Generic pattern unification

We provide a generic second-order unification algorithm for Miller's pattern fragment, implemented in Agda. The syntax with metavariables is parameterised by a notion of signature generalising binding signatures, covering ordered λ-calculus, or (intrinsic) polymorphic syntax such as System F. The correctness of the algorithm is stated and proved on papers using a categorical perspective, based on the observation that the most general unifier is an equaliser in a multi-sorted Lawvere theory, thus generalising the case of first-order unification.

ACM Reference Format:
. 2024. Generic pattern unification. *Proc. ACM Program. Lang.* 1, POPL, Article 1 (January 2024), 27 pages.

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 Robinson [1965] is used in ML-style type inference systems and logic programming languages such as Prolog. More advanced type systems, where variable binding is crucially involved, requires second-order unification Huet [1975], which is undecidable Goldfarb [1981]. However, Miller Miller [1991] 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, or computes the most general unifier.

Recent results in type inference, Dunfield-Krishnaswami Dunfield and Krishnaswami [2019], or Jinxu et. al Zhao et al. [2019], include very large proofs: the former comes with a 190 page appendix, and the latter comes with a Coq proof many thousands of lines long -- and both of these results are for tiny kernel calculi. If we ever hope to extend this kind of result to full programming languages like Haskell or OCaml, we must raise the abstraction level of these proofs, so that they are no longer linear (with a large constant) in the size of the calculus. A close examination of these proofs shows that a large part of the problem is that the type inference algorithms make use of unification, and the correctness proofs for type inference end up essentially re-establishing the entire theory of unification for each algorithm. The reason they do this is because algorithmic typing rules essentially give a first-order functional program with no abstractions over (for example) a signature for the unification algorithm to be defined over, or any axiomatic statement of the invariants the algorithmic typing rules had to maintain.

The present work is a first step towards a general solution to this problem. Our generic unification algorithm implemented in Agda is parameterised by a new notion of signature for syntax with metavariables, whose scope goes beyond the standard binding signatures. One important feature is that the notion of contexts is customisable, making it possible to cover simply-typed second-order syntax, ordered syntax, or (intrinsic) polymorphic syntax such as System F. We focused on Miller's pattern unification, as this is already a step beyond the above-cited works Dunfield and Krishnaswami [2019]; Zhao et al. [2019] that use plain first-order unification. Moreover, this is necessary for types with binders (e.g., fixed-point operators like $\mu a.A[a]$) as well as for rich type systems like dependent types.

Author's address:

2024. 2475-1421/2024/1-ART1 $15.00
https://doi.org/

Plan of the paper

In section §2, we present our generic pattern unification algorithm, parameterised by our generalised notion of binding signature. We introduce categorical semantics of pattern unification in Section §3. We show correctness of the two phases of the unification algorithm in Section §4 and Section §5. Termination and completeness are justified in Sections §6 and 7. We present some examples of signatures in Section §8. Related work is discussed in Section §9.

General notations

Given a list $\vec{x} = (x_1, \dots, x_n)$ and a list of positions $\vec{p} = (p_1, \dots, p_m)$ taken in $\{1, \dots, n\}$, we denote $(x_{p_1}, \dots, x_{p_m})$ by $x_{\vec{p}}$.

Given a category $\mathscr{B}$, we denote its opposite category by $\mathscr{B}^{op}$. If a and b are two objects of $\mathscr{B}$, we denote the set of morphisms between a and b by $\hom_{\mathscr{B}}(a, b)$. We denote the identity morphism at an object x by 1_x. We denote the coproduct of two objects A and B by $A + B$ and the coproduct of a family of objects $(A_i)_{i \in I}$ by $\coprod_{i \in I} A_i$, and similarly for morphisms. If $f : A \to B$ and $g : A' \to B$, we denote the induced morphism $A + A' \to B$ by f, g. Coproduct injections $A_i \to \coprod_{i \in I} A_i$ are typically denoted by in_i. Let T be a monad on a category $\mathscr{B}$. We denote its unit by η, and its Kleisli category by Kl_T: the objects are the same as those of $\mathscr{B}$, and a Kleisli morphism from A to B is a morphism $A \to TB$ in $\mathscr{B}$. We denote the Kleisli composition of $f : A \to TB$ and $g : B \to TC$ by $f[g] : A \to TC$.

