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 the most general unifier of two terms involving metavariables. It is used in logic programming languages such as Prolog, or in proof search, type inference algorithms. Pattern unification [6] 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. In this work, we present a generic construction of the most general unifier, in a categorical setting.

Related work

First-order unification, where all metavariables are constant, was categorically rephrased in [7, Chapter 9]. Pattern unification was introduced in [6], as a particular case of higher-order unification for the simply-typed lambda-calculus, where metavariables are applied to distinct variables. It was categorically rephrased in [9]. 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 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. 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 in Section §1.2, we motivate our categorical setting, 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, defined by the following inductive rules, where $C \in \mathbb{N}$ is a ground context and Γ is a metavariable context $M_1 : n_1, \ldots, M_m : n_m$ specifying a metavariable symbol M_i together with its number of arguments n_i.

$$\frac{x < C}{\Gamma; C \vdash x} \qquad \frac{\Gamma; C \vdash t \quad \Gamma; C \vdash u}{\Gamma; C \vdash t\ u} \qquad \frac{\Gamma; C+1 \vdash t}{\Gamma; C \vdash \lambda t}$$

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

We choose the convention that the variable bound in λt (in the context C) is C, rather than the usual De Bruijn choice of 0. The benefit is that there is no need to shift free variables under a λ, making substitution simpler. Whether a variable is bound or not depends on the ground context.

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 $M(x_1, \ldots, x_n)[\sigma] = \sigma_M[i \mapsto x_i]$, 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 ground context C implicit. The motivation behind this notation is that the symbol $\Rightarrow$ separates the input and the output of a unification algorithm.

Let us now describe the pattern unification algorithm. To handle the failing case (when no unifier exists), we add[1] a formal error metavariable context $\bot$ in which the only term (in any ground context) is a formal error term !, inducing a unique substitution $! : \Gamma \to \bot$, satisfying $t[!] = !$ for any term t.

The main unification phase consists in inspecting the structure of the given pair of terms, until reaching a metavariable application $M(x_1, \ldots, x_n)$ at top level. For the sake of brevity, we skip the error cases and the variable cases.

The congruence case for λ-abstraction is straightforward.

$$\frac{\Gamma \vdash t = u \Rightarrow \sigma \dashv \Delta}{\Gamma \vdash \lambda x.t = \lambda x.u \Rightarrow \sigma \dashv \Delta}$$

For the congruence case for application $t_1\ t_2 = u_1\ u_2$, we need to unify both (t_1, u_1) and (t_2, u_2). Let us introduce the notation $\Gamma \vdash t_1, u_1 = t_2, u_2 \Rightarrow \sigma \dashv \Delta$,

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

meaning that $\sigma : \Gamma \to \Delta$ unifies both (t_1, u_1) and (t_2, u_2), and is the most general unifier, in the sense that it uniquely factors any other unifier of both term pairs.

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

Computing the most general unifier of a list of term pairs can be done sequentially. In the example above, we first compute the most general unifier σ_1 of (t_1, u_1), apply the substitution to (t_2, u_2) and compute the most general unifier of the resulting term pair:

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

Once we reach a metavariable M on either hand side of $\Gamma, M : n \vdash t = u$, for example $t = M(x_1, \ldots, x_n)$, three mutually exclusive situations must be analysed:

1. M does not appear in u;
2. M appears in u at the 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, we only keep the argument positions that are the same in $x_1, \ldots, x_n$ and $y_1, \ldots, y_n$. In other words, 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 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 $x_1, \ldots, x_n$. 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 $u'[\sigma][x_i \mapsto i]$, where $-[x_i \mapsto i]$

is renaming of free variables. In this notation, M is a metavariable symbol that is not declared in Γ, and the ground context C for u' is left implicit.

We thus have the rule

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

The pruning phase consists in deconstructing the input term until reaching a metavariable. The variable case is straightforward.

$$\frac{y \in \vec{x}}{\Gamma \vdash y :> M(\vec{x}) \Rightarrow y; 1_\Gamma \dashv \Gamma} \qquad \frac{y \notin \vec{x}}{\Gamma \vdash y :> M(\vec{x}) \Rightarrow !; ! \dashv \bot} \tag{2}$$

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

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

Application requires to prune two terms, justifying the premise with the intuitive notation in the following rule.

$$\frac{\Gamma \vdash t, u :> M(\vec{x}) + M(\vec{x}) \Rightarrow v; \sigma \dashv \Delta}{\Gamma \vdash t\ u :> M(\vec{x}) \Rightarrow v; \sigma \dashv \Delta} \tag{4}$$

This is done sequentially by first pruning the first term, applying the pruning substitution to the second term, and finally pruning the resulting term.

$$\frac{\Gamma \vdash u_1 :> M(\vec{x}) \Rightarrow v_1; \sigma_1 \dashv \Delta_1 \qquad \Delta_1 \vdash u_2[\sigma_1] :> M(\vec{x}) \Rightarrow v_2; \sigma_2 \dashv \Delta_2}{\Gamma \vdash u_1, u_2 :> M(\vec{x}) + M(\vec{x}) \Rightarrow v_1[\sigma_2], v_2; \sigma_1[\sigma_2] \dashv \Delta_2}$$

