Generic pattern unification: a categorical approach

Ambroise Lafont[0000−0002−9299−641X] and Neel Krishnaswami[0000−0003−2838−5865]

University of Cambridge

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, linear syntax, or (intrinsic) polymorphic syntax such as system F.

Keywords: Unification · Category theory.

1 Introduction

Unification consists in finding a *unifier* of two terms t, u, that is a (metavariable) substitution σ such that $t[\sigma] = u[\sigma]$. Unification algorithms try to compute a most general unifier σ, in the sense that given any other unifier δ, there exists a unique δ' such that $\delta = \sigma[\delta']$.

First-order unification [11] 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 [7], which is undecidable [6]. However, Miller [9] 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.

First-order unification has been explained from a lattice-theoretic point of view by Plotkin [2], and later categorically analysed in [12,5]. However, there is little work on understanding pattern unification algebraically, with the notable exception of [13], working with normalised terms of simply-typed λ-calculus. The present paper[1] can be thought of as a generalisation of their work.

Plan of the paper

In Section §2, we present our categorical setting. In Section §3, 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 1, starting

[1] An expanded version of this work can be found in [8].

with the unification phase (Section §4), the pruning phase (Section §5), the occur-check (Section §6). We finally justify completeness in Section §7.

As an introduction, we start by presenting pattern unification in the case of pure λ-calculus in Section §1.1. In Section §1.2, we then present the generic algorithm summarised in Figure 1, instantiated for a syntax specified by a *binding signature*. Finally, in Section §1.3, we motivate our general setting and provide categorical semantics of the algorithm, by revisiting pure λ-calculus.

1.1 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 [3]. More formally, the syntax is inductively generated by the following inductive rules, where C is a variable context $(0 : \tau, 1 : \tau, \dots, n : \tau)$, often abbreviated as n, and τ denotes the sort of terms, which we often omit, while Γ is a metavariable context $M_1 : n_1, \dots, M_m : n_m$ specifying a metavariable symbol M_i together with its number of arguments n_i.

$$\frac{x \in C}{\Gamma; C \vdash x} \qquad \frac{\Gamma; C \vdash t \quad \Gamma; C \vdash u}{\Gamma; C \vdash t\ u} \qquad \frac{\Gamma; C, |C| : \tau \vdash t}{\Gamma; C \vdash \lambda t}$$

$$\frac{M : n \in \Gamma \quad x_1, \dots, x_n \in C \quad x_1, \dots x_n \text{ distinct}}{\Gamma; C \vdash M(x_1, \dots, x_n)}$$

Note that the De Bruijn level convention means that the variable bound in $\Gamma; C \vdash \lambda t$ is $|C|$, the length of the variable context C.

A *metavariable substitution* $\sigma : \Gamma \to \Gamma'$ assigns to each declaration $M : n$ in Γ a term $\Gamma'; n \vdash \sigma_M$. This assignation extends (through a recursive definition) to any term $\Gamma; C \vdash t$, yielding a term $\Gamma'; C \vdash t[\sigma]$. The base case is $M(x_1, \dots, x_n)[\sigma] = \sigma_M[i \mapsto x_{i+1}]$, where $-[i \mapsto x_{i+1}]$ is variable renaming. 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 $\Gamma; C \vdash t, u$ 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 C 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[2] 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 $\vec{t} = (t_1, \dots, t_n)$ and $\vec{u} = (u_1, \dots, u_n)$ such that $\Gamma; C_i \vdash t_i, u_i$.

[2] This trick will be justified from a categorical point of view in Section §3.

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 following *congruence* rule for application.

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

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 U-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 appears deeply in u
2. M appears in u at top level, i.e., $u = M(y_1, \ldots, y_n)$;
3. M does not appear 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

$$\frac{u \neq M(\ldots) \qquad u_{|\Gamma} \neq \underline{\ldots}}{\Gamma, M : n \vdash M(\vec{x}) = u \Rightarrow ! \dashv \bot}$$

where $u_{|\Gamma} \neq \underline{\ldots}$ means that u does not restrict to the smaller metavariable context Γ, and thus that M does appear in u.

