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 and 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 [14] 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 [9], which is undecidable [8]. However, Miller [11] 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 [4], and later categorically analysed in [15,7]. However, there is little work on understanding pattern unification algebraically, with the notable exception of [17], working with normalised terms of simply-typed λ-calculus. The present paper 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 with the unification phase (Section §4), the pruning phase (Section §5), the occur-check (Section §6). We finally justify completeness in Section §7. Applications are presented in Section §8.

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

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. 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[1] a formal error metavariable context $\bot$ in which the only term (in any variable context) is a formal error term !, inducing a unique substitution $! : \Gamma \to \bot$, satisfying $t[!] = !$ for any term t. For example, we have $\Gamma \vdash t_1\ t_2 = \lambda u \Rightarrow ! \dashv \bot$.

We generalise the notation (and thus the input of the unification algorithm) to lists of terms $\vec{t} = (t_1, \dots, t_n)$ 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.

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

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

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

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

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, 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[2] such a situation by $n \vdash \vec{x} = \vec{y} \Rightarrow \vec{z} \dashv p$. We therefore get the rule

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

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., those that are not among $x_1, \ldots, x_n$. It does so 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 $u'[\sigma][x_{i+1} \mapsto i]$, where $-[x_{i+1} \mapsto i]$ is renaming of free variables. Note that M is not declared in Γ in this notation.

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 $M(\vec{x})$. We thus have the rule

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

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

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, $\Gamma \vdash \vec{u} :> M_1(\vec{x}_1) + \cdots + M_n(\vec{x}_n) \Rightarrow \vec{v}; \sigma \dashv \Delta$ means[3] 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.

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

As for unification (rule U-Split), pruning can be done sequentially, as in the rule P-Split. We can therefore focus on pruning a single term. 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 $N(y_1, \ldots, y_m)$, whose arity must then be restricted to those positions in $\vec{x}$. To be more precise, consider the family $z_1, \ldots, z_p$ of common values in $x_1, \ldots, x_n$ and $y_1, \ldots, y_m$, so that $z_i = x_{l_i} = y_{r_i}$ for some injections $l : \underline{p} \to \underline{n}$ and $r : \underline{p} \to \underline{m}$, where $\underline{q}$ denotes the set $\{0, \ldots, q-1\}$. We denote[4] such a situation by $m \vdash \vec{y} :> \vec{x} \Rightarrow \vec{r}; \vec{l} \dashv p$. Then, the metavariable N is 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:

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

1.2 Categorification

In this section, we define the syntax of pure λ-calculus from a categorical point of view in order to motivate our general setting. We then introduce 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 :

[3] The usage of + as a list separator will be justified in the next section.

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

$\mathbb{F}_m \to \mathrm{Set}$ 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[5] 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.

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

If $\Gamma = (M_1 : m_1, \ldots, M_p : m_p)$ 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 $i \in \{1, \ldots, p\}$. 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.

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

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. 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.

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 2. *We denote a coequaliser* $A \underset{u}{\overset{t}{\rightrightarrows}} \Gamma - \overset{\sigma}{\twoheadrightarrow} \Delta$ *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 $yn_1 + \cdots + yn_p$, then t and u can be thought of as lists of terms $\Gamma; n_i \vdash t_i, u_i$, hence the vector notation used in various rules (e.g., U-Split), which we also involve for metavariable arguments, to make the rules closer to those of the previous section.

The first structural rule of the unification phase 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 [15, Theorem 9]). 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 $a \xrightarrow{f} b \xleftarrow{g} c$. In this case, $a = yn$, $b = T\Gamma$, and c is a coproduct of representable presheaves and thus g can be thought as a list of terms (Remark 1), as hinted by the vector notation. Moreover, we implicitly assume that c is not the empty coproduct (i.e., $\vec{t_2}$ and $\vec{u_2}$ are not empty lists).

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 Table 1, 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

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)

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

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

– Flex-*, no cycle (Section §4.2)

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

– Flex-Flex, same (Section §4.3)

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

– Flex-Rigid, cyclic (Section §4.4)

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

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)

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

– Flex (Section §5.2)

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

Figure 1. Summary of the rules

case, where it is indeed a choice of variable in the ground context C. For simply-typed λ-calculus, however, S_o would be used to specify the output type of a λ-abstraction (see Section §8.1).

I don't understand why there is some much vertical space around this table

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

Table 1. Decomposing $o(\vec{t}, s)$ for pure λ-calculus

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 reinterpret the pruning notation.

