Generic pattern unification

Abstract—**We provide a generic categorical setting for Miller's pattern unification. The syntax with metavariables is generated by a free monad applied to finite coproducts of representable functors; the most general unifier is computed as a coequaliser in the Kleisli category restricted to such coproducts. Our setting handles simply-typed second-order syntax, ordered syntax, or (intrinsic) polymorphic syntax such as system F.**

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

I. 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 [19] 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 [14], which is undecidable [11]. However, Miller [16] 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 [6], or Jinxu et al [24], 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 [24], [6] 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 λ-calculus in Section §I-A. The goal of this paper is to generalise it, by parameterising the algorithm by a signature specifying a syntax. Then, in Section §II-A, we motivate our general setting and provide categorical semantics of the algorithm, by revisiting pure λ-calculus.

Related work

First-order unification has been explained from a lattice-theoretic point of view by Plotkin [4], and later categorically analysed in [20], [10], [2, Section 9.7] as coequalisers. However, there is little work on understanding pattern unification algebraically, with the notable exception of [23], 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 [8] than to the nominal sets' [9] in that everything is explicitly scoped: terms come with their support, metavariables always appear with their scope of allowed variables.

Nominal unification [22] 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.

A. An example: pure λ-calculus.

Consider the syntax of pure λ-calculus extended with metavariables satisfying the pattern restriction, encoded with De Bruijn levels, rather than De Bruijn indices [5]. More formally, the syntax is inductively generated by the following inductive rules, where Γ is a metavariable context $M_1 : m_1, \ldots, M_p : m_p$ specifying a metavariable symbol M_i together with its number of arguments m_i.

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

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

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 Γ a term $\Gamma'; m \vdash \sigma_M$. This

Judgments

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

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

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

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

Unification Phase

- Structural rules

$$\overline{\Gamma \vdash () = () \Rightarrow 1_\Gamma \dashv \Gamma} \quad \overline{\perp \vdash \vec{t} = \vec{u} \Rightarrow \, ! \dashv \perp}$$

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

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

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

- Flex-*, no cycle

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

- Flex-Flex, same

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

- Flex-Rigid, cyclic

$$\frac{u \neq M(\ldots) \qquad u_{|\Gamma} \neq \underline{\ldots}}{\Gamma, M : m \vdash M(\vec{x}) = u \Rightarrow \, ! \dashv \perp} \text{UΛ-Cyclic} \quad + \text{ symmetric rule}$$

Non-cyclic phase

- Structural rules

$$\overline{\Gamma \vdash () :> () \Rightarrow (); 1_\Gamma \dashv \Gamma} \quad \overline{\perp \vdash \vec{t} :> \overrightarrow{M(\vec{x})} \Rightarrow \, !; ! \dashv \perp}$$

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

- Rigid

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

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

- Flex

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

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

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

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

where $-\{i \mapsto x_i\}$ is variable renaming. For example, the identity substitution $1_\Gamma : \Gamma \to \Gamma$ is defined by the term $M(1, \ldots, m)$ for each metavariable declaration $M : m$ in Γ. 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 λu). 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 $\Gamma \vdash t_1\ t_2 = \lambda u \Rightarrow ! \dashv \bot$.

We generalise the notation (and thus the input of the unification algorithm) to lists of terms $\vec{t} = (t_1, \ldots, t_n)$ and $\vec{u} = (u_1, \ldots, u_n)$ such that $\Gamma; n_i \vdash t_i, u_i$. Then, $\Gamma \vdash \vec{t} = \vec{u} \Rightarrow \sigma \dashv \Gamma'$ means that σ 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 σ_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 UΛ-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 $M(x_1, \ldots, x_n)$ at top level on either hand side of $\Gamma, M : n \vdash t, u$. Assume by symmetry $t = M(x_1, \ldots, x_n)$, then three mutually exclusive situations must be considered:

1) M occurs deeply in u;
2) M occurs in u at top level, i.e., $u = M(y_1, \ldots, y_n)$;
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 UΛ-CYCLIC, where $u_{|\Gamma} \neq \underline{\ldots}$ means that u does not restrict to the smaller metavariable context Γ, and thus that M does occur in u.

In the second case, we are unifying $M(\vec{x})$ with $M(\vec{y})$. The most general unifier σ coincides with the identity substitution except for $\sigma_M = M'(\vec{z})$, where $\vec{z} = (z_1, \ldots, z_p)$ 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 UΛ-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 Γ. We denote such a situation by $u_{|\Gamma} = \underline{u'}$, where u' is essentially u but considered in the smaller metavariable context Γ. 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 Γ, while M is a fresh metavariable with respect to Γ and $\vec{x} = (x_1, \ldots, x_m)$ 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 Δ is explicitly defined as the extension of a substitution $\sigma : \Gamma \to \Delta$ with a term $\Delta; m \vdash w$ for substituting M.

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 λ-abstraction. Note that a fresh variable M' is introduced for the body of the λ-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 λw, 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 $\vec{u} = (u_1, \ldots, u_p)$ of terms in context $\Gamma; n_i \vdash u_i$, and lists of pruning patterns $(\vec{x}_1, \ldots, \vec{x}_p)$ where each $\vec{x}_i$ is a

---

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

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

choice of distinct variables in n_i, the judgement $\Gamma \vdash \vec{u} :> M_1(\vec{x}_1), \ldots, M_p(\vec{x}_p) \Rightarrow \vec{w}; \sigma \dashv \Delta$ means that the substitution $\sigma : \Gamma \to \Delta$ extended with $M_i \mapsto w_i$ is the most general unifier of $\vec{u}$ and $M_1(\vec{x}_1), \ldots M_p(\vec{x}_p)$ in the extended metavariable context $\Gamma, M_1 : n_1, \ldots, M_p : n_p$

The sequential rule UΛ-SPLIT can be adapted to the non-cyclic phase as in the rule PΛ-SPLIT in Figure 1. The usage of $+$ as a separator will be formally justified later in Remark 14: it intuitively enforces that the metavariables used in $\vec{f_2}$ are distinct from the metavariable used in f_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 $z_1, \ldots, z_p$ of common values in $x_1, \ldots, x_n$ and $y_1, \ldots, y_m$, so that $z_i = x_{l_i} = y_{r_i}$ for some lists $(l_1, \ldots, l_p)$ and $(r_1, \ldots, r_p)$ of distinct elements of $\{1, \ldots, n\}$ and $\{1, \ldots, m\}$ 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 $M \mapsto N', N \mapsto N'$.

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

B. A generic pattern unification algorithm

Our algorithm is parameterised by a notion of signature that generalises binding signatures [17]. To recall, a binding signature (O, α) specifies a set of operation symbols O together with an arity $\alpha_o = (\overline{o}_1, \ldots, \overline{o}_{\kappa_o})$ which is a list of natural numbers specifying how many arguments are bound in each argument. For example, pure λ-calculus is specified by $O = \{lam, app\}$ with $\alpha_{app} = (0, 0)$ and $\alpha_{lam} = (1)$.

Let us introduce some generalisation.

Definition 4. A *generalised binding signature*, or GB-signature, is a triple $(\mathcal{A}, O, \alpha)$ consisting of
- a category $\mathcal{A}$ whose objects are called *arities* or *variable contexts,* and morphisms are called *renamings*;
- for each arity a, a set of operation symbols O_a;
- for each operation symbol $o \in O_a$, a list of arities $\alpha_o = (\overline{o}_1, \ldots, \overline{o}_{\kappa_o})$

such that O and α are functorial in a suitable sense (see Remark 8 below for more details). We call an element of $\{1, \ldots, \kappa_o\}$ an *o-index*.

Intuitively, O_a is the set of operation symbols available in the variable context a.