In the second case, we are unifying $M(\vec{x})$ with $M(\vec{y})$. The most general unifier substitutes M with $M'(z_1, \ldots, z_p)$, where $z_1, \ldots, z_p$ is the family of common positions i such that $x_i = y_i$. We denote[3] such a situation by $n \vdash \vec{x} = \vec{y} \Rightarrow \vec{z} \dashv p$. We therefore get the rule

$$\frac{n \vdash \vec{x} = \vec{y} \Rightarrow \vec{z} \dashv p}{\Gamma, M : n \vdash M(\vec{x}) = M(\vec{y}) \Rightarrow M \mapsto M'(\vec{z}) \dashv \Gamma, M' : p} \tag{1}$$

The last case is unification of $M(\vec{x})$ with some u such that M does not appear 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. In this case, the algorithm enters a *pruning phase*. To give an example, when unifying an application $t\ u$ with a metavariable application $M(x_1, \ldots, x_n)$ in this situation, we create two fresh metavariables M_1 and M_2, unify t with $M_1(x_1, \ldots, x_n)$ and u with $M_2(x_1, \ldots, x_n)$. Eventually, M is going to be replaced with $(M_1(\vec{x})\ M_2(\vec{x}))[\sigma]$, where σ is the output unifying

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

substitution. We call it the pruning phase because it effectively removes all *outbound* variables in u', i.e., those that are not among the arguments $x_1, \ldots, x_n$ of the metavariable, by producing a substitution that restricts the arities of the metavariables occurring in u'.

Let us introduce a specific notation for this phase: $\Gamma \vdash u' :> M(\vec{x}) \Rightarrow v; \sigma \dashv \Delta$ means that σ is the output pruning substitution, and v is essentially $u'[\sigma][x_{i+1} \mapsto i]$, the term that the metavariable M ought to be substituted with. Note that the metavariable symbol M is fresh in this notation: it appears neither in Γ nor u', and is not in the domain of σ.

The output σ, v defines a substitution $(\Gamma, M : n) \to \Delta$ which can be characterised as the most general unifier of u' and $M(\vec{x})$. We thus have the rule

$$\frac{u_{|\Gamma} = \underline{u'} \qquad \Gamma \vdash u' :> M(\vec{x}) \Rightarrow v; \sigma \dashv \Delta}{\Gamma, M : n \vdash M(\vec{x}) = u \Rightarrow \sigma, M \mapsto v \dashv \Delta} \tag{2}$$

As before, we generalise the pruning phase to handle lists $\vec{u} = (u_1, \ldots, u_n)$ of terms such that $\Gamma; C_i \vdash u_i$, and lists of pruning patterns $(\vec{x}_1, \ldots, \vec{x}_n)$ where each $\vec{x_i}$ is a choice of distinct variables in C_i. Then, $\Gamma \vdash \vec{u} :> M_1(\vec{x}_1), \ldots, M_n(\vec{x}_n) \Rightarrow \vec{v}; \sigma \dashv \Delta$ means that each σ is the common pruning substitution, and v_i is essentially $u_i[\sigma][x_{i,j+1} \mapsto j]$. Again, $(\sigma, \vec{v})$ define a substitution from $\Gamma, M_1 : |\vec{x}_1|, \ldots, M_n : |\vec{x}_n|$ to Δ which can be characterised as the most general unifier of $\vec{u}$ and $M_1(\vec{x}_1), \ldots M_n(\vec{x}_n)$.

We can then handle application as follows.

$$\frac{\Gamma \vdash t, u :> M_1(\vec{x}), M_2(\vec{x}) \Rightarrow v_1, v_2; \sigma \dashv \Delta}{\Gamma \vdash t\ u :> M(\vec{x}) \Rightarrow v_1\ v_2; \sigma \dashv \Delta} \tag{3}$$

As for unification (rule U-Split), pruning can be done sequentially, as in the rule P-Split in Figure 1. The usage of $+$ as a separator will be formally justified later in Remark 2: it intuitively means 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.

$$\frac{y = x_{i+1}}{\Gamma \vdash y :> M(\vec{x}) \Rightarrow i; 1_\Gamma \dashv \Gamma} \qquad \frac{y \notin \vec{x}}{\Gamma \vdash y :> M(\vec{x}) \Rightarrow !; ! \dashv \bot} \tag{4}$$

In λ-abstraction, the bound variable $|C|$ need not be pruned: we extend the list of allowed variables accordingly.