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 $\perp$, with the exception of the premise of the rule P-Flex, where $\mathscr{B} = \mathscr{D}$ is the opposite category of $\mathbb{F}_m$. The latter corresponds to the above rule (6) whose premise precisely means that the following square is a pullback in $\mathbb{F}_m$.

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

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 (3) and in the rule P-Split.

The structural rules of the pruning phase are similar to those of the unification phase. The two rigid rules P-Rig and P-Fail are concerned with pruning a non metavariable term $\Gamma; C \vdash o(\vec{t}; s)$, where s is an object of $S_{o,C}$, as explained above. In the case of pure λ-calculus, $\vec{x} = (x_1, \ldots, x_n)$ is a list of distinct variables chosen in a variable context C. Only one of these two rules applies, depending on s. 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. Therefore, the rule 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 applies when there is no such s', a situation which we denote by $s_{|\vec{x}} \Rightarrow !$. In the variable case, these two rules instantiate to (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 $f : A \to B$ 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 [6].

Remark 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 $T\Gamma : \mathcal{A} \to \mathrm{Set}$ 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[6] a functor P that does not preserve finite connected colimits such that $T(P)$ is the syntax extended with a binary commutative metavariable $M'(-,-)$. Then, the most general unifier, computed in the total Kleisli category, replaces M with P. But in the Kleisli category restricted to coproducts of representable functors, or more generally, to objects of $\mathscr{C}$, the coequaliser replaces M with a constant metavariable, as expected.

Remark 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 $x_1, \dots, x_n \in \{0, \dots, m-1\}$, 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 $\mathscr{D} \xrightarrow{K} \mathcal{C}$. 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.*

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

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

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

Remark 4. $\mathscr{D}^+$ is equivalent to the category of finite families of objects of $\mathcal{A}$. Thinking of objects of $\mathcal{A}$ as metavariable arities (Remark 3), $\mathscr{D}^+$ 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.

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

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 $A_j \to \coprod_i A_i$.

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

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

$$y = J \circ K. \tag{7}$$

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

$$\begin{aligned}
\mathcal{C}(K \lim F, X) &\cong [\mathcal{A}, \mathrm{Set}](JK \lim F, JX) && (J \text{ is fully faithful}) \\
&\cong [\mathcal{A}, \mathrm{Set}](y \lim F, JX) && (\text{By (7)}) \\
&\cong JX(\lim F) && (\text{By the Yoneda Lemma.}) \\
&\cong \lim(JX \circ F) && (X \text{ preserves finite connected limits}) \\
&\cong \lim([\mathcal{A}, \mathrm{Set}](yF-, JX)] && (\text{By the Yoneda Lemma}) \\
&\cong \lim([\mathcal{A}, \mathrm{Set}](JKF-, JX)] && (\text{By (7)}) \\
&\cong \lim \mathcal{C}(KF-, X) && (J \text{ is full and faithful}) \\
&\cong \mathcal{C}(\mathrm{colim} KF, X) && (\text{By left continuity of the hom-set bifunctor})
\end{aligned}$$

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

(iii) A morphism $f : a \to b$ is epimorphic if and only if the following square is a pushout [10, Exercise III.4.4]

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

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

(iv) This follows from Lemma 1, because a morphism $f : A \to B$ is monomorphic if and only if the following square is a pullback

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

(v) This follows from coproducts being computed pointwise (Lemma 1), and representable functors being connected, by the Yoneda Lemma.

Remark 5. Continuing Remark 2, 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

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

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

Example 2. In Section §1.2, $O = \{v, a, l\}$, $J_v = \emptyset$, $J_a = \{1, 2\}$, $J_l = \{1\}$, and $L_{o,j}$ is $-+1$ if $o = l$, or the identity endofunctor otherwise. The functors S_o are given in Table 1.

Lemma 2. *F is finitary and restricts as an endofunctor on $\mathscr{C}$.*

Proof. F is finitary because filtered colimits commute with finite limits [10, Theorem IX.2.1] and colimits. It restricts as stated because finite connected limits commute with coproducts [2, Example 1.3.(vi)] and limits.

Corollary 1. *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 $\mathscr{C}$.*

Proof. By [13].

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.