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

Definition 5. The syntax specified by a GB-signature $(\mathcal{A}, O, \alpha)$ is inductively defined by the two following rules.

$$\frac{o \in O_a \quad \Gamma; \overline{o}_1 \vdash t_1 \quad \ldots \quad \Gamma; \overline{o}_{\kappa_o} \vdash t_{\kappa_o}}{\Gamma; a \vdash o(t_1, \ldots, t_{\kappa_o})} \text{RIG}$$

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

where a context $\Gamma; a$ consists of a variable context a and a metavariable context Γ, that is, a list of metavariable symbols with their associated arities.

Remark 6. Note that 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 [1] generating the same syntax without metavariables.

Example 7. Binding signatures can be compiled into GB-signatures. More specifically, a syntax specified by a binding signature (O, α) 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;
- $O'_n = \{v_1, \ldots, v_n\} \sqcup \{o_n | o \in O\}$;
- $\alpha'_{v_i} = ()$ and $\alpha'_{o_n} = n + \alpha_o$ 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 λ-calculus, or polymorphic syntaxes such as System F (see Section §VIII).

Remark 8. The syntax generated by a GB-signature supports renaming: given a morphism $f : a \to b$ 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 $o(t_1, \ldots, t_{\kappa_o})$, functoriality provides the following components:
- an operation symbol $o\{f\} \in O_b$;
- a list of natural numbers $(\epsilon(o, f)_1 \ldots, \epsilon(o, f)_{\kappa_{o\{f\}}})$ between 1 and κ_o;
- a morphism $f_i^o : \overline{o}_{\epsilon(o,f)_i} \to \overline{o\{f\}}_i$ for each $o\{f\}$-index i.

Then, $o(t_1, \ldots, t_{\kappa_o})\{f\}$ is defined as $o\{f\}(t_{\epsilon(o,f)_1}\{f_1^o\}, \ldots)$.

Remark 9. In our examples, $\kappa_{o\{f\}}$ is always equal to κ_o and $(\epsilon(o, f)_1 \ldots, \epsilon(o, f)_{\kappa_{o\{f\}}})$ is $(1, \ldots, \kappa_o)$, so that f_i^o is a morphism between $\overline{o}_i$ and $\overline{o\{f\}}_i$ and thus $o(t_1, \ldots, t_{\kappa_o})\{f\} = o\{f\}(t_1\{f_1^o\}, \ldots, t_{\kappa_o}\{f_{\kappa_o}^o\})$.

Unification can be formulated by following the same route as the pure λ-calculus. A metavariable substitution σ from

$\Gamma = (M_1 : m_1, \ldots, M_p : m_p)$ to Δ 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

$$o = o'\{x\}, \qquad (2)$$

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 λ-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 λ-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 7. 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_i^o 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.

Plan of the paper

In Section §II, we present our general categorical setting. In Section §III, we state the existence of the most general unifier as a categorical property. Then we describe the construction of the most general unifier, as summarised in Figure **??**, starting with the unification phase (Section §IV), the non-cyclic phase (Section §V), the occur-check (Section §VI). We finally justify completeness in Section §VII.

General notations

Given $n \in \mathbb{N}$, we denote the set $\{0, \ldots n-1\}$ by $\underline{n}$. $\mathscr{B}^{op}$ denotes the opposite category of $\mathscr{B}$. If $\mathscr{B}$ is a category and a and b are two objects, we denote the set of morphisms between a and b by $\hom_{\mathscr{B}}(a, b)$ or $\mathscr{B}(a, b)$. We denote the identity morphism at an object x by 1_x. We denote by $()$ any initial morphism and by ! any terminal morphism. 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 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 $A \to TB$ 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. GENERAL SETTING

In our setting, syntax is specified as an endofunctor F on a category $\mathscr{C}$. We introduce conditions for the latter in Section §II-B and for the former in Section §II-C. Examples are presented in Section §VIII.

A. Categorification

TODO: redoit In this section, we define the syntax of pure λ-calculus from a categorical point of view in order to motivate our general categorical setting. We then explain the semantics of the generic unification algorithm summarised in Figure **??**.

Consider the category of functors $[\mathbb{F}_m, \text{Set}]$ from $\mathbb{F}_m$, the category of finite cardinals and injections between them, to the category of sets. A functor $X : \mathbb{F}_m \to \text{Set}$ can be thought of as assigning to each natural number n a set X_n of expressions with free variables taken in the set $V_n = \{1, \ldots, n\}$. The action on morphisms of $\mathbb{F}_m$ means that these expressions support injective renamings. Note that this structure is no more than what is needed to substitute a metavariable in Equation (1).

Pure λ-calculus defines such a functor Λ by $\Lambda_n = \{t \mid \cdot; n \vdash t\}$, satisfying the recursive equation $\Lambda_n \cong V_n + \Lambda_n \times \Lambda_n + \Lambda_{n+1}$, where $-+-$ is disjoint union. In fact, Λ can be defined as the initial algebra for the endofunctor F on $[\mathbb{F}_m, \text{Set}]$ mapping X to $V + X \times X + X_{-+1}$. In pattern unification, we need to consider extensions of this syntax with metavariables. Let us start with a metavariable M of arity m. The extended syntax $\Lambda^{M:m}$ defined by $\Lambda_n^{M:m} = \{t \mid M : m; n \vdash t\}$ now can be characterised as the initial algebra for the endofunctor mapping X to $F(X) + arg^{M:m}$, where $arg^{M:m} : \mathbb{F}_m \to \text{Set}$ is the functor which provides the set of valid arguments for M in the variable context n, when evaluated at n. We recognise the usual construction of free algebras: denoting T the free monad generated by F, the functor $\Lambda^{M:m}$ is the free F-algebra $T(arg^{M:m})$ on $arg^{M:m}$.

Let us have a closer look at the functor $arg^{M:m}$. According to the pattern restriction, $arg_n^{M:m}$ is isomorphic to the set of injections between the cardinal sets m and n, or equivalently,

Judgments

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

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

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

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

Unification Phase

- Structural rules

$$\overline{\Gamma \vdash () = () \Rightarrow 1_\Gamma \dashv \Gamma} \quad \overline{\bot \vdash \vec{t} = \vec{u} \Rightarrow \,! \dashv \bot}$$

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

- Rigid-rigid

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

- Flex-*, no cycle

$$\frac{u_{|\Gamma} = \underline{u'} \quad \Gamma \vdash u' :> M(x) \Rightarrow w; \sigma \dashv \Delta}{\Gamma, M : m \vdash M(x) = u \Rightarrow \sigma, M \mapsto w \dashv \Delta} \text{U-NOCYCLE} \quad + \text{ symmetric rule}$$

- 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

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

Non-cyclic phase

- Structural rules

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

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

- Rigid

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

- 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 §I-B)

the set of morphisms $\mathbb{F}_m(m, n)$ between m and n in the category $\mathbb{F}_m$. Thus, $arg^{M:m}$ can be taken as the representable functor $\mathbb{F}_m(m, -)$, which we denote by ym.

It follows from the previous discussion that $\Lambda^{M:m}$ is the free F-algebra on ym. In fact, the argument works for any metavariable context Γ. More precisely, using the following notation, we see that $T(\underline{\Gamma})_n$ is the set of terms in the context $\Gamma; n$.

Notation 10. Given a metavariable context $\Gamma = (M_1 : m_1, \ldots, M_p : m_p)$, we denote the finite coproduct $\coprod_{i \in \{1, \ldots, p\}} ym_i$ by $\underline{\Gamma}$, or just by Γ when the context is clear.

In the view to further abstracting pattern unification, these observations motivate considering functor categories $[\mathcal{A}, \text{Set}]$, where $\mathcal{A}$ is a small category where all morphisms are monomorphic (to account for the pattern condition enforcing that metavariable arguments are distinct variables), together with an endofunctor[4] F on it. Then, the abstract definition of a syntax extended with metavariables is the free F-algebra monad T applied to a finite coproduct of representable functors.