$$\frac{\Gamma \vdash t :> M_1(\vec{x}, |C|) \Rightarrow v; \sigma \dashv \Delta}{\Gamma \vdash \lambda t :> M(\vec{x}) \Rightarrow \lambda v; \sigma \dashv \Delta} \tag{5}$$

The remaining case consists in pruning a metavariable $N(x_1, \ldots, x_m)$, whose arity must then be restricted to those positions in $y_1, \ldots, y_n$. To be more precise, consider the family $z_1, \ldots, z_p$ of common values in $x_1, \ldots, x_m$ and $y_1, \ldots, y_n$, 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 $\{0, \ldots, m-1\}$ and $\{0, \ldots, n-1\}$ respectively. We denote[4] such a situation by $m \vdash \vec{x} :> \vec{y} \Rightarrow \vec{l}; \vec{r} \dashv p$. Then, the metavariable N is substituted with $N'(\vec{r})$ for some new metavariable N' of arity p, while the metavariable M is replaced with $N'(\vec{l})$:

$$\frac{m \vdash \vec{x} :> \vec{y} \Rightarrow \vec{l}; \vec{r} \dashv p}{\Gamma, N : m \vdash N(\vec{x}) :> M(\vec{y}) \Rightarrow N'(\vec{l}); N \mapsto N'(\vec{r}) \dashv \Gamma, N' : p} \tag{6}$$

This ends our description of the unification algorithm, in the specific case of pure λ-calculus. The goal of this paper is to generalise it, by parameterising the algorithm by a signature specifying a syntax.

1.2 First generalisation: parameterisation by a binding signature

As a first step, let us parameterise the unification algorithm by a binding signature [10]. A syntax is then specified by a set of symbols O together with a list of natural numbers $\vec{\alpha}_o$ for each $o \in O$ specifying the number of arguments (the size of the list) and the number of bound variables in each argument. For example, pure λ-calculus is specified by $O = \{app, lam\}$ with $\vec{\alpha}_{app} = (0, 0)$ and $\vec{\alpha}_{lam} = (1)$. The unification algorithm described in the previous section straightforwardly generalises to any syntax specified by a binding signature. Figure 1 summarises the generic algorithm that we will later interpret in a more general setting. In the figure, the vector notation for the arguments of metavariables is dropped because in the general setting, they are abstract and could thus instantiate to other objects than lists[5]. We still use the vector notation in the following instantiation for a syntax specified by a binding signature.

In the rule U-RigRig, the expression $o(\vec{t})$ can be an operation or a variable, in which case $\vec{t}$ is the empty list. If o is an operation, the exact nature of $\vec{t}$ depends on the arity $\alpha_o = (n_1, \ldots, n_p)$ of o: then $\vec{t}$ is a list of terms of size p and $\Gamma; |C| + n_i \vdash t_i$ for each $i \in \{1, \ldots, p\}$, where C is the variable context of $o(\vec{t})$. Let us comment the rigid case in the pruning phase. Both rules P-Rig and P-Fail are concerned with pruning a non metavariable term $\Gamma; C \vdash o(\vec{t})$. In the variable case, these two rules instantiate to (4). More precisely, if o is a variable in C, the side condition $o = \vec{x} \cdot o'$ means that $o = x_{o'+1}$. On the other hand, if o is an operation $\vec{x} \cdot o$ is defined as o and thus the rule P-Rig always applies. If $\alpha_o = (n_1, \ldots, n_p)$, then the notation $\mathcal{L}^+ \vec{x}^o$ essentially unfolds to a list of the same size as α_o and whose i^{th} element is $M_i(\vec{x}, |C|, \ldots, |C| + n_i - 1)$. For example, for pure λ-calculus, $\mathcal{L}^+ \vec{x}^o = M_1(\vec{x}), M_2(\vec{x})$ in the application case, and $\mathcal{L}^+ \vec{x}^o = M_1(\vec{x}, |C|)$ in the abstraction case, thus recovering the rules (3) and (5).

Note that the premises of the rules U-FlexFlex and P-Flex are not explicitly defined in figure 1, although for a syntax specified by a binding signature,

[4] Again, the similarity with the pruning notation is no coincidence: as we will see, both are (co)pushouts.

