{\sigma} = (\sigma_1, \dots, \sigma_n)$ and $\vec{\sigma}' = (\sigma'_1, \dots, \sigma'_m)$, that is, a choice of distinct positions $(r_1, \dots, r_n)$ in $\{1, \dots, m\}$ such that $\vec{\sigma} = \sigma'_{\vec{r}}$.

This discussion determines the category of arities as $\mathcal{A} = \mathbb{F}_m[T] \times T$ , where $\mathbb{F}_m[T]$ is the category of finite lists of elements of $T$ and injective renamings between them. Table I summarises the definition of the endofunctor $F$ on $[\mathcal{A}, \mathrm{Set}]$ specifying the syntax, where $|\vec{\sigma}|_\tau$ denotes the number (as a cardinal set) of occurrences of $\tau$ in $\vec{\sigma}$.

The induced signature is pattern-friendly and so the generic pattern unification algorithm applies. Equalisers and pullbacks are computed following the same pattern as in pure $\lambda$-calculus. For example, to unify $M(\vec{x})$ and $M(\vec{y})$, we first compute the vector $\vec{z}$ of common positions between $\vec{x}$ and $\vec{y}$, thus satisfying $x_{\vec{z}} = y_{\vec{z}}$. Then, the most general unifier maps $M : (\vec{\sigma} \to \tau)$ to the term $P(\vec{z})$, where the arity $\vec{\sigma}' \to \tau'$ of the fresh metavariable $P$ is the only possible choice such that $P(\vec{z})$ is a valid term in the variable context $\vec{\sigma} \to \tau$, that is, $\tau' = \tau$ and $\vec{\sigma}' = \sigma_{\vec{z}}$.

### C. Ordered $\lambda$-calculus

Our setting handles linear ordered $\lambda$-calculus, consisting of $\lambda$-terms using all the variables in context. In this context, a metavariable $M$ of arity $m \in \mathbb{N}$ can only be used in the variable context $m$, and there is no freedom in choosing the arguments of a metavariable application, since all the variables must be used, in order. Thus, there is no need to even mention those arguments in the syntax. It is thus not surprising that ordered $\lambda$-calculus is already handled by first-order unification, where metavariables do not take any argument, by considering ordered $\lambda$-calculus as a multi-sorted Lawvere theory where the sorts are the variable contexts, and the syntax is generated by operations $L_n \times L_m \to L_{n+m}$ and abstractions $L_{n+1} \to L_n$.

With our generalisation, we can now handle calculi combining ordered and unrestricted variables, such as the calculus underlying ordered linear logic described in [19]. In this section we explain how our setting handles this specific example.

The set $T$ of types is generated by a set of atomic types and two binary arrow type constructions $\Rightarrow$ and $\twoheadrightarrow$. The syntax extends pure $\lambda$-calculus with a distinct application $t^{>}\ u$ and abstraction $\lambda^{>} u$. Variables contexts are of the shape $\vec{\sigma}|\vec{\omega} \to \tau$, where $\vec{\sigma}$, $\vec{\omega}$, and $\tau$ are taken in $T$. The idea is that a term in such a context has type $\tau$ and must use all the variables of $\vec{\omega}$ in order, but is free to use any of the variables in $\vec{\sigma}$. Assuming a metavariable $M$ of arity $\vec{\sigma}|\vec{\omega} \to \tau$, the above discussion about ordered $\lambda$-calculus justifies that there is no need to specify the arguments for the ordered part $\vec{\omega}$ of the context when applying $M$. Thus, a metavariable application $M(\vec{x})$ in the variable context $\vec{\sigma}'|\vec{\omega}' \to \tau'$ is well-formed if $\tau = \tau'$ and $\vec{x}$ is an injective renaming from $\vec{\sigma}$ to $\vec{\sigma}'$. Therefore, we take $\mathcal{A} = \mathbb{F}_m[T] \times T^* \times T$ for the category of arities, where $T^*$ denote the discrete category whose objects are lists of elements of $T$. The remaining components of the GB-signature are specified in Table I: we alternate typing rules for the unrestricted and the ordered fragments (variables, application, abstraction).