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 ground context n:

$$\frac{m \vdash \vec{y} :> \vec{x} \Rightarrow \vec{r}; \vec{l} \dashv p}{\Gamma, N : m \vdash N(\vec{y}) :> M(\vec{x}) \Rightarrow N'(\vec{l}); N \mapsto N'(\vec{r}) \dashv \Gamma, N' : p} \tag{5}$$

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 $\underline{n} = \{0, \dots, n-1\}$. 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. Note that $Inj(p, n)$ 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 Σ the endofunctor on $[\mathbb{F}_m, \mathrm{Set}]$ mapping F to $I + F \times F + F(-+1)$, where I is the functor mapping n to $\underline{n}$, the functor Λ can be characterised as the initial algebra for Σ, thus satisfying the recursive equation $\Lambda \cong \Sigma(\Lambda)$, while Λ' is characterised as the initial algebra for $\Sigma(-) + yp$, where yp is the (representable) functor $\mathbb{F}_m(p, -) : \mathbb{F}_m \to \mathrm{Set}$, thus satisfying the recursive equation $\Lambda' \cong \Sigma(\Lambda') + yp$. In other words, Λ' is the free Σ-algebra on yp. Denoting T the free Σ-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 Σ on it. Then, the abstract definition of a syntax extended with metavariables is the free Σ-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. A Kleisli morphism $\sigma : yp \to T(yn)$ is equivalently given (by the Yoneda Lemma) by an element of $T(yn)_p$, that is, a λ-term $M : n; p \vdash t$. Note that this is the necessary data to substitute a metavariable N of arity p. Thus, Kleisli morphisms account for metavariable substitution and for term selection. Considering a pair of composable Kleisli morphisms $yp \to T(yn)$ and $yn \to T(ym)$, if we interpret the first one as a term $t \in T(yn)_p$ 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 $yp \underset{u}{\overset{t}{\rightrightarrows}} T(yq_1 + \cdots + yq_n)$ corresponding to selecting a pair of terms $M_1 : q_1, \dots, M_n : q_n; p \vdash t, u$. 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, starting with the unification phase for a list of term pairs. The first structural rule handles the case of an empty list of term pairs: there is nothing to unify. The second rule merely propagates the error. The third structural rule performs sequential unification of a non empty list of term pairs, as described earlier.

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 $\Gamma; C \vdash o(f; s)$ denotes a variable, an application, or a λ-abstraction depending on the label o in $\{v, a, l\}$. Indeed, f 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.

Operation	$o = ?$	f	$s \in ?$
Variable	v	Empty list	$\underline{C} = I_C$
Application	a	$\Gamma; C \vdash f_1, f_2$	$1 = 1_C$
Abstraction	l	$\Gamma; C + 1 \vdash f$	$1 = 1_C$

The other rules of the unification phase follow the scheme described in the previous section. As we will see in Section §4, the premise of the rule (1) precisely means that $p \xrightarrow{z} n \underset{y}{\overset{x}{\rightrightarrows}} C$ is an equaliser in $\mathbb{F}_m$.

We now comment the pruning phase. The structural rules perform a job similar to those of the unification phase. The metavariable pruning case (Flex) is textually similar to the rule (5). As we will see in Section §5, the premise of the rule (5) precisely means that the following square is a pullback in $\mathbb{F}_m$.

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

Finally, let us comment the two rigid rules. In those situations, $f = (x_1, \ldots, x_n)$ is a list of distinct variables chosen in a ground context C. Equivalently, f is a morphism $n \to C$ in $\mathbb{F}_m$. In $\Gamma; C \vdash o(g; s)$, as explained above, s is then an object of $S_{o,C}$. The two rules examines two different cases: the first is when there is $s' \in S_{o,n}$ such that s is the image by the renaming f of s', which we denote by $s_{|f} \Rightarrow \underline{s'}$, and the second is when there is no such s', a situation which we denote by $s_{|f} \Rightarrow !$. In the variable case, this distinction corresponds to the two rules (2). In the case of an application or an abstraction, the second rule never applies, and the first rule accounts for the rules (3) and (4). Indeed, the notation $\mathcal{L}f^o$ unfolds to $M(\vec{x}) + M(\vec{x})$ in the application case, and to $M(C, \vec{x})$ in the abstraction case.