[5] See [8, Section 8.2] for an example where arguments are sets.

they have the same meaning as in the previous section. This is because the generic algorithm works in a more general setting, as we are going to explain in the next section, so that they need to be customised for each specific situation.

1.3 Categorification

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 1.

Consider the category of functors $[\mathbb{F}_m, \mathrm{Set}]$ from $\mathbb{F}_m$, the category of finite cardinals and injections between them, to the category of sets. A functor $X : \mathbb{F}_m \to \mathrm{Set}$ can be thought of as assigning to each natural number n a set X_n of expressions with free variables taken in the set $\underline{n} = \{0, \ldots, n-1\}$. The action on morphisms of $\mathbb{F}_m$ means that these expressions support injective renamings. Pure λ-calculus defines such a functor Λ by $\Lambda_n = \{t \mid \cdot ; n \vdash t\}$. It satisfies the recursive equation $\Lambda_n \cong \underline{n} + \Lambda_n \times \Lambda_n + \Lambda_{n+1}$, where $- + -$ is disjoint union.

In pattern unification, we consider extensions of this syntax with metavariables taking a list of distinct variables as arguments. As an example, let us add a metavariable of arity p. The extended syntax Λ' defined by $\Lambda'_n = \{t \mid M : p; n \vdash t\}$ now satisfies the recursive equation $\Lambda'_n = \underline{n} + \Lambda'_n \times \Lambda'_n + \Lambda'_{n+1} + Inj(p, n)$, where $Inj(p, n)$ is the set of injections between the cardinal sets p and n, corresponding to a choice of arguments for the metavariable. In fact, $Inj(p, n)$ is nothing but the set of morphisms between p and n in the category $\mathbb{F}_m$, which we denote by $\mathbb{F}_m(p, n)$.

Obviously, the functors Λ and Λ' satisfy similar recursive equations. Denoting F the endofunctor on $[\mathbb{F}_m, \mathrm{Set}]$ mapping X to $I + X \times X + X_{-+1}$, where I is the functor mapping n to $\underline{n}$, the functor Λ can be characterised as the initial algebra for F, thus satisfying the recursive equation $\Lambda \cong F(\Lambda)$. In other words, Λ is the free F-algebra on the initial functor 0. On the other hand, Λ' is characterised as the initial algebra for $F(-) + yp$, where yp is the (representable) functor $\mathbb{F}_m(p, -) : \mathbb{F}_m \to \mathrm{Set}$, thus satisfying the recursive equation $\Lambda' \cong F(\Lambda') + yp$. In other words, Λ' is the free F-algebra on yp. Denoting T the free F-algebra monad, Λ is $T(0)$ and Λ' is $T(yp)$. Similarly, the functor would be $T(yp + yq)$ corresponds to extending the syntax with another metavariable of arity q.

In the view to abstracting pattern unification, these observations motivate considering functors categories $[\mathcal{A}, \mathrm{Set}]$, 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[6] 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.

To understand how a unification problem is stated in this general setting, let us come back to the example of pure λ-calculus. We first provide the familiar metavariable context notation with a formal meaning.

[6] In Section §2.2, we make explicit assumptions about this endofunctor for the unification algorithm to be properly generalised.

Notation 1. *We denote a finite coproduct* $\coprod_{i\in\{M,N,\dots\}} yn_i$ *of representable functors by a* (metavariable) context $M : n_M, N : n_N, \dots$

If $\Gamma = (M_1 : m_1, \dots, M_p : m_p)$ and Δ 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 λ-term $\Delta; m_i \vdash \sigma_i$ for each $i \in \{1, \dots, p\}$. Note that this is precisely the data for a metavariable substitution $\Delta \to \Gamma$. Thus, Kleisli morphisms are nothing but metavariable substitution. Moreover, Kleisli composition correspond to composition of substitutions.

A unification problem can be stated as a pair of parallel Kleisli morphisms $yp \underset{u}{\overset{t}{\rightrightarrows}} T\Gamma$ where Γ is a metavariable context, corresponding to selecting 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 coproducts of representable functors[7]. 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 §3.

With this in mind, we can provide the categorical semantics of the unification notation in Figure 1.