Pullbacks and equalisers are computed essentially as in Section §VI-B. For example, the most general unifier of $M(\vec{x})$ and $M(\vec{y})$ maps $M$ to $P(\vec{z})$ where $\vec{z}$ is the vector of common positions of $\vec{x}$ and $\vec{y}$, and $P$ is a fresh metavariable of arity $\sigma_{\vec{z}}|\vec{\omega} \to \tau$.

### D. Intrinsic polymorphic syntax

We present intrinsic System F, in the spirit of [14]. Let $S : \mathbb{F}_m \to \mathrm{Set}$ mapping $n$ to the set $S_n$ of types for system $F$ taking free type variables in $\{1, \dots, n\}$. The set of types in context $n$ is inductively generated as follows, following the De Bruijn level convention.

$$\frac{1 \leq i \leq n}{n \vdash i} \qquad \frac{n \vdash t \quad n \vdash u}{n \vdash t \Rightarrow u} \qquad \frac{n+1 \vdash t}{n \vdash \forall t}$$

Intuitively, a metavariable arity $n|\vec{\sigma} \to \tau$ specifies the number $n$ of free type variables, the list of input types $\vec{\sigma}$, and the output type $\tau$, all living in $S_n$. This provides the underlying set of objects of the category $\mathcal{A}$ of renamings. A term $t$ in $n|\vec{\sigma} \to \tau$ considered as a variable context is intuitively a well-typed term of type $\tau$ potentially involving ground variables of type $\vec{\sigma}$ and type variables in $\{1, \dots, n\}$.

A metavariable $M : (n|\sigma_1, \dots, \sigma_p \to \tau)$ in the variable context $n'|\vec{\sigma}' \to \tau'$ must be supplied with

- a choice $(\eta_1, \dots, \eta_n)$ of $n$ distinct type variables among $\{1, \dots n'\}$, such that $\tau[\vec{\eta}] = \tau'$, and
- an injective renaming $\vec{\sigma}[\vec{\eta}] \to \vec{\sigma}'$, i.e., a list of distinct positions $r_1, \dots, r_p$ such that $\vec{\sigma}[\vec{\eta}] = \sigma'_{\vec{r}}$.

This defines the data for a morphism in $\mathcal{A}$ between $(n|\vec{\sigma} \to \tau)$ and $(n'|\vec{\sigma}' \to \tau')$. The intrinsic syntax of system $F$ can then be specified as in Table I. The induced GB-signature is pattern-friendly. For example, morphisms in $\mathcal{A}$ are easily seen to be monomorphic; we detail in Appendix §B the proof of the following statement.

**Lemma 41.** *$\mathcal{A}$ has finite connected limits.*

TABLE I
EXAMPLES OF GB-SIGNATURES

Ordered $\lambda$-calculus (Section §VI-C)

| Typing rule | $O_n(\vec{\sigma}\|\vec{\omega} \to \tau) = \ldots +$ | $\alpha_o = (\ldots)$ |
|---|---|---|
| $\dfrac{x : \tau \in \Gamma}{\Gamma\|\cdot \vdash x : \tau}$ | $\{v_i \mid i \in \|\vec{\sigma}\|_\tau \text{ and } \vec{\omega} = ()\}$ | $()$ |
| $\dfrac{}{\Gamma\|x : \tau \vdash x : \tau}$ | $\{v^{>} \mid \vec{\omega} = ()\}$ | $()$ |
| $\dfrac{\Gamma\|\Omega \vdash t : \tau' \Rightarrow \tau \quad \Gamma\|\cdot \vdash u : \tau'}{\Gamma\|\Omega \vdash t\ u : \tau}$ | $\{a_{\tau'} \mid \tau' \in T\}$ | $\begin{pmatrix} \vec{\sigma}\|\vec{\omega} \to (\tau' \Rightarrow \tau) \\ \vec{\sigma}\|() \to \tau' \end{pmatrix}$ |
| $\dfrac{\Gamma\|\Omega_1 \vdash t : \tau' \twoheadrightarrow \tau \quad \Gamma\|\Omega_2 \vdash u : \tau'}{\Gamma\|\Omega_1, \Omega_2 \vdash t^{>}\ u : \tau}$ | $\{a_{\tau'}^{\vec{\omega}_1, \vec{\omega}_2} \mid \tau' \in T \text{ and } \vec{\omega} = \vec{\omega}_1, \vec{\omega}_2\}$ | $\begin{pmatrix} \vec{\sigma}\|\vec{\omega}_1 \to (\tau' \Rightarrow \tau) \\ \vec{\sigma}\|\vec{\omega}_2 \to \tau' \end{pmatrix}$ |
| $\dfrac{\Gamma, x : \tau_1\|\Omega \vdash t : \tau_2}{\Gamma\|\Omega \vdash \lambda x.t : \tau_1 \Rightarrow \tau_2}$ | $\{l_{\tau_1,\tau_2} \mid \tau = (\tau_1 \Rightarrow \tau_2)\}$ | $(\vec{\sigma}, \tau_1\|\vec{\omega} \to \tau_2)$ |
| $\dfrac{\Gamma\|\Omega, x : \tau_1 \vdash t : \tau_2}{\Gamma\|\Omega \vdash \lambda^{>} x.t : \tau_1 \twoheadrightarrow \tau_2}$ | $\{l^{>}_{\tau_1,\tau_2} \mid \tau = (\tau_1 \twoheadrightarrow \tau_2)\}$ | $(\vec{\sigma}, \tau_1\|\vec{\omega} \to \tau_2)$ |