Unification phase

– Structural rules (Section §4)

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

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

– Rigid-rigid (Section §4.1)

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

$$\frac{o \neq o'}{\Gamma \vdash o(f; s) = o'(f'; s') \Rightarrow ! \dashv \bot} \qquad \frac{s \neq s'}{\Gamma \vdash o(f; s) = o(f'; s') \Rightarrow ! \dashv \bot}$$

– Flex-*, no cycle (Section §4.2)

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

– Flex-Flex, same (Section §4.3)

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

– Flex-Rigid, cyclic (Section §4.4)

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

Pruning phase

– Structural rules (Section §5)

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

$$\frac{\Gamma \vdash g_1 :> f_1 \Rightarrow u_1; \sigma_1 \dashv \Delta_1 \qquad \Delta_1 \vdash g_2[\sigma_1] :> f_2 \Rightarrow u_2; \sigma_2 \dashv \Delta_2}{\Gamma \vdash g_1, g_2 :> f_1 + f_2 \Rightarrow u_1[\sigma_2], u_2; \sigma_1[\sigma_2] \dashv \Delta_2}\text{P-Split}$$

– Rigid (Section §5.1)

$$\frac{\Gamma \vdash g :> \mathcal{L}f^o \Rightarrow u; \sigma \dashv \Delta \qquad s_{|f} \Rightarrow \underline{s'}}{\Gamma \vdash o(g; s) :> N(f) \Rightarrow o(u; s'); \sigma \dashv \Delta}\text{P-Rig} \qquad \frac{s_{|f} \Rightarrow !}{\Gamma \vdash o(g; s) :> N(f) \Rightarrow !; ! \dashv \bot}$$

– Flex (Section §5.2)

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

Figure 1. Summary of the rules

General notations

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 $f[g] = g^* \circ f$. We denote by $\mathcal{L}$ the left adjoint $\mathcal{L} : \mathscr{B} \to Kl_T$ which is the identity on objects and postcomposes any morphism $A \to B$ by $\eta_B : B \to TB$.

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 [4].

Remark 2.2. 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 $\hom_{\mathbb{F}_m}(n, m)$.

Remark 2.3. The restriction of the monad T to functors preserving finite connected limits is used to justify computation of a new arity. Consider indeed the

unification problem $M(x,y) = M(y,x)$, in the example of pure λ-calculus. We can design[2] 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 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.

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.*

Proof. 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 case of coproducts follows from connected limits commuting with coproducts in Set. The case of filtered colimits follows from finite limits commuting with filtered colimits in Set.

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.

Proof. We prove the second item. $\mathscr{C}$ is the category of models of a limit sketch, and thus is locally presentable, by [1, Proposition 1.51]. As a result, it is bicomplete [1, Remark 1.56].

Remark 2.6. We need those cocompleteness properties so that we can compute free monads of a finitary endofunctor as the colimit of an initial chain.

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 2.7. $\mathscr{D}^+$ is equivalent to the category of finite families of objects of $\mathcal{A}$. Thinking of objects of $\mathcal{A}$ as metavariable arities (Remark 2.2), $\mathscr{D}^+$ can be thought of as the category of metavariable contexts, motivating the following notation.

Notation 2.2. *We denote an object $\coprod_{i \in \{M,N,\dots\}} Ka_i$ of $\mathscr{D}^+$ by the* context $M : a_M, N : a_N, \dots$

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

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

Property 2.8. 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 $A_j \to \coprod_i A_i$.

Proof. (i)-(ii) Because K is fully faithful, an equivalent statement is that $\mathscr{D}$ is closed under finite connected colimits and K preserves them. Now, we assumed that $\mathcal{A}$ has finite connected limits. Let us show that the Yoneda embedding preserves them. We have a natural isomorphism $[\mathcal{A}, \mathrm{Set}](ya, JX) \simeq \mathcal{C}(Ka, X)$, where y is the Yoneda embedding in $\mathcal{A}^o \to [\mathcal{A}, \mathrm{Set}]$, and $J : \mathcal{C} \to [\mathcal{A}, \mathrm{Set}]$ the canonical embedding. Now consider a finite connected limit $\lim F$ in $\mathcal{A}$. Then,

$$\begin{aligned}
\mathcal{C}(K \lim F, X) &\cong [\mathcal{A}, \mathrm{Set}](y \lim F, JX) \\
&\cong JX(\lim F) && \text{(By the Yoneda Lemma.)} \\
&\cong \lim(JX \circ F) && \text{(By definition of } \mathcal{C}) \\
&\cong \lim([\mathcal{A}, \mathrm{Set}](yF-, JX)] && \text{(By the Yoneda Lemma)} \\
&\cong \lim \mathcal{C}(KF-, X) \\
&\cong \mathcal{C}(\mathrm{colim} KF, X) && \text{(By left continuity of the hom-set bifunctor)}
\end{aligned}$$

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

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

We conclude by Property 2.8.(ii), because all morphisms in $\mathscr{D} = \mathcal{A}^o$ are epimorphic by assumption.

(iv) This follows from Lemma 2.4, 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 2.4), and representable functors being connected.

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 $\mathscr{C}$.*

Proof. F is finitary because filtered colimits commute with finite limits and colimits. It restricts as stated because finite connected limits commute with coproducts and limits.

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 $\mathscr{C}$.*

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 2.8.(v).

We now abstract this situation by stating the properties that we will need.

