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 setting for pattern unification using category theory. 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 of this monad. Beyond simply typed second-order syntax, our categorical proof handles unification for linear syntax.

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 [9] 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 [6], which is undecidable [5]. However, Miller [8] identified a decidable fragment: pattern unification allows metavariables to take some arguments, with the restriction that they must be distinct variables. In that case, we can design 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 analysed from a categorical point of view in [10,4]. However, there is little work on understanding pattern unification algebraically, with the notable exception of [11], working with normalised terms of simply-typed λ-calculus. The present paper can be seen as a generalisation of their work. An expanded version of this article with proofs and detailed examples can be found in [7].

Plan of the paper

In Section §2, we present our categorical setting. In Section §3, we state the main result that motivates the pattern unification algorithm. Then we describe the construction of the most general unifier, as summarised in Figure 1, starting with the unification phase (Section §4), the pruning phase (Section §5), the occur-check (Section §6). We finally justify completeness in Section §7.

Let us start by presenting pattern unification in the case of pure λ-calculus: we sketch the algorithm in Section §1.1, and in Section §1.2, we motivate our categorical setting, based on this example.

1.1 An example: pure $\boldsymbol{\lambda}$-calculus.

Consider the syntax of pure λ-calculus extended with metavariables satisfying the pattern restriction, encoded with De Bruijn levels. More formally, the syntax is inductively generated by the following inductive rules, where C is a variable context $(0 : \tau, 1 : \tau, \dots, n : \tau)$ with $|C| = n{+}1$, which we sometimes abbreviate 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 λt (in the context C) is $|C|$ rather than 0.

A *metavariable substitution* $\sigma : \Gamma \to \Gamma'$ assigns to each declaration $M : n$ in Γ a term $\Gamma'; n \vdash \sigma_M$. This assignation can be extended (through a recursive definition) to any term $\Gamma; C \vdash t$, yielding a term $\Gamma'; C \vdash t[\sigma]$. The basic case is $M(x_1, \dots, x_n)[\sigma] = \sigma_M[i \mapsto x_i]$, where $-[i \mapsto x_{i+1}]$ is variable renaming. Composition of substitutions $\sigma : \Gamma_1 \to \Gamma_2$ and $\sigma' : \Gamma_2 \to \Gamma_3$ can then be defined by $(\sigma[\sigma'])_M = \sigma_M[\sigma']$.

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. The motivation behind this notation is that the symbol $\Rightarrow$ separates the input and the output of a unification algorithm. We generalise this notation to lists of terms $\vec{t} = (t_1, \dots, t_n)$ and $\vec{u} = (u_1, \dots, u_n)$ such that $\Gamma; C_i \vdash t_i, u_i$. 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: we first compute the most general unifier σ_1 of (t_1, u_1), then apply the substitution to $(\vec{t_2}, \vec{u_2})$, and finally compute the most general unifier of the resulting list of term pairs:

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

This implies that we can focus on the unification of a single term pair.

To handle failing (when no unifier exists, for example, when unifying a λ-abstraction with an application), 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 $\Gamma \vdash t_1\ t_2 = \lambda u \Rightarrow\ ! \dashv \bot$.

The main unification phase consists in inspecting the structure of the given terms, until reaching a metavariable application $M(x_1, \dots, x_n)$ at top level. Once we reach a metavariable M on either hand side of $\Gamma, M : n \vdash t = u$, for example $t = M(x_1, \dots, x_n)$, three mutually exclusive situations must be considered:

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

In the third case, there is no unifier because the size of both hand sides can never match after substitution. This justifies the rule

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

where $u_{|\Gamma} = !$ means that u does not live in the smaller metavariable context Γ, and thus that M does appear in u.

In the second case, we only keep the argument positions that are the same in $x_1, \dots, x_n$ and $y_1, \dots, y_n$. In other words, the most general unifier substitutes M with $M'(z_1, \dots, z_p)$, where $z_1, \dots, z_p$ is the family of common positions i such that $x_i = y_i$. We denote such a situation by $n \vdash \vec{x} = \vec{y} \Rightarrow \vec{z} \dashv p$. The similarity with the above introduced notation will be categorically justified in Section §4: both are (co)equalisers. We therefore get the rule

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

Finally, the first case happens when we want to unify $M(\vec{x})$ 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* and tries to remove all *outbound* variables in u', i.e., variables that are not among $x_1, \dots, x_n$. It does so by producing a substitution that restricts the arities of the metavariables occurring in u'. Let us introduce a

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

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 $u'[\sigma][x_i \mapsto i]$, where $-[x_i \mapsto i]$ is renaming of free variables. . In fact, the output σ, v defines a substitution $(\Gamma, M : n) \to \Delta$ which can be characterised as the most general unifier of u' and $M(\vec{x})$. Note that M is not declared in Γ in this notation.