Simply-typed $\lambda$-calculus (Section §VI-B)

| Typing rule | $O_n(\vec{\sigma} \to \tau) = \ldots +$ | $\alpha_o = (\ldots)$ |
|---|---|---|
| $\dfrac{x : \tau \in \Gamma}{\Gamma \vdash x : \tau}$ | $\{v_i \mid i \in \|\vec{\sigma}\|_\tau\}$ | $()$ |
| $\dfrac{\Gamma \vdash t : \tau' \Rightarrow \tau \quad \Gamma \vdash u : \tau'}{\Gamma \vdash t\ u : \tau}$ | $\{a_{\tau'} \mid \tau' \in T\}$ | $\begin{pmatrix} \vec{\sigma} \to (\tau' \Rightarrow \tau) \\ \vec{\sigma} \to \tau' \end{pmatrix}$ |
| $\dfrac{\Gamma, x : \tau_1 \vdash t : \tau_2}{\Gamma \vdash \lambda x.t : \tau_1 \Rightarrow \tau_2}$ | $\{l_{\tau_1,\tau_2} \mid \tau = (\tau_1 \Rightarrow \tau_2)\}$ | $(\vec{\sigma}, \tau_1 \to \tau_2)$ |

System F (Section §VI-D)

| Typing rule | $O_n(\vec{\sigma} \to \tau) = \ldots +$ | $\alpha_o = (\ldots)$ |
|---|---|---|
| $\dfrac{x : \tau \in \Gamma}{n\|\Gamma \vdash x : \tau}$ | $\{v_i \mid i \in \|\vec{\sigma}\|_\tau\}$ | $()$ |
| $\dfrac{n\|\Gamma \vdash t : \tau' \Rightarrow \tau \quad n\|\Gamma \vdash u : \tau'}{n\|\Gamma \vdash t\ u : \tau}$ | $\{a_{\tau'} \mid \tau' \in S_n\}$ | $\begin{pmatrix} n\|\vec{\sigma} \to \tau' \Rightarrow \tau \\ n\|\vec{\sigma} \to \tau' \end{pmatrix}$ |
| $\dfrac{n\|\Gamma, x : \tau_1 \vdash t : \tau_2}{n\|\Gamma \vdash \lambda x.t : \tau_1 \Rightarrow \tau_2}$ | $\{l_{\tau_1,\tau_2} \mid \tau = (\tau_1 \Rightarrow \tau_2)\}$ | $(n\|\vec{\sigma}, \tau_1 \to \tau_2)$ |
| $\dfrac{n\|\Gamma \vdash t : \forall \tau_1 \quad \tau_2 \in S_n}{n\|\Gamma \vdash t \cdot \tau_2 : \tau_1[\tau_2]}$ | $\{A_{\tau_1,\tau_2} \mid \tau = \tau_1[\tau_2]\}$ | $(n\|\vec{\sigma} \to \forall \tau_1)$ |
| $\dfrac{n+1\|wk(\Gamma) \vdash t : \tau}{n\|\Gamma \vdash \Lambda t : \forall \tau}$ | $\{\Lambda_{\tau'} \mid \tau = \forall \tau'\}$ | $(n+1\|wk(\vec{\sigma}) \to \tau')$ |