Let us investigate how a unification problem is stated in this general setting[5]. Given metavariable contexts $\Gamma = (M_1 : m_1, \ldots, M_p : m_p)$ and Δ, a Kleisli morphism $\sigma : \Gamma \to T\Delta$ is equivalently given (by the Yoneda Lemma and the universal property of coproducts) by a λ-term $\Delta; m_i \vdash \sigma_i$ for each $i \in \{1, \ldots, p\}$: this is precisely the data for a metavariable substitution $\Delta \to \Gamma$. Thus, Kleisli morphisms are nothing but metavariable substitutions. Moreover, Kleisli composition corresponds to composition of substitutions.

A unification problem can be stated as a pair of parallel Kleisli morphisms $yp \underset{u}{\overset{t}{\rightrightarrows}} T\Gamma$, corresponding (by the Yoneda Lemma) to selecting a pair of elements in $T\Gamma_p$, or equivalently, a pair of terms $\Gamma; p \vdash t, u$. A unifier is nothing but a Kleisli morphism coequalising this pair. The property required by the most general unifier means that it is the coequaliser in the full subcategory spanned by objects of the shape $\underline{\Gamma}$. The main purpose of the pattern unification algorithm consists in constructing this coequaliser, if it exists, which is the case as long as there exists a unifier, as stated in Section §III.

With this in mind, we can give categorical semantics to the unification notation in Figure **??** as follows.

Notation 11. We denote a coequaliser $A \underset{u}{\overset{t}{\rightrightarrows}} \Gamma -\!\!\overset{\sigma}{-}\!\!\succ \Delta$ in a category $\mathscr{B}$ by $\Gamma \vdash t =_{\mathscr{B}} u \Rightarrow \sigma \dashv \Delta$, sometimes even omitting $\mathscr{B}$.

This notation is used in the unification phase, taking $\mathscr{B}$ to be the Kleisli category of T restricted to coproducts of representable functors, and extended with a free terminal

---

[4] In Section §II-C, we make explicit assumptions about this endofunctor for the unification algorithm to properly generalise.

[5] What follows is a generalisation of the first-order case detailed in [20], [2], in the sense that we consider a free monad on more general presheaf categories than categories of sets indexed by a fixed set of sorts.

(error) object $\perp$ (as formally justified in Section §III), with the exception of the premise of the rule U-FLEXFLEX, where $\mathscr{B} = \mathscr{D}$ is the opposite category of $\mathbb{F}_m$. The latter corresponds to the above rule (**??**): the premise precisely means that $p \overset{\vec{z}}{\longrightarrow} m \underset{\vec{y}}{\overset{\vec{x}}{\rightrightarrows}} n$ is indeed an equaliser in $\mathbb{F}_m$, where n is the (implicit) variable context.

Remark 12. In Notation 11, when A is a coproduct $yn_1 + \cdots + yn_p$, then t and u can be thought of as lists of terms $\Gamma; n_i \vdash t_i, u_i$, hence the vector notation used in various rules (e.g., U-SPLIT). Moreover, the usage of comma as a list separator in the conclusion is formally justified by the notation $a + c \overset{f,g}{\longrightarrow} b$ given morphisms $a \overset{f}{\rightarrow} b \overset{g}{\leftarrow} c$.

Note that the rule U-SPLIT is in fact valid in any category (see [20, Theorem 9]). To formally understand the rule U-NOCYCLE which we have already introduced in the case of pure λ-calculus, see (**??**), let us provide the non-cyclic notation with categorical semantics.

Notation 13. We denote a pushout diagram in a category $\mathscr{B}$ as below left by the notation as below right, sometimes even omitting $\mathscr{B}$.

$$
\begin{array}{ccc}
A & \overset{f}{\longrightarrow} & \Gamma' \\
{\scriptstyle t}\downarrow & & \downarrow{\scriptstyle u} \\
\Gamma & \underset{\sigma}{-\!\!-\!\!\succ} & \ulcorner\Delta
\end{array}
\qquad \Leftrightarrow \qquad \Gamma \vdash_{\mathscr{B}} t :> f \Rightarrow u; \sigma \dashv \Delta
$$

Similarly to Notation 11, this is used in Figure **??**, taking $\mathscr{B}$ to be the Kleisli category of T restricted to coproducts of representable functors, and extended with an error object $\perp$, with the exception of the premise of the rule P-FLEX, where $\mathscr{B} = \mathscr{D}$ is the opposite category of $\mathbb{F}_m$. The latter corresponds to the above rule (**??**) whose premise precisely means that the following square is a pullback in $\mathbb{F}_m$.

$$
\begin{array}{ccc}
p & \overset{l}{\longrightarrow} & n \\
{\scriptstyle r}\downarrow & & \downarrow{\scriptstyle x} \\
m & \underset{y}{\longrightarrow} & C
\end{array}
$$

Remark 14. Let us add a few more comments about Notation 13. First, note that if A is a coproduct $yn_1 + \cdots + yn_p$, then t and u can be thought of as lists of terms $\Gamma; n_i \vdash t_i$ and $\Gamma'; n_i \vdash u_i$. In fact, in the situations we will consider, f will be of the shape $yn_1 + \cdots + yn_p \xrightarrow{f_1 + \cdots + f_p} ym_1 + \cdots + ym_p$. This explains our usage of $+$ as a list separator in the rule P-SPLIT.

We will explain the rules P-RIG and P-FAIL later TODO.

B. Base category

We work in a full subcategory $\mathscr{C}$ of functors $\mathcal{A} \to \text{Set}$, namely, those preserving finite connected limits, where $\mathcal{A}$ is a small category in which all morphisms are monomorphisms and which has finite connected limits.

Example 15. In Section §**??**, we considered $\mathcal{A} = \mathbb{F}_m$ the category of finite cardinals and injections. Note that $\mathscr{C}$ is equivalent to the category of nominal sets [9].

Remark 16. The main property that justifies unification of two metavariables as an equaliser or a pullback in $\mathcal{A}$ is that given any metavariable context Γ, the functor $T\Gamma : \mathcal{A} \to \mathrm{Set}$ preserves them, i.e., $T\Gamma \in \mathscr{C}$. In fact, the argument works not only in the category of metavariable contexts and substitutions, but also in the (larger) category of objects of $\mathscr{C}$ and Kleisli morphisms between them. However, counter-examples can be found in the total Kleisli category. Consider indeed the unification problem $M(x, y) = M(y, x)$, in the example of pure λ-calculus. We can define[6] a functor P that does not preserve finite connected colimits such that $T(P)$ is the syntax extended with a binary commutative metavariable $M'(-, -)$. Then, the most general unifier, computed in the total Kleisli category, replaces M with P. But in the Kleisli category restricted to coproducts of representable functors, or more generally, to objects of $\mathscr{C}$, the coequaliser replaces M with a constant metavariable, as expected.

Remark 17. The category $\mathcal{A}$ is intuitively the category of metavariable arities. A morphism in this category can be thought of as data to substitute a metavariable $M : a$ with another. For example, in the case of pure λ-calculus, replacing a metavariable $M : m$ with a metavariable $N : n$ amounts to a choice of distinct variables $x_1, \ldots, x_n \in \{0, \ldots, m-1\}$, i.e., a morphism $\hom_{\mathbb{F}_m}(n, m)$. The condition that all morphisms are monomorphic can be thought of as an abstract version of the pattern restriction.

Lemma 18. *$\mathscr{C}$ is closed under limits, coproducts, and filtered colimits. Moreover, it is cocomplete.*

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 $\mathcal{A}^{op} \to [\mathcal{A}, \mathrm{Set}]$.