2 PRESENTATION OF THE ALGORITHM

In this section, we start by describing a pattern unification algorithm for pure λ-calculus, summarised in Figure 1. Then we present our generic algorithm (Figure 2).

We show the most relevant parts of the Agda code; the interested reader can check the full implementation in the supplemental material. We tend to use Agda as a programming language rather than as a theorem prover. This means that the definitions of our data structures typically do not mention the properties (such as associativity for a category), and we leave for future work the task of mechanising the correctness proof of the algorithm, by investigating the formalisation of various concepts from category theory – a notorious challenge on its own – on which our proof relies on. Furthermore, we are not reluctant to using logically inconsistent features to make programming easier: the type hierachy is collapsed and the termination checker is disabled. Despite those caveats, dependent types are still helpful in guiding the implementation, contrary to a preliminary ocaml version where the code was much less constrained by the typing discipline.

2.1 An example: pure λ-calculus.

Consider the syntax of pure λ-calculus extended with metavariables satisfying the pattern restriction. We list the Agda code in Figure 3, together with a more standard presenatation based on inductive rules generating the syntax. We write $\Gamma; n \vdash t$ to mean t is a wellformed λ-term in the context $\Gamma; n$, consisting of two parts:

(1) a metavariable context $\Gamma = (M_1 : m_1, \dots, M_p : m_p)$, specifying metavariable symbols M_i together with their arities, i.e, their number of arguments m_i, and
(2) a variable context, which is a mere natural number indicating the highest possible free variable.

Free variables are indexed from 1 and we use the De Bruijn level convention: the variable bound in $\Gamma; n \vdash \lambda t$ is $n + 1$, not 0, as it would be using De Bruijn indices De Bruijn [1972]. In Agda, variables

Fig. 1. Pattern unification for λ-calculus (Section §2.1)

```
prune {Γ} (M ( x )) y =
  let p , r , l = commonValues x y in
    ⌊ Γ [ M : p ] ◂ (M : p) ( l ) , M ↦-( r ) ⌋
```

$$\frac{m \vdash x :> y \Rightarrow l; r \dashv p}{\begin{array}{c}\Gamma[M : m] \vdash M(x) :> y \Rightarrow \\ P(l); M \mapsto P(r) \dashv \Gamma[P : p]\end{array}}\textsc{P-Flex}$$

```
prune (App t u) x = do
        Δ₁ ◂ t' , σ₁ ← prune t x
        Δ₂ ◂ u' , σ₂ ← prune (u [ σ₁ ]t) x
        ⌊ Δ₂ ◂ App (t' [ σ₂ ]t) u' , σ₁ [ σ₂ ]s ⌋
```

$$\frac{\begin{array}{c}\Gamma \vdash t :> x \Rightarrow t'; \sigma_1 \dashv \Delta_1 \\ \Delta_1 \vdash u[\sigma_1] :> x \Rightarrow u'; \sigma_2 \dashv \Delta_2\end{array}}{\Gamma \vdash t\ u :> x \Rightarrow t'[\sigma_2]\ u'; \sigma_1[\sigma_2] \dashv \Delta_2}$$

```
prune (Lam t) x = do
        Δ ◂ t' , σ ← prune t (x ↑)
        ⌊ Δ ◂ Lam t' , σ ⌋
```

$$\frac{\Gamma \vdash t :> x\uparrow \Rightarrow t'; \sigma \dashv \Delta}{\Gamma \vdash \lambda t :> x \Rightarrow \lambda t'; \sigma \dashv \Delta}$$

```
prune {Γ} (Var i) x with i { x }⁻¹
... | ⊥ = ⊥
... | ⌊ j ⌋ = ⌊ Γ ◂ Var j , idₛ ⌋
```

$$\frac{i = x_j}{\Gamma \vdash \underline{i} :> x \Rightarrow \underline{j}; 1_\Gamma \dashv \Gamma}$$