Proof. $K : \mathscr{D} \to \mathscr{C}$ preserves finite connected colimits by Property 1.(ii). Thus, postcomposition with the left adjoint $\mathscr{C} \to Kl_T$ yields a functor $\mathscr{D} \to Kl_T$ preserving those colimits. The result follows because this functor factors as $\mathscr{D} \xrightarrow{\mathcal{L}} Kl_{\mathscr{D}^+} \hookrightarrow Kl_T$, where the right functor is full and faithful.

Notation 6. *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 3. *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 1.(v).

Notation 7. *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 3.*

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 Kleisli composition $Ka \xrightarrow{\mathcal{L}\vec{x}} Ka_i \xrightarrow{in_i} \Gamma$ by $M_i(\vec{x}) \in \hom_{Kl_T}(Ka, \Gamma) = \hom_{\mathscr{C}}(Ka, T\Gamma)$.

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

- $o(\vec{t}; s)[\sigma] = o(\vec{t}[\sigma]; s)$;
- $o(\vec{t}; s) \circ Ku = o(\vec{t} \circ u^o; s \circ Ku)$;
- $M_i(\vec{x})[\sigma] = \sigma_i \circ K\vec{x}$
- $M_i(\vec{x}) \circ Ku = M(\vec{x} \circ u)$

2.3 Examples

The following table sketches some examples, detailed in Section §8. The shape of metavariable arities determine the objects of $\mathcal{A}$, as hinted by Remark 3.

	Metavariable arity	Operations (examples)
Pure λ-calculus	$n \in \mathbb{F}_m$	See introduction.
Quantum 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 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 4. *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 $\mathscr{B}^*$.*

Proof. Straightforward, because a colimit is defined as an initial cocone.

The following result is also useful.

Lemma 5. *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 $B, C \in \mathscr{D}^+ \cup \{\bot\}$.

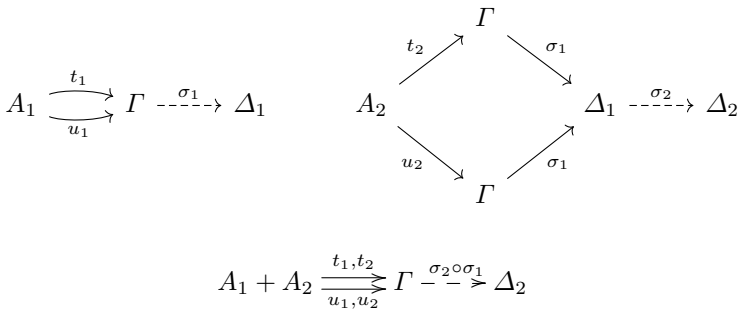
We have already discussed the structural rules of Figure 1 in Section §1.2.

Let us explicitly state the categorical involved lemma for the rule U-Split.

Lemma 6 (Theorem 9, [15]). *In any category, denoting morphism composition $f \circ g$ by $g[f]$, the following rule applies.*

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

In other words, if the first two diagrams below are coequalisers, then the last one as well



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

We now justify the other rules, dealing with a single term pair, in a metavariable context $\Gamma = \coprod_j Kb_j$, that is, a coequaliser diagram $Ka \underset{u}{\overset{t}{\rightrightarrows}} T\Gamma$. 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 $o(\vec{t}; s)$ 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 4, $o(\vec{t}[\sigma]; s) = o'(\vec{u}[\sigma]; s')$. By Property 3, this implies that $o = o'$, $\vec{t}[\sigma] = \vec{u}[\sigma]$, and $s = s'$. Therefore, we get the following failing rules

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

We now assume $o = o'$ and $s = s'$. 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 $o(\vec{t}; s)$ and $o(\vec{u}; s)$ and the category of unifiers for $\vec{t}$ and $\vec{u}$, justifying the rule U-RigRig.

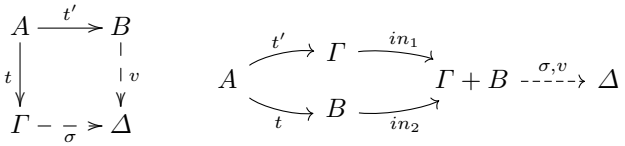
4.2 Flex-*, no cycle

Here we detail unification of $M(\vec{x})$, 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 $u = u'[in_\Gamma]$ for some $u' : Ka \to T\Gamma$. We exploit the following general lemma, recalling Notation 3.