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

Let $\Gamma \in \mathscr{D}^+$ and $b \in \mathscr{D}$. Given $f \in \hom_{\mathscr{D}}(a, a_i)$, we denote the morphism $Ka \xrightarrow{\mathcal{L}Kf} Ka_i \xrightarrow{in_M} \Gamma, M : b$ by $M_i(f) \in \hom_{Kl_T}(Ka, (\Gamma, M : b)) = \hom_{\mathscr{C}}(Ka, T(\Gamma, M : b))$.

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:

- $M_i(f)$ for some unique i and $f : a \to a_i$,
- $o(f; s)$ for some unique $o \in O$, $f : a^o \to T\Gamma$ 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 $g : \Gamma \to T\Delta$. Then, for any $o \in O$, $f : a^o \to T\Gamma$ and $s : Ka \to S_o$, we have $o(f; s)[g] = o(f[g]; s)$, and for any $1 \leq i \leq n$, $x : b \to a_i$, we have $M_i(x)[g] = g_i \circ Kx$.

Lemma 2.15. *Moreover, for any $u : b \to a$,*

$$o(f; s) \circ Ku = o(f \circ u^o; s \circ Ku)$$

We end this section by introducing notations 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 Section §8. The shape of metavariable arities determine the objects of $\mathcal{A}$, as hinted by Remark 2.2.

	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_T selecting objects in $\mathscr{D}^+$ 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.*

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 $\mathscr{B}^*$.*

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

This lemma allows us to work with true coequalisers in Kl_T^*. 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_T^* is not the terminal object, it is also a colimit in Kl_T;
2. existing colimits in Kl_T are also colimits in Kl_T^*;
3. in particular, coproducts in Kl_T (which are computed in $\mathscr{C}$) are also coproducts in Kl_T^*.

Notation 3.1. *We denote by $\bot$ the freely added terminal object in $\mathscr{B}^*$. Recall that $!$ denote any terminal morphism.*

Here is our main result.

Theorem 3.4. *Let $Kl^*_{\mathscr{D}^+}$ be the full subcategory of Kl_T^* consisting of objects of $\mathscr{D}^+ \cup \{\bot\}$. Then, $Kl^*_{\mathscr{D}^+}$ has coequalisers and the inclusion $Kl^*_{\mathscr{D}^+} \to Kl_T^*$ preserves them.*

In other words, for any coequaliser diagram $A \rightrightarrows TB$ in Kl_T where A and B are in $\mathscr{D}^+$, either there is no cocone, either there is a coequaliser $B \to TC$, with $C \in \mathscr{D}^+$.

4 Unification phase

In this section, we describe the main unification phase, whose goal is to compute a colimit in Kl_T^* of a coequaliser diagram chosen in $Kl_{\mathscr{D}^+}$.

Notation 4.1. *We denote a coequaliser $A \underset{u}{\overset{t}{\rightrightarrows}} B - \overset{\sigma}{-\!\!\succ} C$ in a category $\mathscr{B}$ by $B \vdash t =_{\mathscr{B}} u \Rightarrow \sigma \dashv C$, sometimes even omitting $\mathscr{B}$. When $\mathscr{B} = Kl_T^*$, we moreover implicitly assume that $A \in \mathscr{D}^+$ and $B, C \in \mathscr{D}^+ \cup \{\bot\}$.*

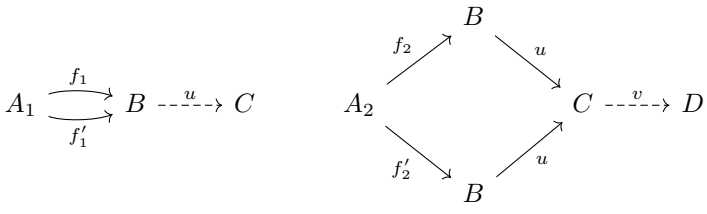
Let us start with simple cases. When $\Gamma = \bot$, the coequaliser is the terminal cocone, i.e., $\bot \vdash t = u \Rightarrow ! \dashv \bot$ holds. When the coproduct is empty, the coequaliser is just Γ, i.e., $\Gamma \vdash () = () \Rightarrow 1_\Gamma \dashv \Gamma$ holds.

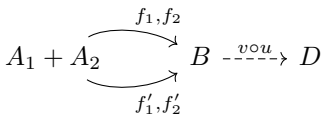
Furthermore, when the coproduct is neither empty nor a singleton, the coequaliser can be computed sequentially thanks to the following general lemma.

Lemma 4.1 (Theorem 9, [7]). *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





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 $o(f; s)$ and $o'(f'; s')$ for some $o, o' \in O$, morphisms $f : a^o \to T\Gamma$, $f' : a^{o'} \to T\Gamma$, and morphisms $s : Ka \to S_o$ and $s' : Ka \to S_{o'}$.

Assume given a unifier, i.e., a Kleisli morphism $\sigma : \Gamma \to T\Delta$ such that $t[\sigma] = u[\sigma]$. By Property 2.14, this entails $o(f[\sigma]; s) = o'(f'[\sigma]; s')$. By Property 2.13, this implies that $o = o'$, $f[\sigma] = f'[\sigma]$, and $s = s'$.