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 for the unification phase, we generalise the pruning phase to handle lists $\vec{u} = (u_1, \dots, u_n)$ of terms such that $\Gamma; C_i \vdash u_i$, and lists of pruning patterns $(\vec{x}_1, \dots, \vec{x}_n)$ where each $\vec{x}_i$ is a choice of distinct variables in C_i. Then, $\Gamma \vdash \vec{u} :> M_1(\vec{x}_1) + \dots + M_n(\vec{x}_n) \Rightarrow \vec{v}; \sigma \dashv \Delta$ means[2] that each σ is the common pruning substitution, and v_i is essentially $u_i[\sigma][x_{i,j} \mapsto j]$. Again, $\sigma, \vec{v}$ define a substitution from $\Gamma, M_1 : |\vec{x}_1|, \dots, M_n : |\vec{x}_n|$ to Δ which can be characterised as the most general unifier of $\vec{u}$ and $M_1(\vec{x}_1), \dots M_n(\vec{x}_n)$.

There is therefore an analogue to the above sequential rule U-Split.

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

We can then handle application as follows.

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

The above rule P-Split implies that we can focus on the unification of a singleton term pair. The variable case is straightforward.

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

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

$$\frac{\Gamma \vdash t :> M'(C, \vec{x}) \Rightarrow v; \sigma \dashv \Delta}{\Gamma \vdash \lambda t :> M(\vec{x}) \Rightarrow v; \sigma \dashv \Delta} \tag{5}$$

The remaining case consists in pruning a metavariable $N(\vec{y})$. In this situation, we need to consider the family $z_1, \dots, z_p$ of common values in $x_1, \dots, x_n$ and $y_1, \dots, y_m$, so that $z_i = x_{l_i} = y_{r_i}$ for some injections $l : \underline{p} \to \underline{n}$ and $r : \underline{p} \to \underline{m}$, where $\underline{q}$ denotes the set $\{0, \dots, q-1\}$. We denote such a situation by $m \vdash \vec{y} :> \vec{x} \Rightarrow \vec{r}; \vec{l} \dashv p$. The similarity with the pruning notation will be categorically justified in Section §5: both are (co)pushouts. In this situation, the metavariable N must be substituted with $N'(\vec{r})$ for some new metavariable N' of arity p, while the term $N(\vec{y})$ becomes $N'(\vec{l})$ in the variable context n:

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

[2] We justify in the next section the use of + as a list separator.

1.2 Categorification

In this section, we define the syntax of pure λ-calculus and state unification from a categorical point of view in order to motivate our general setting.

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 $\Lambda \cong F(\Lambda)$, while 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, if we want to extend the syntax with another metavariable of arity q, then the resulting functor would be $T(yp + yq)$.

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 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. We can give a formal meaning to the usual metavariable context notation.

Notation 1.1. *We denote a finite coproduct* $\coprod_{i \in \{M, N, \ldots\}} yn_i$ *of representable functors by a* (metavariable) context $M : n_M, N : n_N, \ldots$

To understand how a unification problem is stated in this general setting, let us come back to the example of pure λ-caNolculus. If $\Gamma = (M_1 : m_1, \ldots, M_p : m_p)$ and $\Delta = (N_1 : n_1, \ldots, n_q)$ are metavariable contexts, a Kleisli morphism $\sigma : \Delta \to T\Gamma$ is equivalently given (by the Yoneda Lemma and by the universal property of coproduct) by a λ-term $\Gamma; n_i \vdash \sigma_i$ for each $i \in \{1, \ldots q\}$. Note that this is precisely the data for a metavariable substitution $\Delta \to \Gamma$. Thus, Kleisli morphisms account for metavariable substitution and for term selection. Considering a pair of composable Kleisli morphisms $yp \to T\Gamma$ and $\Gamma \to T\Delta$, if we interpret the first one as a term $t \in T\Gamma_p$ and the second one as a metavariable substitution σ, then, the composition corresponds to the substituted term $t[\sigma]$.

A unification problem can be stated as a pair of parallel Kleisli morphisms $yp \overset{t}{\underset{u}{\rightrightarrows}} T\Gamma$ 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. The main purpose of the pattern unification algorithm consists thus in constructing this coequaliser, if it exists, which is the case as long as there exists a unifier.