Lemma 7 ([3], Exercise 2.17.1). *In any category, denoting morphism composition $g \circ f$ by $f[g]$, the following rule applies:*

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

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



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

$$\frac{\Gamma \vdash u' :> M(\vec{x}) \Rightarrow v; \sigma \dashv \Delta \qquad u = T in_\Gamma \circ u'}{\Gamma, M : b \vdash M(\vec{x}) = u \Rightarrow \sigma, M \mapsto v \dashv \Delta} \tag{8}$$

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 exsits $m_{\Gamma;b} : T(\Gamma, M : b) \to T\Gamma + 1$ 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 (see the induction lemma 12 introduced later).

Therefore, we can rephrase (8) 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, 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 $\underline{u'}$.*

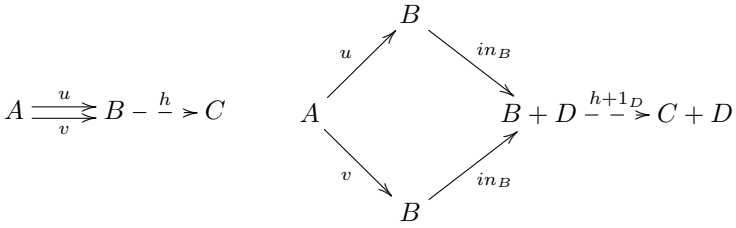
4.3 Flex-Flex, same metavariable

Here we detail unification of $M(\vec{x}) = \mathcal{L}\vec{x}[in_M]$ 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 with $u = \mathcal{L}\vec{x}$ and $v = \mathcal{L}\vec{y}$.

Lemma 9. *In any category, denoting morphism composition $g \circ f$ by $f[g]$, the following rule applies:*

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

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



It follows that it is enough to compute the coequaliser of $\mathcal{L}\vec{x}$ and $\mathcal{L}\vec{y}$. Furthermore, by Property 1.(i) and Property 2, it can be computed as the image of the coequaliser of $\vec{x}$ and $\vec{y}$, 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 $M(\vec{x})$ 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. 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 $\Gamma \xleftarrow{\vec{t}} \coprod_i Ka_i \xrightarrow{\coprod_i \mathcal{L}\vec{x}_i} \coprod_i Kb_i$. 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.

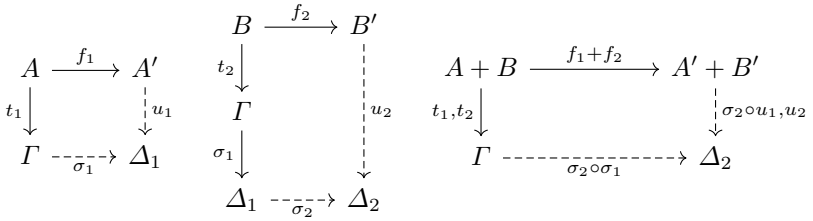
Remark 6. A pushout cocone for the above span consists in morphisms $\Gamma \xrightarrow{\sigma} T\Delta \xleftarrow{\vec{u}} \coprod_i Kb_i$ such that $\vec{t}[\sigma] = \vec{u} \circ \coprod_i K\vec{x}_i$, i.e., $t_i[\sigma] = u_i \circ K\vec{x}_i$ for each i.

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.*

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

In other words, if the first two diagrams below are pushouts, then the last one as well.



We now focus on the other rules of the pruning phase in Figure 1, dealing with a span $T\Gamma \xleftarrow{} Ka \xrightarrow{N(\vec{x})} T(N : b)$, 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 $\Gamma \xleftarrow{o(\vec{t};s)} Ka \xrightarrow{N(\vec{x})} N : b$ where $\vec{t} : a^o \to T\Gamma$ and $s : Ka \to S_o$. By Remark 6, a cocone is a cospan $T\Gamma \xrightarrow{\sigma} T\Delta \xleftarrow{t'} Kb$ such that $o(\vec{t}; s)[\sigma] = t' \circ K\vec{x}$. By Property 4, this means that $o(\vec{t}[\sigma]; s) = t' \circ K\vec{x}$. By Property 3, t' is either some $M(\vec{y})$ or $o'(\vec{u}; s')$, with $\vec{t'} : b^o \to T\Delta$ and $s' : Kb \to S_{o'}$. But in the first case, $t' \circ K\vec{x} = M(\vec{y}) \circ K\vec{x} = M(\vec{y} \circ \vec{x})$ by Property 4, so it cannot equal $o(\vec{t}[\sigma]; s)$, by Property 3. Therefore, $t' = o'(\vec{u}, s')$ for some $\vec{u} : b^o \to T\Delta$ and $s' : Kb \to S_{o'}$. Again, by Property 3, $o = o'$, $\vec{t}[\sigma] = \vec{u} \circ \vec{x}^o$ and $s = s' \circ K\vec{x}$. We introduce some notation for the latter condition.