Therefore, we get the following failing rules

$$\frac{o \neq o'}{\Gamma \vdash o(f; s) = o'(f'; s') \Rightarrow ! \dashv \bot} \qquad \frac{s \neq s'}{\Gamma \vdash o(f; s) = o(f'; s') \Rightarrow ! \dashv \bot}$$

We now assume $o = o'$ and $s = s'$. Then, σ unifies t and u if and only if it unifies f and f'. This induces an isomorphism between the category of unifiers for t and u and the category of unifiers for f and g. We therefore get the rule

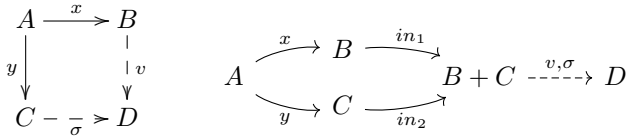
$$\frac{\Gamma \vdash f = g \Rightarrow \sigma \dashv \Delta}{\Gamma \vdash o(f; s) = o(g; s) \Rightarrow \sigma \dashv \Delta}\text{U-RigRig}$$

4.2 Flex-*, no cycle

Here we want to unify $M(f) = \mathcal{L}Kf[in_M]$ for some $f \in \hom_{\mathscr{D}}(a, b)$ 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 $x = \mathcal{L}Kf$ and $y = u'$.

Lemma 4.2 ([2], Exercise 2.17.1). *In any category, if the below left diagram is a pushout, then the below right diagram is a coequaliser.*

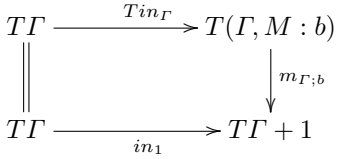


Therefore, it is enough to compute the above left pushout, with $x = \mathcal{L}Kf$ and $y = u'$. Anticipating the pruning phase described in Section §5, this is expressed by the statement $\Gamma \vdash u' :> M(f) \Rightarrow v; \sigma \dashv \Delta$. Therefore, we have the rule

$$\frac{\Gamma \vdash u' :> M(f) \Rightarrow v; \sigma \dashv \Delta \qquad u = T in_\Gamma \circ u'}{\Gamma, M : b \vdash M(f) = u \Rightarrow \sigma, M \mapsto v \dashv \Delta} \tag{6}$$

Let us make the factorisation assumption about u more effective. Indeed, 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 4.3. *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.*



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 4.2. *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 4.4. *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 6 as follows.

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

4.3 Flex-Flex, same metavariable

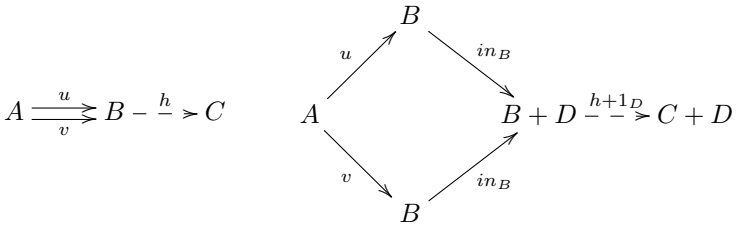
Here we want to unify $M(f) = \mathcal{L}Kf[in_M]$ and $M(g) = \mathcal{L}Kg[in_M]$, with $f, g \in \hom_{\mathscr{D}}(a, b)$.

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

Lemma 4.5. *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.



Therefore, it is enough to compute the coequaliser of $\mathcal{L}Kf$ and $\mathcal{L}Kg$. Since $\mathcal{L}$ is left adjoint (and thus preserves coequalisers) and K preserves coequalisers (Property 2.8.(ii)), we finally get the rule

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

Note that such a coequaliser always exists by Property 2.8.(i).

4.4 Flex-rigid, cyclic

Here, we want to unify $M(f)$ for some $f \in \hom_{\mathscr{D}}(a, b)$ 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 4.4, we thus have the rule

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

5 Pruning phase

The pruning phase corresponds to computing a pushout diagram in Kl_T^* where one branch is a finite coproduct of free morphisms.

Notation 5.1. *We denote a pushout* $\begin{array}{ccc} A & \xrightarrow{f} & B \\ {\scriptstyle g}\downarrow & & \downarrow{\scriptstyle u} \\ C & \xrightarrow[\sigma]{} & D \end{array}$ *in a category $\mathscr{B}$ by $B \vdash_{\mathscr{B}} g :> f \Rightarrow u; \sigma \dashv C$, sometimes even omitting $\mathscr{B}$. When $\mathscr{B} = Kl_T^*$ we moreover implicitly assume that $C, D \in \mathscr{D}^+ \cup \{\bot\}$ and $f = \coprod_{i \in I} \mathcal{L}Kf_i' : \coprod_i Ka_i \to \coprod_i Kb_i$ for some finite set I and morphisms $f_i' : a_i \to b_i$.*