Notation 2. *We denote a coequaliser* $A \underset{u}{\overset{t}{\rightrightarrows}} \Gamma - \overset{\sigma}{\frown} \Delta$ *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 an error object $\bot$ (as formally justified in Section §3), 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 (1), whose premise precisely means that $p \xrightarrow{\vec{z}} n \underset{\vec{y}}{\overset{\vec{x}}{\rightrightarrows}} C$ is indeed an equaliser in $\mathbb{F}_m$.

Remark 1. In Notation 2, when A is a coproduct $yn_1 + \dots + yn_p$, 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 $a + c \xrightarrow{f,g} b$ given morphisms $a \xrightarrow{f} b \xleftarrow{g} c$.

Note that the rule U-Split is in fact valid in any category (see [12, Theorem 9]). To formally understand the rule U-NoCycle which we have already introduced in the case of pure λ-calculus, see (2), let us provide the pruning notation with categorical semantics.

[7] This generalises the first-order case [12, Chapter 8] in the sense that we consider a free monad on a presheaf category, rather than on sets.

Unification Phase

– Structural rules (Section §4)

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

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

– Rigid-rigid (Section §4.1)

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

– Flex-*, no cycle (Section §4.2)

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

– Flex-Flex, same (Section §4.3)

$$\frac{b \vdash x =_{\mathscr{D}} y \Rightarrow z \dashv c}{\Gamma, M : b \vdash M(x) = M(y) \Rightarrow M \mapsto M'(z) \dashv \Gamma, M' : c}\text{U-FlexFlex}$$

– Flex-Rigid, cyclic (Section §4.4)

$$\frac{u = o(\vec{t}) \qquad u_{|\Gamma} \neq \underline{\ldots}}{\Gamma, M : b \vdash M(x) = u \Rightarrow ! \dashv \bot} \quad + \text{symmetric rule}$$

Pruning phase

– Structural rules (Section §5)

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

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

– Rigid (Section §5.1)

$$\frac{\Gamma \vdash \vec{t} :> \mathcal{L}^+ x^o \Rightarrow \vec{u}; \sigma \dashv \Delta \qquad o = x \cdot o'}{\Gamma \vdash o(\vec{t}) :> N(x) \Rightarrow o'(\vec{u}); \sigma \dashv \Delta}\text{P-Rig} \qquad \frac{o \neq x \cdot \ldots}{\Gamma \vdash o(\vec{t}) :> N(x) \Rightarrow !; ! \dashv \bot}\text{P-Fail}$$

– Flex (Section §5.2)

$$\frac{c \vdash_{\mathscr{D}} y :> x \Rightarrow y'; x' \dashv d}{\Gamma, M : c \vdash M(y) :> N(x) \Rightarrow M'(y'); M \mapsto M'(x') \dashv \Gamma, M' : d}\text{P-Flex}$$

Figure 1. Summary of the rules

Notation 3. *We denote a pushout diagram in a category $\mathscr{B}$ as below left by the notation as below right, sometimes even omitting $\mathscr{B}$.*

$$\begin{array}{ccc} A & \xrightarrow{f} & \Gamma' \\ {\scriptstyle t}\downarrow & & \downarrow{\scriptstyle u} \\ \Gamma & \xrightarrow[\sigma]{} & \ulcorner\Delta \end{array} \quad \Leftrightarrow \quad \Gamma \vdash_{\mathscr{B}} t :> f \Rightarrow u; \sigma \dashv \Delta$$

Similarly to Notation 2, this is used in Figure 1, taking $\mathscr{B}$ to be the Kleisli category of T restricted to coproducts of representable functors, and extended with an error object $\bot$, 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 (6) whose premise precisely means that the following square is a pullback in $\mathbb{F}_m$.

$$\begin{array}{ccc} p & \xrightarrow{l} & n \\ {\scriptstyle r}\downarrow & & \downarrow{\scriptstyle x} \\ m & \xrightarrow[y]{} & C \end{array}$$

Again, the rule U-Split is valid in any category (see Lemma 10).

Remark 2. Let us add a few more comments about Notation 3. 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.

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 $g : TB \to TC$ by $f[g] : A \to TC$.

2 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 §2.1 and for the former in Section §2.2. Finally, in Section §2.3, we sketch some examples.

2.1 Base category

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

Example 1. In Section §1.2, 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 [4].

Remark 3. 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 precise argument we develop 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[8] 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 4. 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 $\hom_{\mathbb{F}_m}(n, m)$.