We now sketch the generic unification algorithm as summarised in Figure 1, specialised to the pure λ-calculus. Let us first rephrase the unification notation in a general category.

Notation 1.2. *We denote a coequaliser* $A \overset{t}{\underset{u}{\rightrightarrows}} \Gamma \overset{\sigma}{\dashrightarrow} \Delta$ *in a category* $\mathscr{B}$ *by* $\Gamma \vdash t =_{\mathscr{B}} u \Rightarrow \sigma \dashv \Delta$*, sometimes even omitting* $\mathscr{B}$*.*

Note that if A is a coproduct $yn_1 + \cdots + yn_p$, then t and u can be thought of as lists of terms $\Gamma; n_i \vdash t_i, u_i$, hence the vector notation, which we also use for metavariable arguments, to make the rules even more similar to those of the previous section.

This notation is used in the unification phase with $\mathscr{B}$ the Kleisli category of T restricted to coproducts of representable functors, and extended with an error object $\perp$ (as formally explained 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 $p \xrightarrow{\vec{z}} n \overset{\vec{x}}{\underset{\vec{y}}{\rightrightarrows}} C$ is indeed an equaliser in $\mathbb{F}_m$.

Let us describe the other cases of the unification phase. The first structural rule deals with empty lists: there is nothing to unify. The second rule merely propagates the error. We have already explained the U-Split rule; it is in fact valid in any category (see [10, Theorem 9]). The other rules deal with singleton lists. The rigid-rigid rules handle all the cases where no metavariable is involved at top level. In the case of pure λ-calculus, the term $\Gamma; C \vdash o(\vec{t}; s)$ denotes a variable, an application, or a λ-abstraction depending on the label o in $\{v, a, l\}$. Indeed, $\vec{t}$ is a list of terms whose nature depends on o, and s is an element of $S_{o,C}$ for some functor $S_o : \mathbb{F}_m \to \mathrm{Set}$ depending on o. We summarise the different situations in the following table, where 1 denotes either a singleton set, either the constant functor $\mathbb{F}_m \to \mathrm{Set}$ mapping anything to a singleton set. Note that s is only relevant for the variable case, where it is indeed a choice of a variable. For simply-typed λ-calculus, however, S_o would be used to specify the output type of a λ-abstraction (see [7, Section 8.1]).

Unification Phase

- Structural rules (Section §4)

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

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

- Rigid-rigid (Section §4.1)

$$\frac{\Gamma \vdash \vec{t} = \vec{u} \Rightarrow \sigma \dashv \Delta}{\Gamma \vdash o(\vec{t}; s) = o(\vec{u}; s) \Rightarrow \sigma \dashv \Delta}\,\textsc{U-RigRig}$$

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

- Flex-*, no cycle (Section §4.2)

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

- Flex-Flex, same (Section §4.3)

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

- Flex-Rigid, cyclic (Section §4.4)

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

Pruning phase

- Structural rules (Section §5)

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

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

- Rigid (Section §5.1)

$$\frac{\Gamma \vdash \vec{t} :> \mathcal{L}^{+}\vec{x}^{o} \Rightarrow \vec{u}; \sigma \dashv \Delta \qquad s_{|\vec{x}} \Rightarrow \underline{s'}}{\Gamma \vdash o(\vec{t}; s) :> N(\vec{x}) \Rightarrow o(\vec{u}; s'); \sigma \dashv \Delta}\,\textsc{P-Rig} \qquad \frac{s_{|\vec{x}} \Rightarrow !}{\Gamma \vdash o(\vec{t}; s) :> N(\vec{x}) \Rightarrow !; ! \dashv \bot}\,\textsc{P-Fail}$$

- Flex (Section §5.2)

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

Figure 1. Summary of the rules

Operation	$o = ?$	$\vec{t}$	$s \in ?$
Variable	v	Empty list	$\underline{C} = I_C$
Application	a	$\Gamma; C \vdash t_1, t_2$	$1 = 1_C$
Abstraction	l	$\Gamma; C + 1 \vdash t$	$1 = 1_C$

The other rules of the unification phase follow the scheme described in the previous section. To formally understand the rule U-NoCycle which we have already introduced in (2), let us make the pruning notation precise.