Remark 5.1. For the intuition behind the notation and the relation to the so-called pruning process, consider the case of λ-calculus, as in the introduction. A span $\Gamma \xleftarrow{g} Kn \xrightarrow{\mathcal{L}Kf} Km$ corresponds to a term in $t \in T\Gamma_n$ and a choice of distinct m variables in $\{0, \ldots, n-1\}$, that is, an injection $f : m \to n$. The pushout, if it exists, consists in "coercing" (hence the symbol $:>$) the term t to live in $T\Gamma_m$, by restricting the arity of the metavariables according to $\sigma : \Gamma \to T\Delta$. The resulting term $u \in \hom(Kn, T\Delta) \cong T\Delta_n$ is t but living in the "smaller" context $\{0, \ldots, m-1\}$ in the restricted metavariable context Δ.

Remark 5.2. A cocone consists in morphisms $\coprod_i Kb_i \xrightarrow{u} T\Delta \xleftarrow{\sigma} \Gamma$ such that $g[\sigma] = u \circ \coprod_i \mathcal{L}Kf_i$, i.e., for all $i \in I$, we have $g_i[\sigma] = u_i \circ Kf_i$.

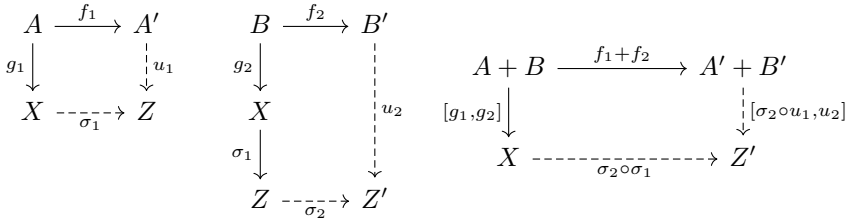
Let us start with simple cases. When $\Gamma = \bot$, the pushout is the terminal cocone, i.e., $\bot \vdash t :> f \Rightarrow !;! \dashv \bot$ holds. . When the coproduct is empty, the pushout is just Γ, i.e., $\Gamma \vdash () :> () \Rightarrow (); 1_\Gamma \dashv \Gamma$ holds.

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

Lemma 5.3. *In any category, denoting morphism composition $f \circ g$ by $g[f]$, the following rule applies.*

$$\frac{\Gamma \vdash g_1 :> f_1 \Rightarrow u_1; \sigma_1 \dashv \Delta_1 \qquad \Delta_1 \vdash g_2[\sigma_1] :> f_2 \Rightarrow u_2; \sigma_2 \dashv \Delta_2}{\Gamma \vdash g_1, g_2 :> f_1 + f_2 \Rightarrow u_1[\sigma_2], u_2; \sigma_1[\sigma_2] \dashv \Delta_2} \text{P-SPLIT}$$

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



We can focus on the case where the coproduct is the singleton (since we focus on finite coproducts of elements of $\mathscr{D}$) and $\Gamma \neq \bot$. Thus, we want to compute the pushout of $T\Gamma \xleftarrow{} Ka \xrightarrow{N(f)} T(N : b)$ 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 $\Gamma \xleftarrow{o(g;s)} Ka \xrightarrow{N(f)} N : b$ where $g : a^o \to T\Gamma$ and $s : Ka \to S_o$. By Remark 5.2, a cocone in Kl_T is given by an

object Δ with morphisms $Kb \xrightarrow{u} T\Delta \xleftarrow{\sigma} \Gamma$ such that $o(g;s)[\sigma] = u \circ Kf$. By Property 2.14, this means that $o(g[\sigma];s) = u \circ Kf$. Now, by Property 2.13, u is either some $M'(f')$ or $o'(g';s')$, with $g' : b^o \to T\Delta$ and $s' : Kb \to S_{o'}$. But in the first case, $u \circ Kf = M'(f') \circ Kf = M'(f' \circ f)$ so it cannot equal $o(g[\sigma];s)$, by Property 2.13. So we are in the second case, and again by Property 2.13, $o = o'$, $g[\sigma] = g' \circ f^o$ and $s = s' \circ Kf$.

Remark 5.4. 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 Kf is epimorphic by Property 2.8.(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 5.2. *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

$$\frac{\Gamma \vdash g :> \mathcal{L}f^o \Rightarrow u; \sigma \dashv \Delta \qquad s_{|f} \Rightarrow \underline{s'}}{\Gamma \vdash o(g;s) :> N(f) \Rightarrow o(u;s'); \sigma \dashv \Delta}\text{P-Rig} \qquad \frac{s_{|f} \Rightarrow !}{\Gamma \vdash o(g;s) :> N(f) \Rightarrow !;! \dashv \bot}$$

Remark 5.5. If S_o is orthogonal [1, Definition 1.32] to all morphisms (i.e., given any span $S_o \leftarrow a \to b$, there exists a unique $b \to S_o$ completing the triangle), as in the case where S_o is the output type "dirac", then this rule never applies.