Pullbacks and equalisers in $\mathcal{A}$ are essentially computed as in Section §VI-B, by computing the vector of common (value) positions. For example, given a metavariable $M$ of arity $m|\vec{\sigma} \to \tau$, to unify $M(\vec{w}|\vec{x})$ with $M(\vec{y}|\vec{z})$, we compute the vector of common positions $\vec{p}$ between $\vec{w}$ and $\vec{y}$, and the vector of common positions $\vec{p}'$ between $\vec{x}$ and $\vec{z}$. Then, the most general unifier maps $M$ to the term $P(\vec{p}|\vec{p}')$, where $P$ is a fresh metavariable. Its arity $m'|\vec{\sigma}' \to \tau'$ is the only possible one for $P(\vec{p}|\vec{p}')$ to be well-formed in the variable context $m|\vec{\sigma} \to \tau$, that is, $m'$ is the size of $\vec{p}$, while $\tau' = \tau[p_i \mapsto i]$ and $\vec{\sigma}' = \sigma_{\vec{p}'}[p_i \mapsto i]$.

## VII. CONCLUSION

We presented a generic unification algorithm with its associated categorical semantics, parameterised by a new notion of signature for syntax with metavariables generalising Miller's pattern fragment. In the future, we would like to extend this setting to cover unification modulo equations. We also plan to see how this work applies to dependently-typed languages, going beyond polymorphic syntax. Finally, we are also interested in investigating linear syntax without restriction on the order the variables are used.

## REFERENCES


[1] Adámek, J., Rosicky, J.: Locally Presentable and Accessible Categories. Cambridge University Press (1994). https://doi.org/10.1017/CBO9780511600579

[2] Adámek, J., Borceux, F., Lack, S., Rosicky, J.: A classification of accessible categories. Journal of Pure and Applied Algebra **175**(1), 7–30 (2002). https://doi.org/https://doi.org/10.1016/S0022-4049(02)00126-3, https://www.sciencedirect.com/science/article/pii/S0022404902001263, special Volume celebrating the 70th birthday of Professor Max Kelly

[3] Altenkirch, T., Morris, P.: Indexed containers. In: Proceedings of the 24th Annual IEEE Symposium on Logic in Computer Science, LICS 2009, 11-14 August 2009, Los Angeles, CA, USA. pp. 277–285. IEEE Computer Society (2009). https://doi.org/10.1109/LICS.2009.33, https://doi.org/10.1109/LICS.2009.33

[4] Barr, M., Wells, C.: Category Theory for Computing Science. Prentice-Hall, Inc., USA (1990)

[5] D., P.G.: A note on inductive generalization. Machine Intelligence **5**, 153–163 (1970), https://cir.nii.ac.jp/crid/1573387448853680384

[6] De Bruijn, N.G.: Lambda-calculus notation with nameless dummies, a tool for automatic formula manipulation, with application to the church-rosser theorem. Indagationes Mathematicae **34**, 381–392 (1972)

[7] Dunfield, J., Krishnaswami, N.R.: Sound and complete bidirectional typechecking for higher-rank polymorphism with existentials and indexed types. Proc. ACM Program. Lang. **3**(POPL), 9:1–9:28 (2019). https://doi.org/10.1145/3290322, https://doi.org/10.1145/3290322

[8] Fiore, M.P., Hur, C.K.: Second-order equational logic. In: Proceedings of the 19th EACSL Annual Conference on Computer Science Logic (CSL 2010) (2010)

[9] Fiore, M., Plotkin, G., Turi, D.: Abstract syntax and variable binding. In: Proc. 14th Symposium on Logic in Computer Science IEEE (1999)