Notation 1.3. *We denote a pushout diagram in a category $\mathscr{B}$ as below left by the notation as below right, sometimes even omitting $\mathscr{B}$.*

$$\begin{tikzcd} A \arrow[r, "f"] \arrow[d, "t"'] & \Gamma' \arrow[d, dashed, "u"] \\ \Gamma \arrow[r, dashed, "\sigma"'] & \Delta \arrow[ul, phantom, very near start, "\ulcorner"] \end{tikzcd} \quad \Leftrightarrow \quad \Gamma \vdash_{\mathscr{B}} t :> f \Rightarrow u; \sigma \dashv \Delta$$

Similarly to Notation 1.2, this is used in Figure 1 with $\mathscr{B}$ 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 $\mathbb{F}_m$.

$$\begin{tikzcd} p \arrow[r, "l"] \arrow[d, "r"'] & n \arrow[d, "x"] \\ m \arrow[r, "y"'] & C \end{tikzcd}$$

Let us add a few more comments about Notation 1.3. 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 (3) and in the rule P-Split.

We now describe the pruning phase as summarised in Figure 1. The structural rules perform a job similar to those of the unification phase. In the two rigid rules P-Rig and P-Fail, $\vec{x} = (x_1, \ldots, x_n)$ is a list of distinct variables chosen in a variable context C. Each rule handles a different situation about $\Gamma; C \vdash o(\vec{t}; s)$, where s is an object of $S_{o,C}$, as explained above.

The rule P-Rig assumes that there is $s' \in S_{o,n}$ such that s is the image by the renaming $\vec{x}$ of s', which we denote by $s_{|\vec{x}} \Rightarrow \underline{s'}$. This is trivially the case for application and abstraction. Therfore, this situation accounts for (3) and (5). The notation $\mathcal{L}^{+}\vec{x}^{o}$ indeed essentially unfolds to $M_1(\vec{x}) + M_2(\vec{x})$ in the application case, and to $M'(C, \vec{x})$ in the abstraction case.

The rule P-Fail happens when there is no such s', a situation which we denote by $s_{|\vec{x}} \Rightarrow !$.

In the variable case, this distinction corresponds to the two rules (4).

General notations

$\mathscr{B}^{op}$ denote 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 $(g_i : A_i \to B)_{i \in I}$ is a family of arrows, we denote by $[g_i] : \coprod_{i \in I} A_i \to B$ the induced coproduct pairing. If $f : A \to B$ and $g : A' \to B$, we sometimes denote the induced morphism $[f, g] : A + A' \to B$ by merely f, g. Conversely, if $g : \coprod_{i \in I} A_i \to B$, we denote by g_i the morphism $A_i \to \coprod_i A_i \to B$

Coproduct injections $A_i \to \coprod_{i \in I} A_i$ are typically denoted by in_i.

Given an adjunction $L \dashv R$ and a morphism $f : A \to RB$, we denote by $f^* : LA \to B$ its transpose, and similarly, if $g : LA \to B$, then $g^* : A \to RB$.

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 $\mathscr{B}$. Any Kleisli morphism $f : A \to TB$ induces a morphism $f^* : TA \to TB$. We denote the Kleisli composition of $f : A \to TB$ and $g : TB \to T\Gamma$ by $f[g] = g^* \circ f$.

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 2.1. The example of the introduction consider $\mathcal{A} = \mathbb{F}_m$ the category of finite cardinals and injections. Note that $\mathscr{C}$ is the category of nominal sets [3].

Remark 2.2. 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 $T\Gamma$ preserves them, i.e., $T\Gamma \in \mathscr{C}$. In fact, this specific coequaliser construction is not only justified in the Kleisli category restricted to the category of metavariable contexts, but in the Kleisli category restricted to objects of $\mathscr{C}$. However, it is not generally valid 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[3] a functor P that does not preserve finite connected colimits

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

such that $T(P)$ is the syntax extended with a binary commutative metavariable $M'(-,-)$. Then, the most general unifier, computed in the unrestricted Kleisli category of T, 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 2.3. 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 $\mathcal{C} \to [\mathcal{A}, \mathrm{Set}]$ factors the Yoneda embedding $\mathcal{A}^{op} \to [\mathcal{A}, \mathrm{Set}]$. We denote the fully faithful embedding as $\mathscr{D} \xrightarrow{K} \mathcal{C}$. A useful lemma that we will exploit is the following:

Lemma 2.4. *$\mathcal{C}$ is closed under limits, coproducts, and filtered colimits.*

In this rest of this section, we abstract this situation by listing a number of properties that we will use in the following to describe the main unification phase.

Property 2.5. The following hold.

(i) $K : \mathscr{D} \to \mathscr{C}$ is fully faithful.
(ii) $\mathscr{C}$ is cocomplete.