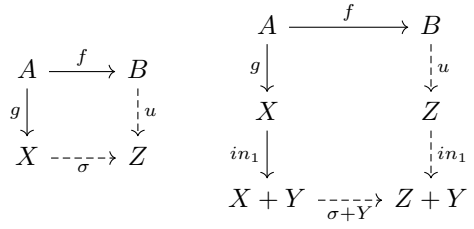
5.2 Flex

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

Lemma 5.6. *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.



Since $\mathcal{L}$ is left adjoint (and thus preserves pushouts) and K preserves pushouts (Property 2.8.(ii)), the pushout can be computed in $\mathscr{D}$ (it exists by Property 2.8.(i)). Therefore, we get the rule

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

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 6.8). 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 $|M(f)| = 0$ and $|o(g; s)| = 1 + |g|$, with $|g| = \sum_i g_i$, for any $g : \coprod_i Ka_i \to T\Gamma$.*

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

Remark 6.3. 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 2.8.(v), since $\mathbb{N}$ is the coproduct $\coprod_{n\in\mathbb{N}} 1$), where $T\Gamma \to \mathbb{N}$ is defined as 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 6.4. *Assume given, for each object a of $\mathcal{A}$, a predicate P_a on $\hom(Ka, T\Gamma)$ such that*

- *$P_a(M(f))$ holds for any $M : b \in \Gamma$ and $f \in \hom_{\mathscr{D}}(a, b)$;*

– $P_a(o(g;s))$ *holds for any* $o \in O$, $s : Ka \to S_o$, *and* $g : \coprod_{j\in J_o} KL_{o,j}a \to S_o$ *such that* $P_{L_{o,j}a}(g_j)$ *holds for every* $j \in J_o$.

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

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

Lemma 6.5. *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 6.6. *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 6.7. *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* $|\mathcal{L}Kf[u]| = |u|$.

Corollary 6.8. *If there is a commuting square in* Kl_T

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

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

Proof. Since $t[\sigma, u] = \mathcal{L}Kf[u]$, we have $|t[\sigma, u]| = |\mathcal{L}Kf[u]|$. Corollary 6.7 implies $|u| \geq |t| + |u| \times |t|_M$. Therefore, either $|t|_M = 0$ and we conclude by Lemma 6.5, either $|t|_M = 1$ and $|t| = 0$ and so t is $M(g)$ for some g.

7 Completeness

Each inductive rule presented so far provides an elementary step for the construction of coequalisers. We need to ensure that this set of rules allows to construct a coequaliser in a finite number of steps. To make the argument more straightforward, we slightly alter the splitting rules U-Split et P-Split (see figure 1) by enforcing that the domain coproduct $A_1 + \cdots + A_n$ of objects of $\mathscr{D}$ is split into A_1 and $A_2 + \cdots + A_n$ in the premises. 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 the 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 2.7). We extend this definition to the case where $\Gamma = \bot$, by taking $|\bot| = 0$. We also define the size[3] $||t||$ of a *term* $t : Ka \to T\Gamma$ recursively by $||M(f)|| = 1$ and $||o(f; s)|| = 1 + ||f||$, 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 (see Figure 1).

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 of the unification phase (see Figure 1), 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.

Lemma 7.3. *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 2.2.

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 [3]). 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.

[3] 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.

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 $\{0, \dots, p-1\}$ 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-calculs 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 rule U-FlexFlex and P-Flex in Figure 1) 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 in Figure 1, 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 8.1. *Assume an endofunctor for syntax as in Section §2.2. If* $\Gamma \vdash t = u \Rightarrow \sigma \dashv \Delta$ *or* $\Gamma \vdash t :> \coprod_i \mathcal{L}Kf_i \Rightarrow u; \sigma \dashv \Delta$*, then* Δ *is a coproduct of pure arities whenever* Γ *is, and* $\Delta \neq \bot$*.*

8.4 Linear syntax

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 [8].

Each typing rule provides an operation

$$\begin{array}{ll} \text{Variable} & 1 \vdash * \qquad\qquad y1 \\ \text{Application} & \dfrac{n \vdash t \qquad m \vdash u}{n+m \vdash t\ u}\ X_n \times X_m \times y_{n+m} \end{array}$$

$$\Gamma$$

8.5 Intrinsic polymorphic syntax

We present intrinsic system F, following [5]. Let $S : \mathbb{F}_m \to \mathrm{Set}$ mapping n to the set Sn of types for system F taking free type variables in $\{0, \ldots, n-1\}$. Intuitively, a metavariable arity $n; \sigma_1, \ldots, \sigma_p \vdash \tau$ specifies the number n of free type variables, the list of input types $\vec{\sigma}$, and the output type τ, all living in S_n. Now, consider the category $\mathcal{A}$ of metavariable arities where a morphism between $n, \Gamma \vdash \tau$ and $n', \Gamma' \vdash \tau'$ is a morphism $\sigma : n \to n'$ in $\mathbb{F}_m$ such that $\tau[\sigma] = \tau'$, and a renaming $\Gamma[\sigma] \to \Gamma'$. More formally, $\mathcal{A}$ is the op-lax colimit of $n \mapsto \mathbb{F}_m[S_n] \times S_n$. The intrinsic syntax of system F can then be defined by a suitable endofunctor, see [5]. The basic idea is that a typing rule