$$\frac{i \notin x}{\Gamma \vdash \underline{i} :> x \Rightarrow !; ! \dashv \bot}$$

```
unify-flex-* {Γ} M x t with occur-check M t
... | Same-MVar y =
  let p , z = commonPositions x y in
  ⌊ Γ [ M : p ] ◂ M ↦-( z ) ⌋
... | Cycle = ⊥
... | No-Cycle t' = do
    Δ ◂ u , σ ← prune t' x
    ⌊ Δ ◂ M ↦ u , σ ⌋
```

$$\frac{m \vdash x = y \Rightarrow z \dashv p}{\begin{array}{c}\Gamma[M : m] \vdash M(x) = M(y) \Rightarrow \\ M \mapsto P(z) \dashv \Gamma[P : p]\end{array}}\textsc{Same-MVar}$$

$$\frac{M \in u \qquad u \neq M(\dots)}{\Gamma, M : m \vdash M(x) = t \Rightarrow ! \dashv \bot}\textsc{Cycle}$$

$$\frac{M \notin t \quad \Gamma\backslash M \vdash t :> x \Rightarrow u; \sigma \dashv \Delta}{\Gamma \vdash M(x) = t \Rightarrow \sigma, M \mapsto u \dashv \Delta}\textsc{No-cycle}$$

```
unify t (M ( x )) = unify-flex-* M x t
unify (M ( x )) t = unify-flex-* M x t
```

(+ symmetric rules)

```
unify (App t u) (App t' u') = do
  Δ₁ ◂ σ₁ ← unify t t'
  Δ₂ ◂ σ₂ ← unify (u [ σ₁ ]t) (u' [ σ₁ ]t)
  ⌊ Δ₂ ◂ σ₁ [ σ₂ ]s ⌋
```

$$\frac{\begin{array}{c}\Gamma \vdash t = t' \Rightarrow \sigma_1 \dashv \Delta_1 \\ \Gamma \vdash u[\sigma_1] = u'[\sigma_2] \Rightarrow \sigma_2 \dashv \Delta_2\end{array}}{\Gamma \vdash t\ u = t'\ u' \Rightarrow \sigma_1[\sigma_2] \dashv \Delta_2}$$

```
unify (Lam t) (Lam t') = unify t t'
```

$$\frac{\Gamma \vdash t = t' \Rightarrow \sigma \dashv \Delta}{\Gamma \vdash \lambda t = \lambda t' \Rightarrow \sigma \dashv \Delta}$$

```
unify {Γ} (Var i) (Var j) with i Fin.≟ j
... | no _ = ⊥
... | yes _ = ⌊ Γ ◂ idₛ ⌋
```

$$\frac{i \neq j}{\Gamma \vdash \underline{i} = \underline{j} \Rightarrow ! \dashv \bot} \qquad \frac{}{\Gamma \vdash \underline{i} = \underline{i} \Rightarrow 1_\Gamma \dashv \Gamma}$$

```
unify _ _ = ⊥
```

$$\frac{o \neq o' \ (\textit{rigid}\text{ term constructors})}{\Gamma \vdash o(\vec{t}) = o'(\vec{t'}) \Rightarrow ! \dashv \bot}$$

Failure propagation (do notation) $\quad \bot \vdash t :> x \Rightarrow !; ! \dashv \bot \quad \bot \vdash t = u \Rightarrow ! \dashv \bot$

Fig. 2. Our generic pattern unification algorithm

```
prune {Γ} (M ( x )) y =
  let p , r , l = pullback x y in
  ⌊ Γ [ M :  p ] ◂ (M :  p) ( l ) , M ↦-( r ) ⌋
```