Notation 19. We set $\mathscr{D} = \mathcal{A}^{op}$ and denote the fully faithful Yoneda embedding as $\mathscr{D} \xrightarrow{K} \mathscr{C}$. We denote by $\mathscr{D}^+ \xrightarrow{K^+} \mathscr{C}$ the full subcategory of $\mathscr{C}$ consisting of finite coproducts of objects of $\mathscr{D}$. Moreover, we adopt Notation 10 for objects of $\mathscr{D}^+$, that is, a coproduct $\coprod_{i \in \{M, N, \ldots\}} Ka_i$ is denoted by a *(metavariable) context* $M : a_M, N : a_N, \ldots$

Remark 20. $\mathscr{D}^+$ is equivalent to the category of finite families of objects of $\mathcal{A}$. Thinking of objects of $\mathcal{A}$ as metavariable arities (Remark 17), $\mathscr{D}^+$ can be thought of as the category of metavariable contexts.

We now abstract the situation by listing a number of properties that we will use to justify the unification algorithm.

Property 21. *The following properties hold.*
(i) $\mathscr{D}$ has finite connected colimits.
(ii) $K : \mathscr{D} \to \mathscr{C}$ preserves finite connected colimits.
(iii) Given any morphism $f : a \to b$ in $\mathscr{D}$, the morphism Kf is epimorphic.

[6]Define P_n as the set of two-elements sets of $\{0, \ldots, n-1\}$.

(iv) Coproduct injections $A_i \to \coprod_j A_j$ in $\mathscr{C}$ are monomorphisms.
(v) For each $d \in \mathscr{D}$, the object Kd is connected, i.e., any morphism $Kd \to \coprod_i A_i$ factors through exactly one coproduct injection $A_j \to \coprod_i A_i$.

Proof. See Appendix §A. $\qquad \square$

Remark 22. Continuing Remark 16, unification of two metavariables as pullbacks or equalisers in $\mathcal{A}$ crucially relies on Property 21.(ii), which holds because we restrict to functors preserving finite connected limits.

C. The endofunctor for syntax

We assume given an endofunctor F on $[\mathcal{A}, \mathrm{Set}]$ defined by

$$F(X)_a = \coprod_{o \in O_a} \prod_{j \in J_{o,a}} X_{L_{o,j,a}},$$

for some functors $O : \mathcal{A} \to \mathrm{Set}$, $J : (\int O)^{op} \to \mathbb{F}$ and $L : (\int J)^{op} \to \mathcal{A}$, , where
- $\mathbb{F}$ is the category of finite cardinals and any morphisms between them;
- $\int O$ denotes the category of elements of O whose objects are pairs of an object a of $\mathcal{A}$ and an element o in O_a, and morphisms between (a, o) and (a', o') are morphisms $f : a \to a'$ such that $O_f(o) = o'$;
- $\int J$ denotes the category of elements of $\int O \xrightarrow{J} \mathbb{F} \hookrightarrow$ Set. Objects are triples (a, o, j), where a is an object of $\mathcal{A}$, $o \in O_a$, and $j \in \{0, \ldots, J_{a,o} - 1\}$, and a morphism in $\int J$ between (a, o, j) and (a', o', j') is a morphism $f : a \to a'$ such that $o = O_f(o')$ and $j' = J_f(j)$.

Example 23. For pure λ-calculus where $\mathcal{A} = \mathbb{F}_m$, we have $O_n = \{a, l\} + \{v_i | 1 \le i \le n\}$, and $J_{v_i} = 0$, $J_a = 2$, $J_l = 1$, and $L_{n,o,j} = n + 1$ is if $o = l$, or n otherwise.

We moreover assume that F restricts as an endofunctor on $\mathscr{C}$, i..e, that it maps functors preserving finite connected limits to functors preserving finite connected limits. This has the following consequence.

Remark 24. F induces a polynomial functor on the category of sets indexed by the objects of $\mathcal{A}$.

Lemma 25. *O preserves finite connected limits.*

Proof. O is isomorphic to $F(1)$, where 1 is the constant functor mapping everything to the singleton set $\{0\}$. Since 1 trivially preserves limits, it is in $\mathscr{C}$ and thus $F(1) \cong O$ also is. $\qquad \square$

Lemma 26. *F is finitary and generates a free monad that restricts to a monad T on $\mathscr{C}$. Moreover, TX is the initial algebra of $Z \mapsto X + FZ$, as an endofunctor on $\mathscr{C}$.*

Proof. F is finitary because filtered colimits commute with finite limits [15, Theorem IX.2.1] and colimits. The free monad construction is due to [18]. $\qquad \square$

We will be mainly interested in coequalisers in the Kleisli category restricted to objects of $\mathscr{D}^+$.

Notation 27. Let $Kl_{\mathscr{D}+}$ denote the full subcategory of Kl_T consisting of objects in $\mathscr{D}^+$. Moreover, we denote by $\mathcal{L}^+ : \mathscr{D}^+ \to Kl_{\mathscr{D}+}$ the functor which is the identity on objects and postcomposes any morphism $A \to B$ by $\eta_B : B \to TB$, and by $\mathcal{L}$ the functor $\mathscr{D} \hookrightarrow \mathscr{D}^+ \xrightarrow{\mathcal{L}^+} Kl_{\mathscr{D}+}$.

Property 28. *The functor $\mathscr{D} \xrightarrow{\mathcal{L}} Kl_{\mathscr{D}+}$ preserves finite connected colimits.*

Notation 29. Given $f \in \hom_{\mathscr{D}}(a,b)$, $u : Kb \to X$, we denote $u \circ Kf$ by $f \cdot u$.

Given $a \in \mathscr{D}$, $o : Ka \to O$, we denote $\coprod_{j \in J_{a,o}} KL_{a,o,j}$ by $\overline{o}$. Given $f \in \hom_{\mathscr{D}}(b,a)$, we denote the induced morphism $\overline{f \cdot o} \to \overline{o}$ by f^o.

Lemma 30. *For any $X \in \mathscr{C}$, a morphism $Ka \to FX$ is equivalently given by a morphism $o \in Ka \to O$, and a morphism $f : \overline{o} \to X$.*

Proof. This follows from Property 21.(v). $\qquad\square$

Notation 31. Given $o : Ka \to O$ and $\vec{t} : \overline{o} \to TX$, we denote the induced morphism $Ka \to FTX \hookrightarrow TX$ by $o(\vec{t})$, where the first morphism $Ka \to FTX$ is induced by Lemma 30.

Let $\Gamma = (M_1 : a_1, \dots, M_p : a_p) \in \mathscr{D}^+$ and $x \in \hom_{\mathscr{D}}(a, a_i)$, we denote the Kleisli composition $Ka \xrightarrow{\mathcal{L}x} Ka_i \xrightarrow{in_i} \Gamma$ by $M_i(x) \in \hom_{Kl_T}(Ka, \Gamma) = \hom_{\mathscr{C}}(Ka, T\Gamma)$.

Property 32. *Let $\Gamma = M_1 : a_1, \dots, M_n : a_n \in \mathscr{D}^+$. Then, any morphism $u : Ka \to T\Gamma$ is one of the two mutually exclusive following possibilities:*

- *$M_i(x)$ for some unique i and $x : a \to a_i$,*
- *$o(\vec{t})$ for some unique $o : Ka \to O$ and $\vec{t} : \overline{o} \to T\Gamma$.*

We say that u is flexible (flex) *in the first case and* rigid *in the other case.*

Property 33. *Let $\Gamma = M_1 : a_1, \dots, M_n : a_n \in \mathscr{D}^+$ and $\sigma : \Gamma \to T\Delta$. Then, for any $o : Ka \to O$, $\vec{t} : \overline{o} \to T\Gamma$, $u : b \to a$, $i \in \{1, \dots, n\}$, $x : a \to a_i$,*

$$o(\vec{t})[\sigma] = o(\vec{t}[\sigma]) \qquad M_i(x)[\sigma] = x \cdot \sigma_i$$
$$u \cdot (o(\vec{t})) = (u \cdot o)(\vec{t} \circ u^o) \qquad u \cdot M_i(x) = M_i(x \circ u)$$

III. MAIN RESULT

The main point of pattern unification is that a pair of parallel morphisms in $Kl_{\mathscr{D}+}$ either has no unifier, or has a coequaliser. Working with this logical disjunction is slightly inconvenient; we rephrase it in terms of a true coequaliser by freely adding a terminal object.