By the Yoneda Lemma, any representable functor is in $\mathscr{C}$ and thus the embedding $\mathscr{C} \to [\mathcal{A}, \mathrm{Set}]$ factors the Yoneda embedding $\mathcal{A}^{op} \to [\mathcal{A}, \mathrm{Set}]$. We set $\mathscr{D} = \mathcal{A}^{op}$ and denote the fully faithful embedding as $\mathscr{D} \xrightarrow{K} \mathscr{C}$. A useful lemma that we will exploit is the following:

Lemma 1. *$\mathscr{C}$ is closed under limits, coproducts, and filtered colimits. Moreover, it is cocomplete.*

Notation 4. *We denote by $\mathscr{D}^+ \xrightarrow{K^+} \mathscr{C}$ the full subcategory of $\mathscr{C}$ consisting of finite coproducts of objects of $\mathscr{D}$.*

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

Remark 5. $\mathscr{D}^+$ is equivalent to the category of finite families of objects of $\mathcal{A}$. Thinking of objects of $\mathcal{A}$ as metavariable arities (Remark 4), $\mathscr{D}^+$ can be thought of as the category of metavariable contexts.

We adapt Notation 1 for objects of $\mathscr{D}^+$, that is, a coproduct $\coprod_{i\in\{M,N,\dots\}} Ka_i$ is denoted by a *(metavariable) context* $M : a_M, N : a_N, \dots$

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

Property 1. 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.
(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 $A_j \to \coprod_i A_i$.

Remark 6. Continuing Remark 3, unification of two metavariables as pullbacks or equalisers in $\mathcal{A}$ crucially relies on Property 1.(ii), which holds because we restrict to functors preserving finite connected limits.

2.2 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 $\int O \xrightarrow{J} \mathbb{F} \hookrightarrow \mathrm{Set}$. Objects are triples (a, o, j), where a is an object of $\mathcal{A}$, $o \in O_a$, and $j \in \{0, \dots, J_{a,o} - 1\}$, 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 2. For pure λ-calculus where $\mathcal{A} = \mathbb{F}_m$, we have $O_n = \{a, l\} + \{v_i | 0 \le i < n\}$, 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.

Lemma 2. *O preserves finite connected limits.*

In [8, Section 2.4], we provide sufficient and necessary conditions on J and L for F to restrict as an endofunctor on $\mathscr{C}$.

Lemma 3. *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 $\mathscr{C}$.*

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

Notation 5. *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 2. The functor $\mathscr{D} \xrightarrow{\mathcal{L}} Kl_{\mathscr{D}^+}$ preserves finite connected colimits.

Notation 6. *Given $f \in \hom_{\mathscr{D}}(a, b)$, $u : Kb \to X$, we sometimes 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 $f \in \hom_{\mathscr{D}}(a, b)$, we denote the induced morphism $\overline{o} \to \overline{f \cdot o}$ by f^o.

Lemma 4. *A morphism $Ka \to FX$ is equivalently given by a morphism $o \in Ka \to O$, and a morphism $f : a^o \to X$.*

Proof. This follows from Property 1.(v).

Notation 7. *Given $o : Ka \to TX$ 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 4.*

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

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

2.3 Examples

The following table sketches some examples, detailed in [8]. The shape of metavariable arities determine the objects of $\mathcal{A}$, as hinted by Remark 4. In the case of quantum λ-calculus, $!\Delta$ consists of non linear types, whereas Γ, Δ consist of linear types only.

	Metavariable arity	Operations (example)
Pure λ-calculus	$n \in \mathbb{F}_m$	See the introduction.
Simply-typed λ-calculus	$\underbrace{\tau_1, \dots, \tau_n \vdash \tau_o}_{\text{simple types}}$	$\dfrac{\Gamma \vdash t : \tau_1 \Rightarrow \tau_2 \qquad \Gamma \vdash u : \tau_1}{\Gamma \vdash t\ u : \tau_2}$
System F	$m; \underbrace{\tau_1, \dots, \tau_n \vdash \tau_o}_{< m \text{ type variables}}$	$\dfrac{m+1 \vert wk(\Gamma) \vdash t : \tau}{m \vert \Gamma \vdash \Lambda t : \Lambda\tau}$ Type-Abstr $\dfrac{m \vert \Gamma \vdash t : \Lambda\tau \qquad m \vdash \sigma}{m \vert \Gamma \vdash t[\sigma] : \tau[* \mapsto \sigma]}$ Type-App

3 Main result

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

Definition 1. *Given a category $\mathscr{B}$, let $\mathscr{B}^*$ be $\mathscr{B}$ extended freely with a terminal object.*

Notation 8. *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 5. *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 6. *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 $Kl^*_{\mathscr{D}^+}$.