Notation 11. *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 $s_{|f} \Rightarrow \underline{s'}$.*

Remark 7. Note that if there were more than one possible s' such that $s_{|f} \Rightarrow \underline{s'}$, then the most general unifier would not exist. But such a s', if it exists, is unique because $K\vec{x}$ is epimorphic, by Property 1.(iii). In fact, this is the only place where we use this property. As a consequence, we could weaken the condition that morphisms in $\mathcal{A}$ are monomorphic and require instead that the image of such a morphism by S_o is monomorphic.

In case $s_{|f} \Rightarrow \underline{s'}$, 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 $\vec{t}[\sigma] = \vec{u} \circ \vec{x}^o$. But, by Remark 6, this is precisely the data for a pushout cocone for $\Gamma \xleftarrow{\vec{t}} a^o \xrightarrow{\vec{x}^o} b^o$. This actually induces an isomorphism between the two categories of cocones. We therefore have the following rules.

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

5.2 Flex

Here, we construct 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 in this span, $N(\vec{x}) = \mathcal{L}\vec{x}$ while $M(\vec{y}) = \mathcal{L}\vec{y}[in_M]$. Thanks to the following lemma, it is actually enough to compute the pushout of $\mathcal{L}\vec{x}$ and $\mathcal{L}\vec{y}$.

Lemma 11. *In any category, denoting morphism composition by $f \circ g = g[f]$, the following rule applies*

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

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

$$\begin{array}{ccc} A & \xrightarrow{f} & B \\ {\scriptstyle g}\downarrow & & \vdots{\scriptstyle u} \\ X & \xrightarrow[\sigma]{} & Z \end{array} \qquad \begin{array}{ccc} A & \xrightarrow{f} & B \\ {\scriptstyle g}\downarrow & & \vdots{\scriptstyle u} \\ X & & Z \\ {\scriptstyle in_1}\downarrow & & \vdots{\scriptstyle in_1} \\ X + Y & \xrightarrow[\sigma + Y]{} & Z + Y \end{array}$$

By Property 1.(i) and Property 2, the pushout of $\mathcal{L}\vec{x}$ and $\mathcal{L}\vec{y}$ is the image by $\mathcal{L}$ of the pushout of $\vec{x}$ and $\vec{y}$, 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 3). 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* $|M(\vec{x})| = 0$ *and* $|o(\vec{t}; s)| = 1 + |\vec{t}|$, *with* $|\vec{t}| = \sum_i t_i$, *for any* $\vec{t} : \coprod_i Ka_i \to T\Gamma$.

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

Definition 3. *For each morphism* $t : Ka \to T(\Gamma, M : b)$ *we define* $|t|_M$ *recursively by* $|M(\vec{x})|_M = 1$, $|N(\vec{x})|_M = 0$ if $N \neq M$, and $|o(\vec{t}; s)|_M = |\vec{t}|_M$ *with the sum convention as above for* $|\vec{t}|_M$.

Remark 8. More formally, given $t : Ka \to T\Gamma$, the size $|t|$ is defined as the natural number n such that $1 \xrightarrow{n} \mathbb{N}$ factors $Ka \to T\Gamma \to \mathbb{N}$ (by Property 1.(v), since $\mathbb{N}$ is the coproduct $\coprod_{n\in\mathbb{N}} 1$), where $T\Gamma \to \mathbb{N}$ is the universal F-algebra morphism induced by the constant morphism $\Gamma \xrightarrow{0} \mathbb{N}$ and the F-algebra $F\mathbb{N} \cong \coprod_o \mathbb{N}^{J_o} \times S_o \xrightarrow{\coprod_o \pi_1} \coprod_o \mathbb{N}^{J_o} \xrightarrow{\coprod_o (1+\sum)} \coprod_o \mathbb{N} \to \mathbb{N}$, informally mapping $o(\vec{n}; s)$ to $1 + \sum_j n_j$.