Same as the rule P-Flex in Figure 1.

```
prune (Rigid o δ) x with o { x }⁻¹
... | ⊥ = ⊥
... | ⌊ o’ ⌋ = do
      Δ ◂ δ’ „ σ ← prune-σ δ (x ^ o’)
      ⌊ Δ ◂ Rigid o’ δ’ , σ ⌋
```

$$\frac{o \neq \ldots\{x\}}{\Gamma \vdash o(\delta) :> x \Rightarrow !;! \dashv \bot}\textsc{P-Rig-Fail}$$

$$\frac{\Gamma \vdash \delta :> x^{o'} \Rightarrow \delta'; \sigma \dashv \Delta \qquad o = o'\{x\}}{\Gamma \vdash o(\delta) :> x \Rightarrow o'(\delta'); \sigma \dashv \Delta}\textsc{P-Rig}$$

```
prune-σ {Γ} [] [] = ⌊ Γ ◂ [] „ ids ⌋
prune-σ (t , δ) (x₀ :: xs) = do
  Δ₁ ◂ t’ , σ₁ ← prune t x₀
  Δ₂ ◂ δ’ „ σ₂ ← prune-σ (δ [ σ₁ ]s) xs
  ⌊ Δ₂ ◂ (t’ [ σ₂ ]t , δ’) „ (σ₁ [ σ₂ ]s) ⌋
```

$$\frac{}{\Gamma \vdash () :> () \Rightarrow (); 1_\Gamma \dashv \Gamma}\textsc{P-Empty}$$

$$\frac{\begin{array}{c}\Gamma \vdash t :> x_0 \Rightarrow t'; \sigma_1 \dashv \Delta_1 \\ \Delta_1 \vdash \delta[\sigma_1] :> x \Rightarrow \delta'; \sigma_2 \dashv \Delta_2\end{array}}{\begin{array}{c}\Gamma \vdash t, \delta :> x_0, x \Rightarrow \\ t'[\sigma_2], \delta'; \sigma_1[\sigma_2] \dashv \Delta_2\end{array}}\textsc{P-Split}$$

unify-flex-* is defined as in Figure 1, replacing commonPositions with equaliser .

```
unify t (M ( x )) = unify-flex-* M x t
unify (M ( x )) t = unify-flex-* M x t
```

See the rules Same-MVar, Cycle, and No-Cycle in Figure 1.

```
unify (Rigid o δ) (Rigid o’ δ’) with o ≟ o’
... | no _ = ⊥
... | yes ≡.refl = unify-σ δ δ’
```

$$\frac{o \neq o'}{\Gamma \vdash o(\delta) = o'(\delta') \Rightarrow ! \dashv \bot}\textsc{Clash}$$

$$\frac{\Gamma \vdash \delta = \delta' \Rightarrow \sigma \dashv \Delta}{\Gamma \vdash o(\delta) = o(\delta') \Rightarrow \sigma \dashv \Delta}\textsc{U-Rig}$$

```
unify-σ {Γ} [] [] = ⌊ Γ ◂ ids ⌋
unify-σ (t₁ , δ₁) (t₂ , δ₂) = do
  Δ ◂ σ ← unify t₁ t₂
  Δ’ ◂ σ’ ← unify-σ (δ₁ [ σ ]s) (δ₂ [ σ ]s)
  ⌊ Δ’ ◂ σ [ σ’ ]s ⌋
```

$$\frac{}{\Gamma \vdash () = () \Rightarrow 1_\Gamma \dashv \Gamma}\textsc{U-Empty}$$

$$\frac{\begin{array}{c}\Gamma \vdash t_1 = t_2 \Rightarrow \sigma \dashv \Delta \\ \Delta \vdash \delta_1[\sigma] = \delta_2[\sigma] \Rightarrow \sigma' \dashv \Delta'\end{array}}{\Gamma \vdash t_1, \delta_1 = t_2, \delta_2 \Rightarrow \sigma[\sigma'] \dashv \Delta'}\textsc{U-Split}$$

Failure propagation (do notation)

$$\frac{}{\bot \vdash \delta :> x \Rightarrow !;! \dashv \bot}\textsc{U-Fail}$$

$$\frac{}{\bot \vdash \delta = \delta' \Rightarrow ! \dashv \bot}\textsc{P-Fail}$$

in the variable context n consist of elements of Fin n, the type of natural numbers between[1] 1 and n. We also use a nameless encoding of metavariable contexts: they are mere lists of metavariable arities, and metavariables are referred to by their index in the list. Let us focus on the last constructor building a metavariable application in the context $\Gamma; n$. The argument of type $m \in \Gamma$ is an index of any element m in the list Γ. The constructor also takes an argument of type $m \Rightarrow n$, which unfolds