Definition 34. Given a category $\mathscr{B}$, let $\mathscr{B}^*$ be $\mathscr{B}$ extended freely with a terminal object.

Notation 35. We denote by $\bot$ the freely added terminal object in $\mathscr{B}^*$. Recall that $!$ denotes any terminal morphism.

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

Lemma 36. *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 $\mathscr{B}^*$.*

The following result is also useful.

Lemma 37. *Given a category $\mathscr{B}$, the canonical embedding functor $\mathscr{B} \to \mathscr{B}^*$ creates colimits.*

As a consequence,

1) whenever the colimit in $Kl_{\mathscr{D}+}^*$ is not $\bot$, it is also a colimit in $Kl_{\mathscr{D}+}$;
2) existing colimits in $Kl_{\mathscr{D}+}$ are also colimits in $Kl_{\mathscr{D}+}^*$;
3) in particular, coproducts in $Kl_{\mathscr{D}+}$ (which are computed in $\mathscr{C}$) are also coproducts in $Kl_{\mathscr{D}+}^*$.

The main point of pattern unification is the following result.

Theorem 38. *$Kl_{\mathscr{D}+}^*$ has coequalisers.*

In the next sections, we show how the generic unification algorithm summarised in Figure **??** provides a construction of such coequalisers.

IV. UNIFICATION PHASE

In this section, we describe the main unification phase, which computes a coequaliser in $Kl_{\mathscr{D}+}^*$. We denote a coequaliser $\coprod_i Ka_i \underset{\vec{u}}{\overset{\vec{t}}{\rightrightarrows}} \Gamma \xrightarrow{\sigma} \Delta$ in $Kl_{\mathscr{D}+}^*$ by $\Gamma \vdash \vec{t} = \vec{u} \Rightarrow \sigma \dashv \Delta$, following Notation 11.

Let us start with the structural rules. When $\Gamma = \bot$, the coequaliser is the terminal cocone, i.e., $\bot \vdash \vec{t} = \vec{u} \Rightarrow ! \dashv \bot$ holds. When the coproduct is empty, the coequaliser is just Γ, i.e., $\Gamma \vdash () = () \Rightarrow 1_\Gamma \dashv \Gamma$ holds.

Thanks to the U-SPLIT rule which constructs coequalisers sequentially (as discussed in Section §II-A), we can now focus on the rules dealing with singleton lists, that is, with coequaliser diagrams $Ka \underset{u}{\overset{t}{\rightrightarrows}} T\Gamma$. By Property 32, $t, u : Ka \to T\Gamma$ are either rigid or flexible. In the next subsections, we discuss all the different mutually exclusive situations (up to symmetry):

- both t or u are rigid (Section §IV-A),
- $t = M(\dots)$ and M does not occur in u (Section §IV-B),
- t and u are $M(\dots)$ (Section §IV-C),
- $t = M(\dots)$ and M occurs deeply in u (Section §IV-D).

A. Rigid-rigid

Here we detail unification of $o(\vec{t})$ and $o'(\vec{u})$ for some $o, o' : Ka \to O$, morphisms $\vec{t} : \overline{o} \to T\Gamma$, $\vec{u} : \overline{o'} \to T\Gamma$.

Assume given a unifier $\sigma : \Gamma \to \Delta$. By Property 33, $o(\vec{t}[\sigma]) = o'(\vec{u}[\sigma])$. By Property 32, this implies that $o = o'$, $\vec{t}[\sigma] = \vec{u}[\sigma]$. Therefore, we get the following failing rule

$$\frac{o \neq o'}{\Gamma \vdash o(\vec{t}) = o'(\vec{u}) \Rightarrow ! \dashv \bot}$$

We now assume $o = o'$. Then, $\sigma : \Gamma \to \Delta$ is a unifier if and only if it unifies $\vec{t}$ and $\vec{u}$. This induces an isomorphism between

the category of unifiers for $o(\vec{t})$ and $o(\vec{u})$ and the category of unifiers for $\vec{t}$ and $\vec{u}$, justifying the rule U-RigRig.

B. Flex-, no cycle*

Here we detail unification of $M(x)$, which is nothing but $\mathcal{L}x[in_M]$, and $u : Ka \to T(\Gamma, M : b)$, such that M does not occur in u, in the sense that $u = u'[in_\Gamma]$ for some $u' : Ka \to T\Gamma$. We exploit the following general lemma, recalling Notation 13.

Lemma 39 ([3], Exercise 2.17.1)**.** *In any category, denoting morphism composition $g \circ f$ by $f[g]$, the following rule applies:*

$$\frac{\Gamma \vdash t :> t' \Rightarrow v; \sigma \dashv \Delta}{\Gamma + B \vdash t[in_1] = t'[in_2] \Rightarrow \sigma, v \dashv \Delta}$$

Taking $t = M(x) = \mathcal{L}x : Ka \to (M : b)$ and $t' = u'$, we thus have the rule

$$\frac{\Gamma \vdash u' :> M(x) \Rightarrow v; \sigma \dashv \Delta \qquad u = u'[in_\Gamma]}{\Gamma, M : b \vdash M(x) = u \Rightarrow \sigma, M \mapsto v \dashv \Delta} \quad (3)$$

Let us make the factorisation assumption about u more effective. We can define by recursion a partial morphism from $T(\Gamma, M : b)$ to $T\Gamma$ that tries to compute u' from an input data u.

Lemma 40. *There exists $m_{\Gamma;b} : T(\Gamma, M : b) \to T\Gamma + 1$ such that a morphism $u : Ka \to T(\Gamma, M : b)$ factors as $Ka \xrightarrow{u'} T\Gamma \hookrightarrow T(\Gamma, M : b)$ if and only if $m_{\Gamma;b} \circ u = in_1 \circ u'$.*

Proof. We construct m by *recursion*, by equipping $T\Gamma + 1$ with an adequate F-algebra. Considering the embedding $(\Gamma, M : b) \xrightarrow{\eta + !} T\Gamma + 1$, we then get the desired morphism by universal property of $T(\Gamma, M : b)$ as a free F-algebra. The claimed property is proven by induction. $\square$

Therefore, we can rephrase (3) as the rule U-NoCycle in Figure **??**, using the following notations (in the second one, we take Γ as the empty context).

Notation 41. Given $u : Ka \to T(\Gamma, M : b)$, we denote $m_{\Gamma;b} \circ u$ by $u_{|\Gamma}$. Moreover, for any $u' : Ka \to T\Gamma$, we denote $in_1 \circ u' : Ka \to T\Gamma + 1$ by $\underline{u'}$.

Notation 42. Let Γ and Δ be metavariable contexts and $a \in \mathscr{D}$. Any $t : Ka \to T(\Gamma + \Delta)$ induces a Kleisli morphism $(\Gamma, M : a) \to T(\Gamma + \Delta)$ which we denote by $M \mapsto t$.

C. Flex-Flex, same metavariable

Here we detail unification of $M(x) = \mathcal{L}x[in_M]$ and $M(y) = \mathcal{L}y[in_M]$, with $x, y \in \hom_\mathscr{D}(a, b)$. We exploit the following lemma with $u = \mathcal{L}x$ and $v = \mathcal{L}y$.