[10] Gabbay, M.J., Pitts, A.M.: A new approach to abstract syntax involving binders. In: Proc. 14th Symposium on Logic in Computer Science IEEE (1999)

[11] Goguen, J.A.: What is unification? - a categorical view of substitution, equation and solution. In: Resolution of Equations in Algebraic Structures, Volume 1: Algebraic Techniques. pp. 217–261. Academic (1989)

[12] Goldfarb, W.D.: The undecidability of the second-order unification problem. Theoretical Computer Science **13**(2), 225–230 (1981). https://doi.org/https://doi.org/10.1016/0304-3975(81)90040-2, https://www.sciencedirect.com/science/article/pii/0304397581900402

[13] Gray, J.W.: Fibred and cofibred categories. In: Eilenberg, S., Harrison, D.K., MacLane, S., Röhrl, H. (eds.) Proceedings of the Conference on Categorical Algebra. pp. 21–83. Springer Berlin Heidelberg, Berlin, Heidelberg (1966)

[14] Hamana, M.: Polymorphic abstract syntax via grothendieck construction (2011)

[15] Huet, G.P.: A unification algorithm for typed lambda-calculus. Theor. Comput. Sci. **1**(1), 27–57 (1975). https://doi.org/10.1016/0304-3975(75)90011-0, https://doi.org/10.1016/0304-3975(75)90011-0

[16] Mac Lane, S.: Categories for the Working Mathematician. No. 5 in Graduate Texts in Mathematics, Springer, 2nd edn. (1998)

[17] Miller, D.: A logic programming language with lambda-abstraction, function variables, and simple unification. J. Log. Comput. **1**(4), 497–536 (1991). https://doi.org/10.1093/logcom/1.4.497, https://doi.org/10.1093/logcom/1.4.497

[18] Plotkin, G.: An illative theory of relations. In: Cooper, R., et al. (eds.) Situation Theory and its Applications. pp. 133–146. No. 22 in CSLI Lecture Notes, Stanford University (1990)

[19] Polakow, J., Pfenning, F.: Properties of terms in continuation-passing style in an ordered logical framework. In: Despeyroux, J. (ed.) 2nd Workshop on Logical Frameworks and Meta-languages (LFM'00). Santa Barbara, California (Jun 2000), proceedings available as INRIA Technical Report

[20] Reiterman, J.: A left adjoint construction related to free triples. Journal of Pure and Applied Algebra **10**, 57–71 (1977)

[21] Robinson, J.A.: A machine-oriented logic based on the resolution principle. J. ACM **12**(1), 23–41 (jan 1965). https://doi.org/10.1145/321250.321253, https://doi.org/10.1145/321250.321253

[22] Rydeheard, D.E., Burstall, R.M.: Computational category theory. Prentice Hall International Series in Computer Science, Prentice Hall (1988)

[23] Urban, C., Pitts, A., Gabbay, M.: Nominal unification. In: Baaz, M., Makowsky, J.A. (eds.) Computer Science Logic. pp. 513–527. Springer Berlin Heidelberg, Berlin, Heidelberg (2003)

[24] Vezzosi, A., Abel, A.: A categorical perspective on pattern unification. RISC-Linz p. 69 (2014)

[25] Zhao, J., d. S. Oliveira, B.C., Schrijvers, T.: A mechanical formalization of higher-ranked polymorphic type inference. Proc. ACM Program. Lang. **3**(ICFP), 112:1–112:29 (2019). https://doi.org/10.1145/3341716, https://doi.org/10.1145/3341716

## APPENDIX A
## PROOF OF LEMMA 27

**Notation 42.** *Given a functor $F : D \to \mathscr{B}$, we denote the limit (resp. colimit) of $F$ by $\int_{d:D} F(d)$ or $\lim F$ (resp. $\int^{d:D} F(d)$ or $\operatorname{colim} F$), and the canonical projection $\lim F \to Fd$ by $p_d$.*

This section is dedicated to the proof of the following lemma.

**Lemma 43.** *Given a GB-signature $S = (\mathcal{A}, O, \alpha)$ such that $\mathcal{A}$ has finite connected limits, $F_S$ restricts as an endofunctor on the full subcategory $\mathscr{C}$ of $[\mathcal{A}, \mathrm{Set}]$ consisting of functors preserving finite connected limits if and only if each $O_n \in \mathscr{C}$, and $\alpha : \int J \to \mathcal{A}$ preserves finite limits.*

We first need a couple of lemmas.