[1] Fin n is actually defined in the standard library as an inductive type designed to be (canonically) isomorphic with $\{0, \ldots, n-1\}$.

```
MetaContext : Set                              _⇒_ : ℕ → ℕ → Set
MetaContext = List ℕ                           m ⇒ n = Vec (Fin n) m

data Tm (Γ : MetaContext) (n : ℕ) : Set where
  Var : Fin n → Tm Γ n
  App : Tm Γ n → Tm Γ n → Tm Γ n
  Lam : Tm Γ (1 + n) → Tm Γ n
  _(_) : ∀ {m} → m ∈ Γ → m ⇒ n → Tm Γ n
```

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

$$\frac{M : m \in \Gamma \qquad \overbrace{x : m \Rightarrow n}^{x_1,\ldots,x_m \in \{1,\ldots,n\} \text{ distinct}}}{\Gamma; n \vdash M(x_1, \ldots, x_m)}$$

Fig. 3. Syntax of λ-calculus (Section §2.1)

```
_∘_ : ∀ {p q r} → (q ⇒ r) → (p ⇒ q) → (p ⇒ r)
xs ∘ [] = []
xs ∘ (y :: ys) = Vec.lookup xs y :: (xs ∘ ys)

_↑ : ∀ {p q} → p ⇒ q → (1 + p) ⇒ (1 + q)
_↑ {p}{q} x = Vec.insert (Vec.map Fin.inject₁ x)
                    (Fin.fromℕ p) (Fin.fromℕ q)

_{_} : ∀ {Γ n p} → Tm Γ n → n ⇒ p → Tm Γ p

App t u { x } = App (t { x }) (u { x })
Lam t { x } = Lam (t { x ↑ })
Var i { x } = Var (i { x })
M ( y ) { x } = M ( x ∘ y )
```

$$\frac{x : q \Rightarrow r \qquad y : p \Rightarrow q}{\underbrace{x \circ y}_{(x_{y_1}, \ldots, x_{y_p})} \quad : p \Rightarrow r}$$

$$\frac{x : p \Rightarrow q}{\underbrace{x \uparrow}_{(x_1,\ldots,x_p,q+1)} \quad : p+1 \Rightarrow q+1}$$

$$\frac{\Gamma; n \vdash t \qquad x : n \Rightarrow p}{\Gamma; p \vdash t\{x\}}$$

Fig. 4. Renaming for λ-calculus (Section §2.1)

as Vec (Fin n) m: this is the type of lists of size m consisting of elements of Fin n, that is, natural numbers between 1 and n. Note this does not fully enforce the pattern restriction: metavariable arguments are not required to be distinct. However, our unification algorithm is guaranteed to produce correct outputs only if this constraint is satisfied in the inputs.

The Agda implementation of metavariable substitution is listed in Figure 5. A *metavariable substitution* $\sigma : \Gamma \longrightarrow \Delta$ assigns to each metavariable M of arity m in Γ a term $\Delta; m \vdash \sigma_M$. In Agda, the type of substitutions between Γ and Δ is defined as VecList (Tm Δ) Γ, where VecList.t X ℓ is (inductively) defined as the product type $X\ a_1 \times \cdots \times X\ a_n$ for any dependent type $X : A \longrightarrow$ Set and list $\ell = [a_1, \ldots, a_n]$ of elements of A.

This assignation extends (through a recursive definition) to any term $\Gamma; n \vdash t$, yielding a term $\Delta; n \vdash t[\sigma]$. The base case is $M(x_1, \ldots, x_m)[\sigma] = \sigma_M\{x\}$, where $-\{x\}$ is variable renaming, defined by recursion (see Figure 4). Renaming a λ-abstraction requires extending the renaming $x : p \Rightarrow q$ to $x \uparrow : p + 1 \Rightarrow q + 1$ to take into account additional the bound variable $\underline{p + 1}$ which is renamed to $\underline{q + 1}$. Then, $(\lambda t)\{x\}$ is defined as $\lambda(t\{x \uparrow\})$.

```
_⟶_ : MetaContext → MetaContext → Set
Γ ⟶ Δ = VecList (Tm Δ) Γ

_[_]t : ∀ {Γ n} → Tm Γ n → ∀ {Δ} → (Γ ⟶ Δ) → Tm Δ n
App t u [ σ ]t = App (t [ σ ]t) (u [ σ ]t)
Lam t [ σ ]t = Lam (t [ σ ]t)
Var i [ σ ]t = Var i
M ( x ) [ σ ]t = VecList.nth M σ { x }
```