Lemma 43. *In any category, denoting morphism composition $g \circ f$ by $f[g]$, the following rule applies:*

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

It follows that it is enough to compute the coequaliser of $\mathcal{L}x$ and $\mathcal{L}y$. Furthermore, by Property 21.(i) and Property 28, it can be computed as the image of the coequaliser of x and y, thus justifying the rule U-FlexFlex, using Notation 42.

D. Flex-rigid, cyclic

Here we handle unification of $M(x)$ for some $x \in \hom_\mathscr{D}(a, b)$ and $u : Ka \to \Gamma, M : b$, such that u is rigid and M occurs in u, i.e., $\Gamma \to \Gamma, M : b$ does not factor u. In Section §VI, we show that in this situation, there is no unifier. Then, Lemma 40 and Notation 41 justify the rule U-Cyclic.

V. Non-cyclic phase

The non-cyclic phase computes a pushout in $Kl^*_{\mathscr{D}+}$ of a span $\Gamma \xleftarrow{\vec{t}} \coprod_i Ka_i \xrightarrow{\coprod_i \mathcal{L}x_i} \coprod_i Kb_i$. We always implicitly assume (and enforce) that the right branch is a finite coproduct of free morphisms.

Remark 44. A pushout cocone for the above span consists in morphisms $\Gamma \xrightarrow{\sigma} T\Delta \xleftarrow{\vec{u}} \coprod_i Kb_i$ such that $\vec{t}[\sigma] = \vec{u} \circ \coprod_i Kx_i$, i.e., $t_i[\sigma] = \vec{x_i} \cdot u_i$ for each i.

When $\Gamma \xrightarrow{\sigma} \Delta \xleftarrow{\vec{u}} \coprod_i Kb_i$ is a pushout of the above span, we use Notation 13 and denote such a situation by $\Gamma \vdash \vec{t} :> \coprod_i \mathcal{L}x_i \Rightarrow \vec{u}; \sigma \dashv \Delta$.

Let us start with the structural rules. When $\Gamma = \bot$, the pushout is the terminal cocone, i.e., $\bot \vdash \vec{t} :> \vec{f} \Rightarrow !; ! \dashv \bot$ holds. When the coproduct is empty, the pushout is just Γ, i.e., $\Gamma \vdash () :> () \Rightarrow (); 1_\Gamma \dashv \Gamma$ holds. Finally, let us note that the sequential rule P-Split is in fact valid in any category.

Lemma 45. *In any category, denoting morphism composition $f \circ g$ by $g[f]$, the following rule applies.*

$$\frac{\Gamma \vdash t_1 :> f_1 \Rightarrow u_1; \sigma_1 \dashv \Delta_1 \quad \Delta_1 \vdash t_2[\sigma_1] :> f_2 \Rightarrow u_2; \sigma_2 \dashv \Delta_2}{\Gamma \vdash t_1, t_2 :> f_1 + f_2 \Rightarrow u_1[\sigma_2], u_2; \sigma_1[\sigma_2] \dashv \Delta_2}\text{P-Split}$$

Thanks to the above rule, we can now focus on the case where $\vec{t}$ is a singleton list, thus dealing with a span $T\Gamma \xleftarrow{t} Ka \xrightarrow{N(x)} T(N : b)$. By Property 32, the left morphism $Ka \to T\Gamma$ is either flexible or rigid. Each case is handled separately in the following subsections.

A. Rigid

Here, we describe the construction of a pushout of $\Gamma \xleftarrow{o(\vec{t})} Ka \xrightarrow{N(x)} N : b$ where $o : Ka \to O$ and $\vec{t} : \bar{o} \to T\Gamma$. By Remark 44, a cocone is a cospan $T\Gamma \xrightarrow{\sigma} T\Delta \xleftarrow{t'} Kb$ such that $o(\vec{t})[\sigma] = x \cdot t'$. By Property 33, this means that $o(\vec{t}[\sigma]) = x \cdot t'$. By Property 32, t' is either some $M(y)$ or $o'(\vec{u})$. But in the first case, $x \cdot t' = x \cdot M(y) = M(y \circ x)$ by Property 33, so it cannot equal $o(\vec{t}[\sigma])$, by Property 32. Therefore, $t' = o'(\vec{u})$ for some $o' : Kb \to O$ and $\vec{u} : \bar{o'} \to T\Delta$. By Property 33, $x \cdot t' = (x \cdot o')(\vec{u} \circ x^o)$. By Property 32, $o = x \cdot o'$, and $\vec{t}[\sigma] = \vec{u} \circ x^o$.

Note that there is at most one o' such that $o = x \cdot o'$, by Property 21.(iii). In this case, it follows from the above observations that a cocone is equivalently given by a cospan $T\Gamma \xrightarrow{\sigma} T\Delta \xleftarrow{\vec{u}} b^o$ such that $\vec{t}[\sigma] = \vec{u} \circ x^o$. But, by Remark 44, this is precisely the data for a pushout cocone for $\Gamma \xleftarrow{\vec{t}} a^o \xrightarrow{\mathcal{L}^+ x^o} b^o$. This actually induces an isomorphism

between the two categories of cocones, thus justifying the rules P-RIG and P-FAIL.

B. Flex

Here, we construct the pushout of $(\Gamma, M : c) \xleftarrow{M(y)} Ka \xrightarrow{N(x)} N : b$. Note that in this span, $N(x) = \mathcal{L}x$ while $M(y) = \mathcal{L}y[in_M]$. Thanks to the following lemma, it is actually enough to compute the pushout of $\mathcal{L}x$ and $\mathcal{L}y$.

Lemma 46. *In any category, denoting morphism composition by $f \circ g = g[f]$, the following rule applies*

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

By Property 21.(i) and Property 28, the pushout of $\mathcal{L}x$ and $\mathcal{L}y$ is the image by $\mathcal{L}$ of the pushout of x and y, thus justifying the rule P-FLEX.

VI. OCCUR-CHECK

The occur-check allows to jump from the main unification phase (Section §IV) to the non-cyclic phase (Section §V), whenever the metavariable appearing at the top-level of the l.h.s does not occur in the r.h.s. This section is devoted to the proof that if there is a unifier of $M(\vec{x})$ and t, then either M does not occur in t, or it occurs at top-level (see Corollary 51). The argument formalises 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, formally justified by exploiting the universal property of TX as the free F-algebra on X.

Definition 47. The size $|t| \in \mathbb{N}$ of a morphism $t : Ka \to T\Gamma$ is recursively defined by $|M(x)| = 0$ and $|o(\vec{t})| = 1 + |\vec{t}|$, with $|\vec{t}| = \sum_i t_i$, for any $\vec{t} : \coprod_i Ka_i \xrightarrow{\ldots, t_i, \ldots} T\Gamma$.

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

Definition 48. For each morphism $t : Ka \to T(\Gamma, M : b)$ 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 49. *For any $t : Ka \to T(\Gamma, M : b)$, if $|t|_M = 0$, then $T\Gamma \hookrightarrow T(\Gamma, M : b)$ factors t. Moreover, for any $\Gamma = (M_1 : a_1, \ldots, M_n : a_n)$, $t : Ka \to T\Gamma$, and $\sigma : \Gamma \to T\Delta$, we have $|t[\sigma]| = |t| + \sum_i |t|_{M_i} \times |\sigma_i|$.*

Corollary 50. *For any $t : Ka \to T(\Gamma, M : b)$, $\sigma : \Gamma \to T\Delta$, $x \in \hom_{\mathscr{D}}(a, b)$, $u : Kb \to T\Delta$, we have $|t[\sigma, u]| \geq |t| + |u| \times |t|_M$ and $|M(x)[u]| = |u|$.*

Corollary 51. *If there is a commuting square in Kl_T*

$$\begin{array}{ccc} Ka & \xrightarrow{t} & \Gamma, M : b \\ {\scriptstyle M(x)}\downarrow & & \downarrow{\scriptstyle \sigma, u} \\ M : b & \xrightarrow{u} & \Delta \end{array}$$

then either $t = M(y)$ for some y, or $T\Gamma \hookrightarrow T(\Gamma, M : b)$ factors t.