Notation 2.1. *We denote by $\mathscr{D}^+ \xrightarrow{K^+} \mathscr{C}$ the full subcategory of $\mathscr{C}$ consisting of finite coproducts of objects of $\mathscr{D}$.*

Remark 2.6. $\mathscr{D}^+$ is equivalent to the category of finite families of objects of $\mathcal{A}$. Thinking of objects of $\mathcal{A}$ as metavariable arities (Remark 2.3), $\mathscr{D}^+$ can be thought of as the category of metavariable contexts.

We adopt Notation 1.1 for objects of $\mathscr{D}^+$. We will be interested in coequalisers in the Kleisli category restricted to $\mathscr{D}^+$.

Property 2.7. The following properties hold.

(i) $\mathscr{D}$ has finite connected colimits.
(ii) K 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 2.8. Continuing Remark 2.2, unification of two metavariables as pullbacks or equalisers in $\mathcal{A}$ is justified by Property 2.7.(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

$$F(X) = \coprod_{o \in O} \prod_{j \in J_o} X \circ L_{o,j} \times S_o,$$

for some set O, where for each $o \in O$, $S_o \in \mathscr{C}$, J_o is a finite set, and $L_{o,j}$ is an endofunctor on $\mathcal{A}$ preserving finite connected limits for each $j \in J_o$.

Remark 2.9. S_o typically accounts for variables (in this case, J_o is empty) or can be used to specify the output type of an operation, in a simply-typed setting.

Lemma 2.10. *F is finitary and restricts as an endofunctor on $\mathscr{C}$.*

Corollary 2.11. *F 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 $\mathscr{D}^+$. Let us introduce a specific notation.

Notation 2.2. *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 $\mathscr{D} \hookrightarrow \mathscr{D}^+ \xrightarrow{\mathcal{L}^+} Kl_{\mathscr{D}^+}$.*

Notation 2.3. *Given $o \in O$, and $a \in \mathscr{D}$, we denote $\coprod_{j \in J_o} KL_{o,j}a$ by a^o. Given $f : a \to b$, we denote the induced morphism $a^o \to b^o$ by f^o.*

Lemma 2.12. *A morphism $Ka \to FX$ is equivalently given by $o \in O$, a morphism $s : Ka \to S_o$, and a morphism $f : a^o \to X$.*

Proof. This follows from Property 2.7.(v).

Notation 2.4. *Given $o \in O$, a morphism $s : Ka \to S_o$, and $\vec{t} : a^o \to TX$, we denote the induced morphism $Ka \to FTX \hookrightarrow TX$ by $o(\vec{t}; s)$, where the first morphism $Ka \to FTX$ is induced by Lemma 2.12.*

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

We now list a number of properties that we will exploit in the correctness proof.

Property 2.13. Let $\Gamma = M_1 : a_1, \ldots, M_n : a_n \in \mathscr{D}^+$. Then, any morphism $u : Ka \to T\Gamma$ is one of the two mutually exclusive following possibilities:

- $M_i(\vec{x})$ for some unique i and $\vec{x} : a \to a_i$,
- $o(\vec{t}; s)$ for some unique $o \in O$, $\vec{t} : a^o \to T\Gamma$ and $s : Ka \to S_o$.

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

Property 2.14. Let $\Gamma = M_1 : a_1, \ldots, M_n : a_n \in \mathscr{D}^+$ and $\sigma : \Gamma \to T\Delta$. Then, for any $o \in O$, $\vec{t} : a^o \to T\Gamma$ and $s : Ka \to S_o$, we have $o(\vec{t}; s)[\sigma] = o(\vec{t}[\sigma]; s)$, and for any $1 \le i \le n$, $x : b \to a_i$, we have $M_i(x)[\sigma] = \sigma_i \circ Kx$.

Moreover, for any $u : b \to a$,

$$o(\vec{t}; s) \circ Ku = o(\vec{t} \circ u^o; s \circ Ku)$$

Property 2.15. The functor $\mathscr{D} \xrightarrow{\mathcal{L}} Kl_{\mathscr{D}^+}$ preserves finite connected colimits.

We end this section by introducing a useful notation for Kleisli morphisms.

Notation 2.5. *Let Γ and Δ be contexts and $a \in \mathscr{D}$. Any $t : Ka \to T(\Gamma + \Delta)$ induces a Kleisli morphism $\Gamma, M : a \to T(\Gamma + \Delta)$ that we denote by $M \mapsto t$.*

2.3 Examples

The following table sketches some examples, detailed in [7]. The shape of metavariable arities determine the objects of $\mathcal{A}$, as hinted by Remark 2.3.