Given $t : Ka \to T(\Gamma, M : b)$, the natural number $|t|_M$ is computed similarly by the postcomposition with the universal F-algebra morphism $T(\Gamma, M : b) \to \mathbb{N}$ induced by $\Gamma, M : b \xrightarrow{0,1} \mathbb{N}$ and the F-algebra structure $F\mathbb{N} \cong \coprod_o \mathbb{N}^{J_o} \times S_o \xrightarrow{\coprod_o \pi_1} \coprod_o \mathbb{N}^{J_o} \xrightarrow{\coprod_o \sum} \coprod_o \mathbb{N} \to \mathbb{N}$, informally mapping $o(\vec{n}; s)$ to $\sum_j n_j$.

The lemmas below are easy consequences of the following standard induction lemma.

Lemma 12. *Assume given, for each object* a *of* $\mathcal{A}$, *a predicate* P_a *on* $\hom(Ka, T\Gamma)$ *such that*

- $P_a(M(\vec{x}))$ *holds for any* $M : b \in \Gamma$ *and* $\vec{x} \in \hom_{\mathscr{D}}(a, b)$;
- $P_a(o(\vec{t}; s))$ *holds for any* $o \in O$, $s : Ka \to S_o$, *and* $\vec{t} : \coprod_{j\in J_o} KL_{o,j}a \to S_o$ *such that* $P_{L_{o,j}a}(t_j)$ *holds for every* $j \in J_o$.

Then, $P_a(t)$ *holds for any* $t : Ka \to T\Gamma$.

Proof. Consider the functor $X : \mathcal{A} \to \mathrm{Set}$ defined by $X_a = \{t \in Ka \to T\Gamma | \forall f : b \to a, P_a(t \circ Kf)\}$. By the Yoneda lemma, there is an injective projection $X \to T\Gamma$. By universal property of $T\Gamma$ as the free F-algebra on Γ, this projection has a section, and is thus an isomorphism.

Lemma 13. *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 14. *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 2. *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 3. *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 $t = M(\vec{x})$ *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 2 implies $|u| \geq |t| + |u| \times |t|_M$. Therefore, either $|t|_M = 0$ and we conclude by Lemma 13, 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 4). We extend this definition to the case where $\Gamma = \bot$, by taking $|\bot| = 0$. We also define the size[7] $||t||$ 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 15. *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'*.*

[7] The difference with Definition 2 is that metavariables are no longer of empty size. As a consequence, any term has a non empty size.

Lemma 16. *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.

Lemma 17. *If there is a finite derivation tree of $\Gamma \vdash f :> g \Rightarrow u; \sigma \dashv \Delta$ and $\Delta \neq \bot$, then $|\Gamma| = |\Delta|$ and σ is a* renaming.

8 Applications

In the following examples, we always motivate the definition of the category $\mathcal{A}$ based on what we expect from metavariables arities and their substitution, following Remark 3.

8.1 Simply-typed second-order syntax

In this section, we present the example of simply-typed λ-calculus. Our treatment generalises to any second-order binding signature (see [5]). Let T denote the set of simple types generated by a set of simple types. A metavariable arity $\tau_1, \ldots, \tau_n \vdash \tau_f$ is given by a list of input types $\tau_1, \ldots, \tau_n$ and an output type τ_f. Substituting a metavariable $M : (\Gamma \vdash \tau)$ with another $M' : (\Gamma' \vdash \tau')$ requires that $\tau = \tau'$ and involves an injective renaming $\Gamma \to \Gamma'$. Thus, we consider $\mathcal{A} = \mathbb{F}_m[T] \times T$, where $\mathbb{F}_m[T]$ is the category of finite lists of elements of T and injective renamings between them.

The following table summarises what we expect from the endofunctor F on $[\mathcal{A}, \mathrm{Set}]$ specifying the syntax, where $|\Gamma|_\tau$ denotes the number (as a cardinal set) of occurrences of τ in Γ, for each construction of the syntax.

Operation	Typing rule	$F(X)_{\Gamma \vdash \tau} = ?$
Variable	$\dfrac{x : \tau \in \Gamma}{\Gamma \vdash x : \tau}$	$\lvert\Gamma\rvert_\tau$
Application	$\dfrac{\Gamma \vdash t : \tau_1 \Rightarrow \tau_2 \qquad \Gamma \vdash u : \tau_1}{\Gamma \vdash t\ u : \tau_2}$	$\coprod_{\tau_1} X_{\Gamma \vdash \tau_1 \Rightarrow \tau} \times X_{\Gamma \vdash \tau_1}$
Abstraction	$\dfrac{\Gamma, x : \tau_1 \vdash t : \tau_2}{\Gamma \vdash \lambda x.t : \tau_1 \Rightarrow \tau_2}$	$X_{\Gamma, \tau_1 \vdash \tau_2}$ if $\tau = \tau_1 \Rightarrow \tau_2$

Let us denote by $\underline{\Gamma} : \mathcal{A} \to \mathbb{F}_m[T]$ the first projection and by $\underline{\tau} : \mathcal{A} \to T$ the second one. Defining $F(X)$ as the coproduct of each row of the last column in the following table satisfies those expectations.