**Lemma 44.** *If $\mathscr{B}$ is a small category with finite connected limits, then a functor $G : \mathscr{B} \to \mathrm{Set}$ preserves those limits if and only if $\int \mathscr{B}$ is a coproduct of filtered categories.*

*Proof.* This is a direct application of [2, Theorem 2.4 and Example 2.3.(iii)]. □

**Corollary 45.** *Assume $\mathcal{A}$ has finite connected limits. Then $J : \mathbb{N} \times \mathcal{A} \to \mathrm{Set}$ preserves finite connected limits if and only if each $O_n : \mathcal{A} \to \mathrm{Set}$ does.*

*Proof.* This follows from $\int J \cong \coprod_{n\in\mathbb{N}} \coprod_{j\in\{1,\dots,n\}} \int O_n$. □

**Lemma 46.** *Let $F : \mathscr{B} \to \mathrm{Set}$ be a functor. A functor $G : D \to \int F$ is equivalently given by a functor $H : D \to \mathscr{B}$ and an element $x \in \lim(F \circ H)$, retrieving $G$ as $Gd = (Hd, x_d)$.*

*Proof.* $\int F$ is isomorphic to the opposite of the comma category $y/F$, where $y : \mathscr{B}^{op} \to [\mathscr{B}, \mathrm{Set}]$ is the Yoneda embedding. The statement follows from the universal property of a comma category. □

**Lemma 47.** *Let $F : \mathscr{B} \to \mathrm{Set}$ preserving a limit $\lim G$. Let $x \in F \lim G$, thus inducing a functor $G_x : D \to \int F$ by the previous lemma. Then, the limit of $G_x$ is $(\lim D, x)$.*

*Proof.* Let $\mathscr{C}$ denote the full subcategory of $[\mathscr{B}, \mathrm{Set}]$ of functors preserving $\lim G$. Note that $\int F$ is isomorphic to the opposite of the comma category $K/F$, where $K : \mathscr{B}^{op} \to \mathscr{C}$ is the Yoneda embedding, which preserves $\operatorname{colim} G$, by an argument similar to the proof of Lemma 31. The forgetful functor from a comma category $L/R$ to the product of the categories creates colimits that $L$ preserve. □

**Corollary 48.** *Given categories $I$ and $\mathscr{B}$ such that any functor $d : I \to \mathscr{B}$ has a limit, if a functor $F : \mathscr{B} \to \mathrm{Set}$ preserves each such limit, then the following properties are equivalent for a functor $G : \int F \to \mathscr{B}'$.*

*(i) $G$ preserves limits over any $d' : I \to \int F$.*

*(ii) For any $d : I \to \mathscr{B}$ and $x \in F \lim d$, the canonical morphism $G(\lim d, x) \to \int_{i:I} G(d_i, Fp_i(x))$ is an isomorphism, where $p_i : \lim d \to d_i$ is the canonical projection.*

*Proof.* $(i) \Rightarrow (ii)$ Define $d' : I \to \int F$ as $d'_i = (d_i, Fp_i(x))$. The implication follows from $\lim d' = (\lim d, x)$, by Lemma 47.

$(ii) \Rightarrow (i)$ By Lemma 46, there exists $d : I \to \mathscr{B}$ and $x \in F \lim d$ such that $d'_i = (d_i, Fp_i(x))$. Again, the implication follows from $\lim d' = (\lim d, x)$, by Lemma 47. □