	Metavariable arity	Operations (examples)
Pure λ-calculus	$n \in \mathbb{F}_m$	See introduction.
Linear λ-calculus	$n \in \mathbb{N}$	$\dfrac{p \vdash t \quad q \vdash u}{p + q \vdash t\ u}$
Simply-typed λ-calculus	$\underbrace{\tau_1, \ldots, \tau_n \vdash \tau_o}_{\text{simple types}}$	$\dfrac{\Gamma \vdash t : \tau_1 \Rightarrow \tau_2 \qquad \Gamma \vdash u : \tau_1}{\Gamma \vdash t\ u : \tau_2}$

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

Notation 3.1. *We denote by $\perp$ 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 3.2. *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 $\mathscr{B}^*$.*

This lemma allows us to work with true coequalisers in $Kl^*_{\mathscr{D}+}$. The following result is also useful.

Lemma 3.3. *Given a category $\mathscr{B}$, the canonical embedding functor $\mathscr{B} \to \mathscr{B}^*$ creates colimits.*

This has the following useful consequences:

1. whenever the colimit in $Kl^*_{\mathscr{D}+}$ is not $\perp$, 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 $Kl^*_{\mathscr{D}+}$.

Here is our main result.

Theorem 3.4. *$Kl^*_{\mathscr{D}+}$ has coequalisers.*

4 Unification phase

In this section, we describe the main unification phase, whose goal is to compute a colimit in $Kl^*_{\mathscr{D}+}$. We heavily use Notation 1.2 for coequalisers in $Kl^*_{\mathscr{D}+}$, implicitly assuming that $A \in \mathscr{D}^+$ and $B, C \in \mathscr{D}^+ \cup \{\perp\}$.

We have already discussed the structural rules of Figure 1 in the introduction. What remains to be addressed is the case where the coproduct is a singleton and $\Gamma = \coprod_j Kb_j$, that is, a coequaliser diagram $Ka \underset{u}{\overset{t}{\rightrightarrows}} T\Gamma$. By Property 2.13, $t, u : Ka \to T\Gamma$ 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 want to unify $o(\vec{t}; s)$ and $o'(\vec{u}; s')$ for some $o, o' \in O$, morphisms $\vec{t} : a^o \to T\Gamma$, $\vec{u} : a^{o'} \to T\Gamma$, and morphisms $s : Ka \to S_o$ and $s' : Ka \to S_{o'}$.

Assume given a unifier $\sigma : \Gamma \to \Delta$. By Property 2.14, $o(\vec{t}[\sigma]; s) = o'(\vec{u}[\sigma]; s')$. By Property 2.13, this implies that $o = o'$, $\vec{t}[\sigma] = \vec{u}[\sigma]$, and $s = s'$.

Therefore, we get the following failing rules

$$\frac{o \neq o'}{\Gamma \vdash o(\vec{t}; s) = o'(\vec{u}; s') \Rightarrow\ ! \dashv \perp} \qquad \frac{s \neq s'}{\Gamma \vdash o(\vec{t}; s) = o(\vec{u}; s') \Rightarrow\ ! \dashv \perp}$$

We now assume $o = o'$ and $s = s'$. Then, σ is a unifier if and only if it unifies $\vec{t}$ and $\vec{u}$. This induces an isomorphism between the category of unifiers for $o(\vec{t}; s)$ and $o(\vec{u}; s)$ and the category of unifiers for $\vec{t}$ and $\vec{u}$, thus justifying the rule U-RigRig.

4.2 Flex-*, no cycle

Here we want to unify $M(\vec{x})$, which is nothing but $\mathcal{L}\vec{x}[in_M]$, and $u : Ka \to T(\Gamma, M : b)$, such that M does not occur in u, in the sense that there exists $u' : Ka \to T\Gamma$ such that $u = u'[in_\Gamma]$.

We exploit the following general lemma with $a = \mathcal{L}\vec{x}$ and $b = u'$.

Lemma 4.1 ([1], Exercise 2.17.1). *In any category, denoting morphism composition $g \circ f$ by $f[g]$, the following rule applies:*

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

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

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

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 intuitively tries to compute u' from an input data u.

Lemma 4.2. *There is a morphism $m_{\Gamma;b} : T(\Gamma, M : b) \to T\Gamma + 1$ such that the following square commutes and is a pullback.*

$$\begin{tikzcd}
T\Gamma \arrow[r, "Tin_\Gamma"] \arrow[d, equal] & T(\Gamma, M : b) \arrow[d, "m_{\Gamma;b}"] \\
T\Gamma \arrow[r, "in_1"'] & T\Gamma + 1
\end{tikzcd}$$

Proof. The proof consists in 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.