$$\frac{m_1|\Gamma_1 \vdash t_1 : \tau_1 \quad \ldots \quad m_n|\Gamma_n \vdash t_n : \tau_n}{m|\Gamma \vdash C(t_1, \ldots t_n) : C'(\tau_1, \ldots, \tau_n)}$$

is interpreted as a rule $F(X)_{m|\Gamma\vdash\tau} = X_{m_1|\Gamma_1\vdash\tau_1} \times \cdots \times X_{m_n|\Gamma_n\vdash\tau_n} \times y(m|\Gamma \vdash C'(\tau_1, \ldots, \tau_n))(m|\Gamma \vdash \tau)$

Operations	Typing rule	$o(?, s)$	$o(f, ?)$
Type abstraction	$\dfrac{m+1\|wk(\Gamma) \vdash t : \tau}{m\|\Gamma \vdash \Lambda t : \Lambda\tau}$	$m+1\|wk(\Gamma) \vdash t : \tau'$	$\delta_{\tau,\Lambda\tau'}$
	$\dfrac{m\|\Gamma \vdash t : \Lambda\tau_1 \quad \tau_2 \in S_m}{m\|\Gamma \vdash t \cdot \tau_2 : \tau_1[\tau_2]}$		

$$F(X)_{n|\Gamma\vdash\tau} = \coprod_{\tau'} X(n+1|\Gamma \vdash \tau')\delta_{\tau,\forall\tau'}$$

problem! $\delta_{-,\forall\tau'}$ does not preserve connected limits.

variables:Here, V is defined as $V(n; \Gamma \vdash \sigma) = \#\{\sigma \in \Gamma\}$.

$$F(X) = \coprod_{m,\tau\in S_m} y(m|\tau \vdash \tau)$$

$$
\begin{aligned}
F(X)_{n;\Gamma\vdash\sigma} &= V_{n;\Gamma\vdash\sigma} && \text{(variables)}\\
&+ \coprod_{\tau} X(n+1;\Gamma\vdash\tau)\delta_{\sigma,\forall\tau} && \text{(type abstraction)}\\
&+ \coprod_{\tau',\sigma'} X(n;\Gamma\vdash\forall\tau')\delta_{\sigma,\tau'[\sigma']} && \text{(type application)}\\
&+ \coprod_{\tau,\sigma'} X(n;\Gamma,\tau\vdash\sigma')\delta_{\sigma,\tau\Rightarrow\sigma'} && \text{(abstraction)}\\
&+ \coprod_{\tau'} X(n;\Gamma\vdash\tau'\Rightarrow\sigma)\times X(n;\Gamma\vdash\tau') && \text{(application)}
\end{aligned}
$$

But here are the problems: $\delta_{\underline{\tau},\forall\tau}$, $\delta_{\underline{\tau},\tau'[\sigma']}$, $\delta_{\underline{\tau},\tau\Rightarrow\sigma'}$ do not define a functor $\mathcal{A}\to$ Set. One solution: do not take the op-lax colimit, but merely the coproduct. Disadvantage: metavariables can live only in a context with an exact number of type variables (specified in the arity). Or maybe a variant where metavariables do not take any bound type variables as arguments?

For $\delta_{\underline{\tau},\forall\sigma}$,we could replace $\coprod_\tau X(n+1;\Gamma\vdash\tau)\delta_{\sigma,\forall\tau}$ with something like $\coprod_{p,\tau\in M_{p+1}}\coprod_{(f,r)\in y(p\vdash\forall\tau)} X(\underline{n}+1;\underline{\Gamma}\vdash f\cdot\tau)$ where M_{p+1} is intuitively $S_{n+1}\backslash S_n$, but then we have a kind of dependent sum... And I am not sure this trick would work with type application anyway.

References

1. Adámek, J., Rosicky, J.: Locally Presentable and Accessible Categories. Cambridge University Press (1994). https://doi.org/10.1017/CBO9780511600579
2. 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
3. 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)
4. Gabbay, M.J., Pitts, A.M.: A new approach to abstract syntax involving binders. In: Proc. 14th Symposium on Logic in Computer Science IEEE (1999)
5. Hamana, M.: Polymorphic abstract syntax via grothendieck construction (2011)
6. 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
7. Rydeheard, D.E., Burstall, R.M.: Computational category theory. Prentice Hall International Series in Computer Science, Prentice Hall (1988)
8. Tanaka, M.: Abstract syntax and variable binding for linear binders. In: MFCS '00. LNCS, vol. 1893. Springer (2000)
9. Vezzosi, A., Abel, A.: A categorical perspective on pattern unification. RISC-Linz p. 69 (2014)