Operation	Typing rule	$F(X) = ?$
Variable	$\dfrac{x : \tau \in \Gamma}{\Gamma \vdash x : \tau}$	$\coprod_\tau y(\tau \vdash \tau)$
Application	$\dfrac{\Gamma \vdash t : \tau_1 \Rightarrow \tau_2 \qquad \Gamma \vdash u : \tau_1}{\Gamma \vdash t\ u : \tau_2}$	$\coprod_{\tau_1} X_{\underline{\Gamma} \vdash \tau_1 \Rightarrow \underline{\tau}} \times X_{\underline{\Gamma} \vdash \tau_1}$
Abstraction	$\dfrac{\Gamma, x : \tau_1 \vdash t : \tau_2}{\Gamma \vdash \lambda x.t : \tau_1 \Rightarrow \tau_2}$	$\coprod_{\tau_1,\tau_2} X_{\underline{\Gamma},\tau_1 \vdash \tau_2} \times y(\cdot \vdash \tau_1 \Rightarrow \tau_2)$

This definition can be adapted to fit into the scheme of Section §2.2 by having in O one label v_τ for each simple type τ, accounting for variables, one label a_τ for each τ, accounting for application, and one label l_{τ_1,τ_2} for each pair of simple types (τ_1, τ_2), accounting for applications.

Operation	S_o	J_o	$L_{o,j} : \mathcal{A} \to \mathcal{A}$
Variable	$S_{v_\tau} = y(\tau \vdash \tau)$	$J_{v_\tau} = 0$	-
Application	$S_{a_\tau} = 1$	$J_{a_\tau} = 2$	$L_{a_\tau,1} = \underline{\Gamma} \vdash \tau \Rightarrow \underline{\tau}$ $L_{a_\tau,2} = \underline{\Gamma} \vdash \tau$
Abstraction	$S_{l_{\tau_1,\tau_2}} = y(\cdot \vdash \tau_1 \Rightarrow \tau_2)$	$J_{l_{\tau_1,\tau_2}} = 1$	$L_{l_{\tau_1,\tau_2}} = \underline{\Gamma}, \tau_1 \vdash \tau_2$

8.2 Arguments as sets

If we think of the arguments of the metavariable as specifying the available variables, then it makes sense to gather them in a set rather than in a list. This motivates considering the category $\mathcal{A} = \mathbb{I}$ whose objects are natural numbers and a morphism $n \to p$ is a subset of $\{0, \ldots, p-1\}$ of cardinal n. For instance, $\mathbb{I}$ can be taken as subcategory of $\mathbb{F}_m$ consisting of strictly increasing injections, or as the subcategory of the augmented simplex category consisting of injective functions. Again, we can define the endofunctor for λ-calculus as in Section §1.2. Then, a metavariable takes as argument a set of available variables, rather than a list of distinct variables. In this approach, unifying two metavariables (see the rules U-FlexFlex and P-Flex) both amount to computing a set intersection.

8.3 Arities as sets

In this example, we describe pure λ-calculus extended with metavariable whose arities are sets of free variables. They do not take any explicit argument and they cannot be applied to bound variables.

We adopt a locally nameless approach, with two kinds of variables: the named ones, chosen in an infinite set $\mathcal{V}$ of names (e.g., $\mathbb{N}$), and the unnamed ones, as before, which will be used for binding. We thus choose $\mathcal{A}$ to be $\mathbb{S} \times \mathbb{F}_m$ where $\mathbb{S}$ is the category of finite subsets of $\mathcal{V}$ and inclusions (not injections!) between

them. Note that pure λ-calculus can be specified by an endofunctor F defined by $F(X)_{A,n} = n + A + X_{A,n+1} + X_{A,n} \times X_{A,n}$.