Theorem 1. *$Kl^*_{\mathscr{D}^+}$ has coequalisers.*

The pattern unification algorithm indeed provides a construction of coequalisers in $Kl^*_{\mathscr{D}^+}$.

4 Unification phase

In this section, we describe the main unification phase, which computes a coequaliser in $Kl^*_{\mathscr{D}^+}$. We use Notation 2 for coequalisers in $Kl^*_{\mathscr{D}^+}$, implicitly assuming that $A \in \mathscr{D}^+$ and $B, C \in \mathscr{D}^+ \cup \{\bot\}$.

We mainly focus on the rules dealing with a single term pair, in a metavariable context $\Gamma = \coprod_j Kb_j$, that is, a coequaliser diagram $Ka \overset{t}{\underset{u}{\rightrightarrows}} T\Gamma$. By Property 3, $t, u : Ka \to T\Gamma$ 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 §4.1),
- $t = M(\dots)$ and M does not occur in u (Section §4.2),
- t and u are $M(\dots)$ (Section §4.3),
- $t = M(\dots)$ and M occurs deeply in u (Section §4.4).

4.1 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 4, $o(\vec{t}[\sigma]) = o'(\vec{u}[\sigma])$. By Property 3, 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.

4.2 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 3.

Lemma 7 ([1], 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

$$\frac{\Gamma \vdash u' :> M(x) \Rightarrow v; \sigma \dashv \Delta \qquad u = Tin_\Gamma \circ u'}{\Gamma, M : b \vdash M(x) = u \Rightarrow \sigma, M \mapsto v \dashv \Delta} \tag{7}$$

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 8. *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 desired property is proven by induction.

Therefore, we can rephrase (7) as the rule U-NoCycle in Figure 1, using the following notation.

Notation 9. *Given $u : Ka \to T(\Gamma, M : b)$, we denote $m_{\Gamma;b} \circ u$ by $u_{|\Gamma}$. Moreover, and for any $u' : Ka \to T\Gamma$, we denote $in_1 \circ u' : Ka \to T\Gamma + 1$ by $\underline{u'}$.*

4.3 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 9. *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 1.(i) and Property 2, it can be computed as the image of the coequaliser of x and y, thus justifying the rule U-FlexFlex, using the following notation.

Notation 10. *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 $M \mapsto t$.*

4.4 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 appears in u, i.e., $\Gamma \to \Gamma, M : b$ does not factor u. In Section §6, we show that in this situation, there is no unifier. Then, Lemma 8 and Notation 9 justify the rule P-Fail.

5 Pruning phase

The pruning 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 use Notation 3 for pushouts, always implicitly assuming (and enforcing) that the right branch is a finite coproduct of free morphisms, as in the above span.

Remark 7. 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.

We have already discussed the structural rules of the pruning phase in Section §1.2. Let us add that the sequential rule P-Split is valid in any category.

Lemma 10. *In any category, denoting morphism composition $f \circ g$ by $g[f]$, the following rule applies.*

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

We now focus on the other rules of the pruning phase in Figure 1, dealing with a span $T\Gamma \xleftarrow{} Ka \xrightarrow{N(x)} T(N:b)$, that is, a single term in a metavariable context Γ. By Property 3, the left morphism $Ka \to T\Gamma$ is either flexible or rigid. Each case is handled separately in the following subsections.

5.1 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 $\vec{t} : \overline{o} \to T\Gamma$. By Remark 7, 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 4, this means that $o(\vec{t}[\sigma]) = x \cdot t'$. By Property 3, 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 4, so it cannot equal $o(\vec{t}[\sigma])$, by Property 3. Therefore, $t' = o'(\vec{u})$ for some $o' : Kb \to O$ and $\vec{u} : \overline{o'} \to T\Delta$. By Property 4, $x \cdot t' = (x \cdot o')(\vec{u} \circ x^o)$. By Property 3, $o = x \cdot o'$, and $\vec{t}[\sigma] = \vec{u} \circ x^o$. We introduce some notation for the latter condition.