**Corollary 49.** *Assuming that $\mathcal{A}$ has finite connected limits and each $O_n$ preserves finite connected limits, the finite limite preservation on $\alpha : \int J \to \mathcal{A}$ of Lemma 43 can be reformulated as follows: given a finite connected diagram $d : D \to \mathcal{A}$ and element $o \in O_n(\lim d)$, the following canonical morphism is an isomorphism*

$$\overline{o}_j \to \int_{i:D} \overline{o\{p_i\}}_j$$

*for any $j \in \{1, \ldots, n\}$.*

*Proof.* This is a direct application of Corollary 48 and Corollary 45. □

**Lemma 50** (Limits commute with dependent pairs)**.** *Given functors $K : D \to \mathrm{Set}$ and $G : \int K \to \mathrm{Set}$, the following canonical morphism is an isomorphism*

$$\int_{d:D} \coprod_{x \in K(d)} G(d, x) \to \coprod_{\alpha \in \lim K} \int_d G(d, \alpha_d)$$

*Proof.* It is straightforward to check that both sets share the same universal property. □

*Proof of Lemma 43.* Let us prove the only if statement.

Let $d : D \to \mathcal{A}$ be a finite connected diagram and $X$ be a functor preserving finite connected limits. Then,

$$\begin{aligned}
\int_i F(X)_{d_i} &= \int_{i:I} \coprod_n \coprod_{o \in O_n(d_i)} X_{\overline{o}_1} \times \cdots \times X_{\overline{o}_n} \\
&\cong \coprod_n \int_{i:I} \coprod_{o \in O_n(d_i)} X_{\overline{o}_1} \times \cdots \times X_{\overline{o}_n} \\
&\qquad \text{(Coproducts commute with connected limits)} \\
&\cong \coprod_n \coprod_{o \in \int_i O_n(d_i)} \int_{i:I} X_{\overline{o_i}_1} \times \cdots \times X_{\overline{o_i}_n} \\
&\qquad \text{(By Lemma 50)} \\
&\cong \coprod_n \coprod_{o \in O_n(\lim d)} \int_{i:I} X_{\overline{o\{p_i\}}_1} \times \cdots \times X_{\overline{o\{p_i\}}_n} \\
&\qquad \text{(By assumption on } O_n(-)) \\
&\cong \coprod_n \coprod_{o \in O_n(\lim d)} \int_{i:I} X_{\overline{o\{p_i\}}_1} \times \int_{i:I} \cdots \times \int_{i:I} X_{\overline{o\{p_i\}}_n} \\
&\qquad \text{(By commutation of limits)} \\
&\cong \coprod_n \coprod_{o \in O_n(\lim d)} X_{\int_{i:I} \overline{o\{p_i\}}_1} \times \cdots \times X_{\int_{i:I} \overline{o\{p_i\}}_n} \\
&\qquad \text{(By assumption on } X) \\
&\cong \coprod_n \coprod_{o \in O_n(\lim d)} X_{\overline{o}_1} \times \cdots \times X_{\overline{o}_n} \qquad \text{(By 49)} \\
&= F(X)_{\lim d}
\end{aligned}$$

Conversely, let us assume that $F$ restricts to an endofunctor on $\mathscr{C}$. Then, $F(1) = \coprod_n O_n$ preserves finite connected limits. By Lemma 44, each $O_n$ preserves finite connected limits. Now, let us show that $\alpha : \int J \to \mathcal{A}$ preserves finite connected limits. By Corollary 49, it is enough to prove that given a finite connected diagram $d : D \to \mathcal{A}$ and element $o \in O_n(\lim d)$, the following canonical morphism is an isomorphism

$$\overline{o}_j \to \int_{i:D} \overline{o\{p_i\}}_j$$

Assume $X$ preserves finite connected limits. Then,

$$\begin{aligned}
\int_i F(X)_{d_i} &= \int_{i:I} \coprod_n \coprod_{o \in O_n(d_i)} X_{\overline{o}_1} \times \cdots \times X_{\overline{o}_n} \\
&\cong \coprod_n \int_{i:I} \coprod_{o \in O_n(d_i)} X_{\overline{o}_1} \times \cdots \times X_{\overline{o}_n} \\
&\qquad \text{(Coproducts commute with connected limits)} \\
&\cong \coprod_n \coprod_{o \in \int_i O_n(d_i)} \int_{i:I} X_{\overline{o_i}_1} \times \cdots \times X_{\overline{o_i}_n} \\
&\qquad \text{(By Lemma 50)} \\
&\cong \coprod_n \coprod_{o \in O_n(\lim d)} \int_{i:I} X_{\overline{o\{p_i\}}_1} \times \cdots \times X_{\overline{o\{p_i\}}_n} \\
&\qquad (O_n(-)\text{preserves finite connected limits)} \\
&\cong \coprod_n \coprod_{o \in O_n(\lim d)} \int_{i:I} X_{\overline{o\{p_i\}}_1} \times \int_{i:I} \cdots \times \int_{i:I} X_{\overline{o\{p_i\}}_n} \\
&\qquad \text{(By commutation of limits)} \\
&\cong \coprod_n \coprod_{o \in O_n(\lim d)} X_{\int_{i:I} \overline{o\{p_i\}}_1} \times \cdots \times X_{\int_{i:I} \overline{o\{p_i\}}_n} \\
&\qquad \text{(By assumption on } X)
\end{aligned}$$