A metavariable arity, as an object of $\mathcal{A}$, consists of two components: the first one is a finite set of named variables, and the second one specifies a number of arguments among unnamed variables. We say that an arity is pure if the second component 0, and a metavariable is said pure if its arity is. The pure metavariables are the ones mentioned at the beginning of this section. Unifying a pure metavariable with itself, as in the rule U-FlexFlex, is a no-op, while unifying a pure metavariable with another one (rule P-Flex) produces a new pure metavariable whose arity is the intersection of the input metavariable arities. Exploiting this observation, an easy induction is enough to show that the most general unifier targets a pure metavariable context.

Lemma 18. *Assume an endofunctor for syntax as in Section §2.2. If $\Gamma \vdash t = u \Rightarrow \sigma \dashv \Delta$ or $\Gamma \vdash t :> \coprod_i \mathcal{L} f_i \Rightarrow u; \sigma \dashv \Delta$, then Δ is a coproduct of pure arities whenever Γ is, and $\Delta \neq \bot$.*

8.4 Quantum λ-calculus

In this section we explain how the quantum λ-calculus of [12] fits into our setting. We denote by S the set of types, which is inductively generated as follows

$$A, B, C \in S ::= \mathbf{qubit} | A \multimap B | !(A \multimap B) | 1 | A \otimes B | A + B | A^{\ell}$$

where A^{ℓ} is intuitively the type of finite lists of type A. We denote by $!S$ the set of non-linear types, that is, types of the shape $!-$. In fact $!S$ is isomorphic to $S \times S$, since any non linear type must be of the shape $!(A \multimap B)$.

A metavariable could have linear arguments, and non linear ones. Therefore, we take $\mathcal{A}$ to be $\mathbb{F}_m[A$

We take the example of linear λ-calculus. A metavariable arity is then a natural number specifying the number of arguments, and because of linearity, a metavariable can only be substituted with another one with the same arity. We therefore $\mathcal{A} = \mathcal{P}$ to be category whose objects are finite cardinals and morphisms are bijections as in [16].

Each typing rule provides an operation

$$\text{Variable} \quad 1 \vdash * \qquad\qquad y1$$

$$\text{Application} \quad \frac{n \vdash t \qquad m \vdash u}{n + m \vdash t\ u} \quad X_n \times X_m \times y_{n+m}$$

$$\Gamma$$

References

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

2. AdÃ¡mek, J., Borceux, F., Lack, S., RosickÃœ, J.: A classification of accessible categories. Journal of Pure and Applied Algebra **175**(1), 7–30 (2002). https://doi.org/https://doi.org/10.1016/S0022-4049(02)00126-3, https://www.sciencedirect.com/science/article/pii/S0022404902001263, special Volume celebrating the 70th birthday of Professor Max Kelly
3. Borceux, F.: Handbook of Categorical Algebra 1: Basic Category Theory. Encyclopedia of Mathematics and its Applications, Cambridge University Press (1994). https://doi.org/10.1017/CBO9780511525858
4. D., P.G.: A note on inductive generalization. Machine Intelligence **5**, 153–163 (1970), https://cir.nii.ac.jp/crid/1573387448853680384
5. Fiore, M.P., Hur, C.K.: Second-order equational logic. In: Proceedings of the 19th EACSL Annual Conference on Computer Science Logic (CSL 2010) (2010)
6. 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)
7. 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)
8. 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
9. 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
10. Mac Lane, S.: Categories for the Working Mathematician. No. 5 in Graduate Texts in Mathematics, Springer, 2nd edn. (1998)
11. 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
12. Pagani, M., Selinger, P., Valiron, B.: Applying quantitative semantics to higher-order quantum computing. In: Jagannathan, S., Sewell, P. (eds.) The 41st Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL '14, San Diego, CA, USA, January 20-21, 2014. pp. 647–658. ACM (2014). https://doi.org/10.1145/2535838.2535879, https://doi.org/10.1145/2535838.2535879
13. Reiterman, J.: A left adjoint construction related to free triples. Journal of Pure and Applied Algebra **10**, 57–71 (1977)
14. 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
15. Rydeheard, D.E., Burstall, R.M.: Computational category theory. Prentice Hall International Series in Computer Science, Prentice Hall (1988)
16. Tanaka, M.: Abstract syntax and variable binding for linear binders. In: MFCS '00. LNCS, vol. 1893. Springer (2000)
17. Vezzosi, A., Abel, A.: A categorical perspective on pattern unification. RISC-Linz p. 69 (2014)