$$\boxed{\dfrac{\Gamma; n \vdash t \qquad \sigma : \Gamma \to \Delta}{\Delta; n \vdash t[\sigma]}}$$

```
_[_]s : ∀ {Γ₁ Γ₂ Γ₃} → (Γ₁ ⟶ Γ₂) → (Γ₂ ⟶ Γ₃) → (Γ₁ ⟶ Γ₃)
δ [ σ ]s = VecList.map (λ _ t → t [ σ ]t) δ
```

$$\boxed{\dfrac{\sigma : \Gamma_1 \to \Gamma_2 \qquad \delta : \Gamma_2 \to \Gamma_3}{\underbrace{\sigma[\delta]}_{M \mapsto \sigma_M[\delta]} \; : \Gamma_1 \to \Gamma_3}}$$

Fig. 5. Metavariable substitution for λ-calculus (Section §2.1)

The identity substitution $1_\Gamma : \Gamma \to \Gamma$ is defined by the term $M(1, \dots, m)$ for each metavariable declaration $M : m \in \Gamma$. The composition $\delta[\sigma] : \Gamma_1 \to \Gamma_3$ of two substitutions $\delta : \Gamma_1 \to \Gamma_2$ and $\sigma : \Gamma_2 \to \Gamma_3$ is defined as $M \mapsto \delta_M[\sigma]$.

A *unifier* of two terms $\Gamma; n \vdash t, u$ is a substitution $\sigma : \Gamma \to \Delta$ such that $t[\sigma] = u[\sigma]$. A *most general unifier* of t and u is a unifier $\sigma : \Gamma \to \Delta$ that uniquely factors any other unifier $\delta : \Gamma \to \Delta$', in the sense that there exists a unique $\delta' : \Delta \to \Delta'$ such that $\delta = \sigma[\delta']$. We denote this situation by $\Gamma \vdash t = u \Rightarrow \sigma \dashv \Delta$, leaving the variable context n implicit. Intuitively, the symbol $\Rightarrow$ separates the input and the output of the unification algorithm, which either returns a most general unifier, or fails when there is no unifier at all (for example, when unifying $t_1\ t_2$ with λu). Note that the type signature of our unification algorithm merely enforces that the output is an outgoing substitution:

```
data _⟶? (Γ : MetaContext) : Set where
  _◂_ : ∀ Δ → (Γ ⟶ Δ) → Γ ⟶?

unify : ∀ {Γ n} → Tm Γ n → Tm Γ n → Maybe (Γ ⟶?)
```

Here, Maybe X is an inductive type with an error constructor $\bot$ and a success constructor $\lfloor - \rfloor$ taking as argument an element of type X.

The unification algorithm recursively inspects the structure of the given terms until reaching a metavariable at the top-level, as seen in the second box of Figure 1. We exploit the do notation to propagate failure. For example, the application case fails at the beginning if unify $t\ t'$ does. Otherwise it continues with the success return values $\Delta_1 \blacktriangleleft \sigma_1$. From a mathematical point of view, we will argue that it is more convenient to handle failure by considering a formal error metavariable context[2] $\bot$ 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, as demonstrated in the last case when unifying two different *rigid* term constructors (application, λ-abstraction, or variables). With this extended meaning, the inductive rule for application remains sound, in a sense that will be clarified in Section §3.1. Formally, failure propagation is modelled by the rule $\bot \vdash t = u \Rightarrow\ ! \dashv \bot$.

When reaching a metavariable application $M(x)$ at the top-level of either term, denoting by t the other term, three situations must be considered:

[2] In Section §3.1, we interpret $\bot$ as a terminal object freely added to the category of metavariable contexts and substitutions between them.