Proof. Since $t[\sigma, u] = M(x)[u]$, we have $|t[\sigma, u]| = |M(x)[u]|$. Corollary 50 implies $|u| \geq |t| + |u| \times |t|_M$. Therefore, either $|t|_M = 0$ and we conclude by Lemma 49, or $|t|_M = 1$ and $|t| = 0$ and so t is $M(y)$ for some y. $\square$

VII. COMPLETENESS

Each inductive rule presented so far provides an elementary step for the construction of coequalisers. We need to ensure that this set of rules allows to construct a coequaliser in a finite number of steps. To make the argument more straightforward, we explicitly assume that in the splitting rules U-SPLIT and P-SPLIT in figure **??**, the expressions with vector notation are not empty lists.

The following two properties are sufficient to ensure that applying rules eagerly eventually leads to a coequaliser: *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. 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 context Γ (as an element of $\mathscr{D}^+$), which can be defined as its size as a finite family of elements of $\mathcal{A}$ (see Remark 20). We extend this definition to the case where $\Gamma = \bot$, by taking $|\bot| = 0$. We also define the size $||t||$ of a *term* $t : Ka \to T\Gamma$ as in Definition 47 except that we assign a size of 1 to metavariables, so 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 $\Gamma \vdash f :> g \Rightarrow u; \sigma \dashv \Delta$ as $||f||$. 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 52. *For any $t : Ka \to T\Gamma$ and $\sigma : \Gamma \to T\Delta$, if σ is a renaming, i.e., $\sigma = \mathcal{L}^+\sigma'$, for some σ', then $||t[\sigma]|| = ||t||$.*

Lemma 53. *If there is a finite derivation tree of $\Gamma \vdash f :> g \Rightarrow u; \sigma \dashv \Delta$ and $\Delta \neq \bot$, then $|\Gamma| = |\Delta|$ and σ is a 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.

Lemma 54. *If there is a finite derivation tree of $\Gamma \vdash t = u \Rightarrow \sigma \dashv \Delta$, then $|\Gamma| \geq |\Delta|$, and moreover if $|\Gamma| = |\Delta|$ and $\Delta \neq \bot$, then σ is a renaming.*

VIII. APPLICATIONS

In the examples, we motivate the definition of the category $\mathcal{A}$ based on what we expect from metavariable arities, following Remark 17.

In all our examples, O is a coproduct $\coprod_{\ell \in V} O_\ell$ for some $O_\ell : \mathcal{A} \to \mathrm{Set}$ and $J_{a,in_\ell(o)} = \gamma_\ell$ for some finite cardinal γ_ℓ. In this case, L is equivalently given by a family of functors $H_{\ell,j} : \int O_\ell \to \mathrm{Set}$ for each $j \in \gamma_\ell$ by $L_{a,in_\ell(o),j} = H_{\ell,j}(a,o)$. Then,

$$F(X)_a \cong \coprod_{\ell \in V} \coprod_{o \in O_\ell} \prod_{j \in \gamma_\ell} X_{H_{\ell,j}(a,o)}$$

A. Simply-typed second-order syntax

In this section, we present the example of simply-typed λ-calculus. Our treatment generalises to any second-order binding signature (see [7]).

Let T denote the set of simple types generated by a set of simple types. A metavariable arity $\tau_1, \ldots, \tau_n \vdash \tau_f$ is given by a list of input types $\tau_1, \ldots, \tau_n$ and an output type τ_f. Substituting a metavariable $M : (\Gamma \vdash \tau)$ with another $M' : (\Gamma' \vdash \tau')$ requires that $\tau = \tau'$ and involves an injective renaming $\Gamma \to \Gamma'$. Thus, we consider $\mathcal{A} = \mathbb{F}_m[T] \times T$, 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 $|\Gamma|_\tau$ denotes the number (as a cardinal set) of occurrences of τ in Γ.

B. 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. Again, we can define the endofunctor for λ-calculus as in Section §??. Then, a metavariable takes as argument a set of available 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.

C. Linear ordered λ-calculus

Properties of Terms in Continuation-Passing Style in an Ordered Logical Framework

We take the example of linear λ-calculus. A metavariable arity is then a natural number specifying the number of arguments, and because of linearity, a metavariable can only be substituted with another one with the same arity. We therefore $\mathcal{A} = \mathcal{P}$ to be category whose objects are finite cardinals and morphisms are bijections as in [21].

Each typing rule provides an operation

Variable	$1 \vdash *$	$y1$
Application	$\dfrac{n \vdash t \quad m \vdash u}{n + m \vdash t\, u}$	$X_n \times X_m \times y_{n+m}$

D. Intrinsic polymorphic syntax

We present intrinsic system F, following [13]. 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 $\{0, \ldots, n-1\}$. Intuitively, a metavariable arity $n; \sigma_1, \ldots, \sigma_p \vdash \tau$ specifies the number n of free type variables, the list of input types $\vec{\sigma}$, and the output type τ, all living in S_n. Substituting a metavariable $M : (n; \vec{\sigma} \vdash \tau)$ with another $M' : (n'; \vec{\sigma}' \vdash \tau')$ requires a choice $(\alpha_0, \ldots, \alpha_{n-1})$ of n distinct type variables among $\{0, \ldots n'-1\}$, such that $\tau[\vec{\alpha}] = \tau'$, and an injective renaming $\vec{\sigma}'[\vec{\alpha}] \to \vec{\sigma}$. We therefore consider the category $\mathcal{A}$ of metavariable arities where a morphism between $n; \Gamma \vdash \tau$ and $n'; \Gamma' \vdash \tau'$ is a morphism $\sigma : n \to n'$ in $\mathbb{F}_m$ such that $\tau[\sigma] = \tau'$, and a renaming $\Gamma[\sigma] \to \Gamma'$. More formally, $\mathcal{A}$ is the op-lax colimit of $n \mapsto \mathbb{F}_m[S_n] \times S_n$. The intrinsic syntax of system F can then be specified as in Table I.

To understand how unification of two metavariables works (see the rules U-RIGRIG and P-RIG), let us explain how finite connected limits are computed in $\mathcal{A}$.

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. Note that this is op-lax colimit of $n \mapsto \mathbb{F}_m[S_n]$, and there is an obvious projection $\mathcal{A} \to \mathcal{A}'$, which creates finite limits, as we will show.

Lemma 55. $\mathcal{A}'$ *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 $\Gamma \in \mathbb{F}_m[S_n]$ and $f : n' \to n$ is a morphism in $\mathbb{F}_m$, then $f_! \Gamma \in \mathbb{F}_m[S_{n'}]$ is essentially Γ restricted to elements of S_n that are in the image of S_f. Note that $f_!$ is right adjoint to $\Gamma \mapsto \Gamma[f]$, and is thus continuous. We now apply [12, Theorem 4.2 and Proposition 4.1]: each $\mathbb{F}_m[S_n]$ has those limits. $\square$

Lemma 56. *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 $n_i; \Gamma_i \vdash \tau_i$. Let $n; \Gamma$ be the limit of $i \mapsto n_i; \Gamma_i$ 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 $n; \Gamma \vdash \tau$ is the limit of d. $\square$

More concretely, a finite connected limit of $i \mapsto n_i; \Gamma_i \vdash \tau_i$ in $\mathcal{A}$ is computed as follows:

1) compute the limit n of $(n_i)_i$ in $\mathbb{F}_m$, denoting $p_i : n \to n_i$ is the canonical projections;
2) define τ as the (only) element of S_n such that $\tau[p_i] = \tau_i$
3) define Γ as the limit in $\mathbb{F}_m[S_n]$ of $p_{i,!} \Gamma_i$, where $p_{i,!} : \mathbb{F}_m[S_{n_i}] \to \mathbb{F}_m[S_n]$ is the reindexing functor described in the proof of Lemma 55.