Then, we also have

$$\begin{aligned}
\int_i F(X)_{d_i} &\cong F(X)_{\lim d} \\
&\cong \coprod_n \coprod_{o \in O_n(\lim d)} X_{\overline{o}_1} \times \cdots \times X_{\overline{o}_n}
\end{aligned}$$

This implies that each function $X_{\overline{o}_j} \to X_{\int_{i:I} \overline{o\{p_i\}}_j}$ is a bijection, or equivalently (by the Yoneda), that $\mathscr{C}(K\overline{o}_j, X) \to \mathscr{C}(K \int_{i:I} \overline{o\{p_i\}}_j, X)$ is an isomorphism. Since the Yoneda embedding is fully faithful, $\overline{o}_j \to \int_{i:D} \overline{o\{p_i\}}_j$ is an isomorphism. □

## Appendix B
## Proof of Lemma 41

In this section, we show that the category $\mathcal{A}$ of arities for System F (Section §VI-D) is finitely complete. First, note that $\mathcal{A}$ is the op-lax colimit of the functor from $\mathbb{F}_m$ to the category of small categories mapping $n$ to $\mathbb{F}_m[S_n] \times S_n$. Let us introduce the category $\mathcal{A}'$ whose definition follows that of $\mathcal{A}$, but without the output types: objects are pairs of a natural number $n$ and an element of $S_n$. Formally, this is the op-lax colimit of $n \mapsto \mathbb{F}_m[S_n]$.

**Lemma 51.** *$\mathcal{A}'$ has finite limits, and the projection functor $\mathcal{A}' \to \mathbb{F}_m$ preserves them.*

*Proof.* The crucial point is that $\mathcal{A}'$ is not only op-fibred over $\mathbb{F}_m$ by construction, it is also fibred over $\mathbb{F}_m$. Intuitively, if $\vec{\sigma} \in \mathbb{F}_m[S_n]$ and $f : n' \to n$ is a morphism in $\mathbb{F}_m$, then $f_!\vec{\sigma} \in \mathbb{F}_m[S_{n'}]$ is essentially $\vec{\sigma}$ restricted to elements of $S_n$ that are in the image of $S_f$. Note that $f_!$ is right adjoint to $\vec{\sigma} \mapsto \vec{\sigma}[f]$, and is thus continuous. We now apply [13, Theorem 4.2 and Proposition 4.1]: each $\mathbb{F}_m[S_n]$ has those limits. □

**Lemma 52.** *The projection functor $\mathcal{A} \to \mathcal{A}'$ creates finite limits.*

*Proof.* Let $d : I \to \mathcal{A}$ be a functor. We denote $d_i$ by $n_i|\vec{\sigma}_i \to \tau_i$. Let $n|\vec{\sigma}$ be the limit of $i \mapsto n_i|\vec{\sigma}_i$ in $\mathcal{A}'$. By the previous lemma, $n$ is the limit of $i \mapsto n_i$. Note that $S : \mathbb{F}_m \to \mathrm{Set}$ preserves finite connected limits. Thus, we can define $\tau \in S_n$ as corresponding to the universal function $1 \to S_n$ factorising the cone $(1 \xrightarrow{\tau_i} S_{n_i})_i$. It is easy to check that $n|\vec{\sigma} \to \tau$ is the limit of $d$. □

**Corollary 53.** *$\mathcal{A}$ is finitely complete.*