(1) t is a metavariable application $M(y)$;
(2) t is not a metavariable application and M occurs deeply in t;
(3) M does not occur in t.

The occur-check function returns Same-MVar y in the first case, Cycle in the second case, and No-Cycle t' in the last case, where t' is t but considered in the context Γ without M, denoted by $\Gamma \backslash M$.

In the first case, we need to consider the *vector of common positions* of x and y, that is, the maximal vector of (distinct) positions $(z_1, \ldots, z_p)$ such that $x_{\vec{z}} = y_{\vec{z}}$. We denote[3] such a situation by $m \vdash x = y \Rightarrow z \dashv p$. The most general unifier σ coincides with the identity substitution except that $M : m$ is replaced by a fresh metavariable $P : p$ in the context Γ, and σ maps M to $P(z)$.

Example 2.1. Let x, y, z be three distinct variables, and let us consider unification of $M(x, y)$ and $M(z, x)$. Given a unifier σ, since $M(x, y)[\sigma] = \sigma_M\{\underline{1} \mapsto x, \underline{2} \mapsto y\}$ and $M(z, x)[\sigma] = \sigma_M\{\underline{1} \mapsto z, \underline{2} \mapsto x\}$ must be equal, σ_M cannot depend on the variables $\underline{1}$ and $\underline{2}$. It follows that the most general unifier is $M \mapsto P$, replacing M with a fresh constant metavariable P. A similar argument shows that the most general unifier of $M(x, y)$ and $M(z, y)$ is $M \mapsto P(\underline{2})$.

The corresponding rule Same-MVar does not stipulate how to generate the fresh metavariable symbol P, although there is an obvious choice, consisting in taking M which has just been removed from the context Γ. Accordingly, the implementation keeps M but changes its arity to p, resulting in a context denoted by $\Gamma[M : p]$.

The second case tackles unification of a metavariable application with a term in which the metavariable occurs deeply. It is handled by the failing rule Cycle: there is no unifier because the size of both hand sides can never match after substitution.

The last case described by the rule No-cycle is unification of $M(x)$ with a term t in which M does not occur. This kind of unification problem is handled specifically by a previously defined function prune, which we now describe. The motivation behind this pruning phase lies in the intuition that $M(x)$ and t should be unified by replacing M with $t[x_i \mapsto i]$. However, this only makes sense if the free variables of t are in x. For example, if t is a variable that does not occur in x, then obviously there is no unifier. Nonetheless, it is possible to prune the *outbound* variables in t as long as they only occur in metavariable arguments, by restricting the arities of those metavariables. As an example, if t is a metavariable application $N(x, y)$, then although the free variables are not all included in x, the most general unifier still exists, essentially replacing N with M, discarding the outbound variables y.

For this pruning phase, we use the notation $\Gamma \vdash t :> x \Rightarrow t'; \sigma \dashv \Delta$, where t is a term in the metavariable context Γ, while x is the argument of the metavariable whose arity is left implicit, as well as its (irrelevant) name. The output is the most general unifier of t and $M(x)$, considered in the extended metavariable context $M : m, \Gamma$. Note that the data for a substitution from $M : m, \Gamma$ to Δ consists of a term $\Delta; m \vdash t'$ for substituting M, and a substitution $\sigma : \Gamma \longrightarrow \Delta$. Following the above pruning intuition, t' is the term t where the outbound variables have been pruned. Accordingly, the type signature for the pruning phase is

$$\mathsf{prune} : \forall \{\Gamma\ n\} \rightarrow \mathsf{Tm}\ \Gamma\ n \rightarrow \forall \{m\} \rightarrow m \Rightarrow n \rightarrow \mathsf{Maybe}\ (m :: \Gamma \longrightarrow ?)$$

This function recursively inspects its argument. The base metavariable case corresponds to unification of $M(x)$ and $M'(y)$ where M and M' are distinct metavariables. In this case, we need to consider the vectors of *common value positions* $(l_1, \ldots, l_p)$ and $(r_1, \ldots, r_p)$ between $x_1, \ldots, x_m$ and

[3] The similarity with the above introduced notation is no coincidence: as we will see (Remark 3.4), both are (co)equalisers.