TABLE I
EXAMPLES OF ENDOFUNCTORS FOR SYNTAX

$$F(X)_a \cong \coprod_{\ell \in V} \coprod_{o \in O_\ell} \prod_{j \in \gamma_\ell} X_{H_{\ell,j}(a,o)}$$

Simply-typed λ-calculus (Section §VIII-A)

Typing rule	$O_\ell(\Gamma \vdash \tau)$	$H_{\ell,j}(\Gamma \vdash \tau)$
$\dfrac{x:\tau \in \Gamma}{\Gamma \vdash x:\tau}$	$\{v_i \mid i \in |\Gamma|_\tau\}$	-
$\dfrac{\Gamma \vdash t:\tau' \Rightarrow \tau \quad \Gamma \vdash u:\tau'}{\Gamma \vdash t\,u:\tau}$	$\{a_{\tau'} \mid \tau' \in T\}$	$H_{-,0} = \Gamma \vdash \tau' \Rightarrow \tau$ $H_{-,1} = \Gamma \vdash \tau'$
$\dfrac{\Gamma, x:\tau_1 \vdash t:\tau_2}{\Gamma \vdash \lambda x.t:\tau_1 \Rightarrow \tau_2}$	$\{l_{\tau_1,\tau_2} \mid \tau = (\tau_1 \Rightarrow \tau_2)\}$	$H_{-,0} = \Gamma, \tau_1 \vdash \tau_2$

System F (Section §VIII-D)

Typing rule	$O_\ell(n;\Gamma \vdash \tau)$	$H_{\ell,j}(n;\Gamma \vdash \tau, o)$
$\dfrac{x:\tau \in \Gamma}{n;\Gamma \vdash x:\tau}$	$\{v_i \mid i \in |\Gamma|_\tau\}$	-
$\dfrac{n;\Gamma \vdash t:\tau' \Rightarrow \tau \quad n;\Gamma \vdash u:\tau'}{n;\Gamma \vdash t\,u:\tau}$	$\{a_{\tau'} \mid \tau' \in S_n\}$	$H_{-,0} = n;\Gamma \vdash \tau' \Rightarrow \tau$ $H_{-,1} = n;\Gamma \vdash \tau'$
$\dfrac{n;\Gamma, x:\tau_1 \vdash t:\tau_2}{n;\Gamma \vdash \lambda x.t:\tau_1 \Rightarrow \tau_2}$	$\{l_{\tau_1,\tau_2} \mid \tau = (\tau_1 \Rightarrow \tau_2)\}$	$H_{-,0} = n;\Gamma, \tau_1 \vdash \tau_2$
$\dfrac{n;\Gamma \vdash t:\forall\tau_1 \quad \tau_2 \in S_n}{n;\Gamma \vdash t \cdot \tau_2 : \tau_1[\tau_2]}$	$\{A_{\tau_1,\tau_2} \mid \tau = \tau_1[\tau_2]\}$	$H_{-,0} = n;\Gamma \vdash \forall\tau_1$
$\dfrac{n+1;wk(\Gamma) \vdash t:\tau}{n;\Gamma \vdash \Lambda t:\forall\tau}$	$\{\Lambda_{\tau'} \mid \tau = \forall\tau'\}$	$H_{-,0} = n+1;wk(\Gamma) \vdash \tau'$

This means that to unify $M(\vec{\alpha};\vec{\tau})$ with $M(\vec{\alpha}';\vec{\tau}')$, with M of arity $p;\vec{u} \vdash u'$, we first need to compute the vector of common position $\vec{i}$ between $\vec{\alpha}$ and $\vec{\alpha}'$, i.e, the largest vector $(i_1 < \cdots < i_n)$ such that $\alpha_{i_j} = \alpha'_{i_j}$. Then, we consider $\vec{\sigma}$ and $\vec{\sigma}'$ so that $\vec{\sigma}[j \mapsto i_{j+1}]$ is the sub-list of $\vec{\tau}$ that only use type variables in $\alpha_{i_1}, \ldots, \alpha_{i_n}$, and similarly for $\vec{\sigma}'$ and $\vec{\tau}'$. Finally, we define $(t_1, \ldots, t_m)$ as the vector of common positions between $\vec{\sigma}$ and $\vec{\sigma}'$. The most general unifier is then $M \mapsto N(\vec{i}, \vec{t})$ for a fresh metavariable N of arity $n; \sigma_{t_1}, \ldots \sigma_{t_m} \vdash t'$, where t' is $u'[i_{j+1} \mapsto j]$.

REFERENCES

[1] 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

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

[3] Borceux, F.: Handbook of Categorical Algebra 1: Basic Category Theory. Encyclopedia of Mathematics and its Applications, Cambridge University Press (1994). https://doi.org/10.1017/CBO9780511525858

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

[5] 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)

[6] 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

[7] 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)

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

[9] 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)

[10] 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)

[11] 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

[12] 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)

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

[14] 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

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

[16] 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

[17] 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)

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

[19] 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

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

[21] Tanaka, M.: Abstract syntax and variable binding for linear binders. In: MFCS '00. LNCS, vol. 1893. Springer (2000)

[22] 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)

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

[24] 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
PROOF OF PROPERTY 21

(i) We assume that $\mathcal{A}$ has finite connected limits. Hence, its opposite category $\mathcal{D} = \mathcal{A}^{op}$ has finite connected colimits.

(ii) Let $y : \mathcal{A}^{op} \to [\mathcal{A}, \mathrm{Set}]$ denote the Yoneda embedding and $J : \mathcal{C} \to [\mathcal{A}, \mathrm{Set}]$ denote the canonical embedding, so that

$$y = J \circ K. \tag{4}$$

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

$$\mathcal{C}(K \lim F, X) \cong [\mathcal{A}, \mathrm{Set}](JK \lim F, JX)$$
$$(J \text{ is fully faithful})$$
$$\cong [\mathcal{A}, \mathrm{Set}](y \lim F, JX) \quad (\text{By (4)})$$
$$\cong JX(\lim F) \quad (\text{By the Yoneda Lemma.})$$
$$\cong \lim(JX \circ F)$$
$$(X \text{ preserves finite connected limits})$$
$$\cong \lim([\mathcal{A}, \mathrm{Set}](yF-, JX)]$$
$$(\text{By the Yoneda Lemma})$$
$$\cong \lim([\mathcal{A}, \mathrm{Set}](JKF-, JX)] \quad (\text{By (4)})$$
$$\cong \lim \mathcal{C}(KF-, X) \quad (J \text{ is full and faithful})$$
$$\cong \mathcal{C}(\mathrm{colim}\ KF, X)$$
$$(\text{By left continuity of the hom-set bifunctor})$$

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

(iii) A morphism $f : a \to b$ is epimorphic if and only if the following square is a pushout [15, Exercise III.4.4]

$$\begin{array}{ccc} a & \xrightarrow{f} & b \\ f\downarrow & & \| \\ b & = & b \end{array}$$

We conclude by *(ii)*, because all morphisms in $\mathcal{D} = \mathcal{A}^{op}$ are epimorphic by assumption.

(iv) This follows from Lemma 18, because a morphism $f : A \to B$ is monomorphic if and only if the following square is a pullback

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

(v) This follows from coproducts being computed pointwise (Lemma 18), and representable functors being connected, by the Yoneda Lemma.