Notation 4.1. *Given $u : Ka \to T(\Gamma, M : b)$, we denote $m_{\Gamma;b} \circ u$ by $u_{|\Gamma}$. Moreover, we denote the morphism $Ka \xrightarrow{!} 1 \xrightarrow{in_2} T\Gamma + 1$ by merely $!$ and for any $u' : Ka \to T\Gamma$, we denote $in_1 \circ u' : Ka \to T\Gamma + 1$ by $\underline{u'}$.*

Corollary 4.3. *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 $u_{|\Gamma} = \underline{u'}$.*

Therefore, we can rephrase Rule (7) as follows.

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

4.3 Flex-Flex, same metavariable

Here we want to unify $M(\vec{x}) = \mathcal{L}\vec{x}[in_M]$ and $M(\vec{y}) = \mathcal{L}\vec{y}[in_M]$, with $\vec{x}, \vec{y} \in \hom_{\mathscr{D}}(a, b)$.

We exploit the following lemma in Kl_T, with $u = \mathcal{L}\vec{x}$ and $v = \mathcal{L}\vec{y}$.

Lemma 4.4. *In any category, denoting morphism composition $g \circ f$ by $f[g]$, the following rule applies:*

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

Therefore, it is enough to compute the coequaliser of $\mathcal{L}\vec{x}$ and $\mathcal{L}\vec{y}$. By Property 2.15, we finally get the rule U-FlexFlex.

4.4 Flex-rigid, cyclic

Here, we want to unify $M(\vec{x})$ for some $\vec{x} \in \hom_{\mathscr{D}}(a, b)$ 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. Using Corollary 4.3, we thus have the rule

$$\frac{u = o(\vec{t}; s) \qquad u_{|\Gamma} = !}{\Gamma, M : b \vdash M(\vec{x}) = u \Rightarrow\, ! \dashv \bot}$$

5 Pruning phase

The pruning phase corresponds to computing a pushout in $Kl^*_{\mathscr{D}+}$ of a span $\Gamma \xleftarrow{\vec{t}} \coprod_i a_i \xrightarrow{\coprod_i \mathcal{L}\vec{x}_i} \coprod_i b_i$, where the right branch is a finite coproduct of free morphisms. We heavily use Notation 1.3 for pushouts in $Kl^*_{\mathscr{D}+}$.

Remark 5.1. A cocone consists in morphisms $\coprod_i Kb_i \xrightarrow{\vec{u}} T\Delta \xleftarrow{\sigma} \Gamma$ such that $\vec{t}[\sigma] = \vec{u} \circ \coprod_i K\vec{x}_i$, i.e., for all $i \in I$, we have $t_i[\sigma] = u_i \circ K\vec{x}_i$.

Let us start with simple cases. 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.

The pushout can be decomposed into smaller components, thanks to the following lemma.

Lemma 5.2. *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 case where the coproduct is the singleton and $\Gamma \neq \bot$. Thus, we want to compute the pushout of $T\Gamma \longleftarrow Ka \xrightarrow{N(\vec{x})} T(N : b)$ in Kl_T. By Property 2.13, the left morphism $Ka \to T\Gamma$ is either flexible or rigid. Each case is handled seperately in the following subsections.

5.1 Rigid

Here, we want to compute the pushout of $\Gamma \xleftarrow{o(\vec{t};s)} Ka \xrightarrow{N(\vec{x})} N : b$ where $\vec{t} : a^o \to T\Gamma$ and $s : Ka \to S_o$. By Remark 5.1, a cocone in Kl_T is given by an object Δ with morphisms $Kb \xrightarrow{u} T\Delta \xleftarrow{\sigma} \Gamma$ such that $o(\vec{t};s)[\sigma] = u \circ K\vec{x}$. By Property 2.14, this means that $o(\vec{t}[\sigma];s) = u \circ K\vec{x}$. Now, by Property 2.13, u is either some $M'(\vec{x}')$ or $o'(\vec{t}';s')$, with $\vec{t}' : b^o \to T\Delta$ and $s' : Kb \to S_{o'}$. But in the first case, $u \circ K\vec{x} = M'(\vec{x}') \circ K\vec{x} = M'(\vec{x}' \circ \vec{x})$ so it cannot equal $o(\vec{t}[\sigma];s)$, by Property 2.13. So we are in the second case, and again by Property 2.13, $o = o'$, $\vec{t}[\sigma] = \vec{t}' \circ \vec{x}^o$ and $s = s' \circ K\vec{x}$.