In case $o = x \cdot o'$, 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 7, this is precisely the data for a pushout cocone for $\Gamma \xleftarrow{\vec{t}} a^o \xrightarrow{x^o} b^o$. This actually induces an isomorphism between the two categories of cocones. We therefore have the following rules.

$$\frac{\Gamma \vdash \vec{t} :> \mathcal{L}^+ x^o \Rightarrow \vec{u}; \sigma \dashv \Delta \qquad o = x \cdot o'}{\Gamma \vdash o(\vec{t}) :> N(x) \Rightarrow o'(\vec{u}); \sigma \dashv \Delta} \qquad \frac{o \neq x \cdot \ldots}{\Gamma \vdash o(\vec{t}) :> N(x) \Rightarrow !; ! \dashv \bot}$$

5.2 Flex

Here, we construct the pushout of $(\Gamma, M : c) \xleftarrow{M(y)} Ka \xrightarrow{N(x)} N : b$ where $y : a \to c$. 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 11. *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 1.(i) and Property 2, 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.

6 Occur-check

The occur-check allows to jump from the main unification phase (Section §4) to the pruning phase (Section §5), whenever the metavariable appearing at the top-level of the l.h.s does not appear in the r.h.s. This section is devoted to the proof that if there is a unifier, then either the metavariable does not occur in the r.h.s, either it occurs at top-level (see Corollary 2). 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, formally justified by exploiting the universal property of TX as the free F-algebra on X.

Definition 2. *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 $\vec{t} : \coprod_i Ka_i \to T\Gamma$.*

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

Definition 3. *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 12. *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 1. *For any $t : Ka \to T(\Gamma, M : b)$, $\sigma : \Gamma \to T\Delta$, $f \in \hom_{\mathscr{D}}(a, b)$, $u : Kb \to T\Delta$, we have $|t[\sigma, u]| \geq |t| + |u| \times |t|_M$ and $|\mathcal{L}f[u]| = |u|$.*

Corollary 2. *If there is a commuting square in* Kl_T

$$\begin{array}{ccc} Ka & \xrightarrow{t} & \Gamma, M : b \\ {\scriptstyle \mathcal{L}f}\downarrow & & \downarrow{\scriptstyle \sigma,u} \\ Kb & \xrightarrow[u]{} & \Delta \end{array}$$

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

Proof. Since $t[\sigma, u] = \mathcal{L}f[u]$, we have $|t[\sigma, u]| = |\mathcal{L}f[u]|$. Corollary 1 implies $|u| \geq |t| + |u| \times |t|_M$. Therefore, either $|t|_M = 0$ and we conclude by Lemma 12, either $|t|_M = 1$ and $|t| = 0$ and so t is $M(x)$ for some x.

7 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 et P-Split (see figure 1), the expressions with vector notation are not empty lists. The following two properties are then 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 5). 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 2 except that we assign a size of 1 to metavariables.

Let us first quickly justify termination of the pruning 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 pruning phase.

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 $(|\Gamma|, ||t|| + ||u||)$. The following lemmas 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 13. *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, i.e., it is* $\mathcal{L}^+\sigma'$ *for some* σ'*.*

Lemma 14. *For any* $t : Ka \to T\Gamma$ *and* $\sigma : \Gamma \to T\Delta$*, if* σ *is a renaming, then* $||t[\sigma]|| = ||t||$*.*

References

1. 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
2. D., P.G.: A note on inductive generalization. Machine Intelligence **5**, 153–163 (1970), https://cir.nii.ac.jp/crid/1573387448853680384
3. 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)
4. 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)
5. 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)
6. 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
7. 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
8. Lafont, A., Krishnaswami, N.: Generic pattern unification: a categorical approach (2022), https://raw.githubusercontent.com/amblafont/unification/master/lncs-long.pdf, preprint
9. 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
10. 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)
11. 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
12. Rydeheard, D.E., Burstall, R.M.: Computational category theory. Prentice Hall International Series in Computer Science, Prentice Hall (1988)
13. Vezzosi, A., Abel, A.: A categorical perspective on pattern unification. RISC-Linz p. 69 (2014)