Remark 5.3. Note that if there are at least two possible s', then a most general unifier cannot exist. But such a s', if it exists, is unique because $K\vec{x}$ is epimorphic by Property 2.7.(iii). In fact, this is the only place where we need that this epimorphicity. As a consequence, we could weaken the condition that morphisms in $\mathcal{A}$ are all monomorphic and require instead that for any morphism f in $\mathcal{A}$, the map $S_o f$ is monomorphic.

Before stating the rules that these considerations imply, let us introduce some notations.

Notation 5.1. *Given* $f \in \hom_{\mathscr{D}}(a,b)$ *and* $s : Ka \to S_o$, *we write* $s_{|f} \Rightarrow\, !$ *to mean that* Kf *does not factor* s. *Otherwise, if* $s = s' \circ Kf$, *then we write* $s_{|f} \Rightarrow \underline{s'}$.

Therefore we get the rules

$$\frac{\Gamma \vdash \vec{t} :> \mathcal{L}^+\vec{x}^o \Rightarrow \vec{u};\sigma \dashv \Delta \qquad s_{|\vec{x}} \Rightarrow \underline{s'}}{\Gamma \vdash o(\vec{t};s) :> N(\vec{x}) \Rightarrow o(\vec{u};s');\sigma \dashv \Delta} \qquad \frac{s_{|\vec{x}} \Rightarrow\, !}{\Gamma \vdash o(\vec{t};s) :> N(\vec{x}) \Rightarrow\, !;! \dashv \bot}$$

5.2 Flex

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

Lemma 5.4. *In any category, denoting morphism composition by* $f \circ g = g[f]$, *the following rule applies*

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

With Property 2.15, this justifies 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 the metavariable does not appear on the r.h.s, either it appears at top-level (see Corollary 6.6). The basic intuition is that $t = u[M \mapsto t]$ is impossible if M appears deep 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, that can be categorically justified by exploiting the universal property of TX as the free F-algebra on X.

Definition 6.1. *The size $|t| \in \mathbb{N}$ of a morphism $t : Ka \to T\Gamma$ is recursively defined by $|M(\vec{x})| = 0$ and $|o(\vec{t}; s)| = 1 + |\vec{t}|$, with $|\vec{t}| = \sum_i t_i$, for any $\vec{t} : \coprod_i Ka_i \to T\Gamma$.*

Definition 6.2. *For each morphism $t : Ka \to T(\Gamma, M : b)$ we define $|t|_M$ recursively by $|M(\vec{x})|_M = 1$, $|N(\vec{x})|_M = 0$* if $N \neq M$, and $|o(\vec{t}; s)|_M = |\vec{t}|_M$ *with the sum convention as above for $|\vec{t}|$.*

Lemma 6.3. *For any $t : Ka \to T(\Gamma, M : b)$, if $|t|_M = 0$, then $T\Gamma \hookrightarrow T(\Gamma, M : b)$ factors t.*

The crucial lemma is the following.

Lemma 6.4. *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 6.5. *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 6.6. *If there is a commuting square in Kl_T*

$$\begin{tikzcd} Ka \arrow[r, "t"] \arrow[d, "\mathcal{L}f"'] & \Gamma, M : b \arrow[d, "\sigma{,}u"] \\ Kb \arrow[r, "u"'] & \Delta \end{tikzcd}$$

then either $t = M(\vec{x})$ for some $\vec{x}$, 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 6.5 implies $|u| \geq |t| + |u| \times |t|_M$. Therefore, either $|t|_M = 0$ and we conclude by Lemma 6.3, either $|t|_M = 1$ and $|t| = 0$ and so t is $M(\vec{x})$ for some $\vec{x}$.

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. 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 2.6). We extend this definition to the case where $\Gamma = \bot$, by taking $|\bot| = 0$. We also define the size[4] $||t||$ of a *term* $t : Ka \to T\Gamma$ recursively by $||M(\vec{x})|| = 1$ and $||o(\vec{t}; s)|| = 1 + ||\vec{t}||$, where the size of a list of terms is the sum of the sizes of each term in the list.

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 7.1. *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 7.2. *For any $t : Ka \to T\Gamma$ and $\sigma : \Gamma \to T\Delta$, if σ is a renaming, then $||t[\sigma]|| = ||t||$.*

The proof of the first lemma relies on the fact that when the pruning phase does not fail, it produces a renaming targetting a metavariable context of the same size as the input one.

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

[4] The difference with the size definition in Section §6 is that metavariables are not of empty size. As a consequence, no term is of empty size.

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