$y_1, \ldots, y_{m'}$, i.e., the pair of maximal lists $(\vec{l}, \vec{r})$ of distinct positions such that $x_{\vec{l}} = y_{\vec{r}}$. We denote[4] such a situation by $m \vdash x :> y \Rightarrow l; r \dashv p$. The most general unifier σ coincides with the identity substitution except that the metavariables M and M' are removed from the context and replaced by a single metavariable declaration $P : p$. Then, σ maps M to $P(l)$ and M' to $P(r)$.

Example 2.2. Let x, y, z be three distinct variables. The most general unifier of $M(x, y)$ and $N(z, x)$ is $M \mapsto N'(1), N \mapsto N'(2)$. The most general unifier of $M(x, y)$ and $N(z)$ is $M \mapsto N', N \mapsto N'$.

As for the rule Same-Var, the corresponding rule P-Flex does not stipulate how to generate the fresh metavariable symbol P, although the implementation makes an obvious choice, reusing the name M.

The intuition for the application case is that if we want to unify $M(x)$ with $t\ u$, we can refine $M(x)$ to be $M_1(x)\ M_2(x)$, where M_1 and M_2 are two fresh metavariables to be unified with t and u. The same intuition applies for λ-abstraction, but here we apply the fresh metavariable corresponding to the body of the λ-abstraction to the bound variable $n + 1$, which needs not be pruned. In the variable case, $i\{x\}^{-1}$ returns the index j such that $i = x_j$, or fails if no such j exist.

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

2.2 Generalisation

In the previous section, we described a unification algorithm for λ-calculus. In this section, we show how to abstract over λ-calculus to get a generic algorithm, parameterised by a new notion of signature to account for syntax with metavariables. We call them *friendly generalised binding signatures*, or friendly GB-signatures. They consist of two components:

(1) a GB-signature (formally introduced Definition 3.11), specifying a syntax with metavariables supporting renaming, metavariable substitution;
(2) some additional structures used in the unification algorithm and properties ensuring its correctness, making the GB-signature *friendly* (formally defined in Definition 3.12), which abstracts the definition of vectors of common positions or values as well as the inverse renaming $-\{-\}^{-1}$ used in the variable case of Figure **??**.

Let us focus on the notion of GB-signature, starting from binding signatures Aczel [2016]. To recall, a binding signature (O, α) specifies for each natural number n a set of n-ary operation symbols O_n and for each $o \in O_n$, an arity $\alpha_o = (\overline{o}_1, \ldots, \overline{o}_n)$ as a list of natural numbers specifying how many variables are bound in each argument. For example, pure λ-calculus is specified by $O_1 = \{lam\}$, $O_2 = \{app\}$, $\alpha_{app} = (0, 0)$, $\alpha_{lam} = (1)$, and $O_n = \emptyset$ for any natural number $n \notin \{1, 2\}$. Now, a GB-signature, implemented as in Figure 6, consists in a tuple $(\mathcal{A}, O, \alpha)$ consisting of

- a small category $\mathcal{A}$ whose objects are called *arities* or *variable contexts*, and whose morphisms are called *renamings*;
- for each variable context a and natural number n, a set of n-ary operation symbols $O_n(a)$;
- for each operation symbol $o \in O_n(a)$, a list of variable contexts $\alpha_o = (\overline{o}_1, \ldots, \overline{o}_n)$.

such that O and α are functorial in a suitable sense (see Remark 2.6 below). Intuitively, $O_n(a)$ is the set of n-ary operation symbols available in the variable context a. In the Agda code, O is not indexed by natural numbers. Instead, for each variable context a, the type $O\ a$ which gathers all the available operation symbols in the variable context a, whatever their arities are. Moreover, the

[4] The similarity with the notation for the pruning phase is no coincidence: both can be interpreted as pullbacks (or pushouts), as we will see in Remark 5.1.

```
record Signature : Set where
  field
    A : Set
    _⇒_ : A → A → Set
    id : ∀ {a} → (a ⇒ a)
    _∘_ : ∀ {a b c} → (b ⇒ c) → (a ⇒ b) → (a ⇒ c)
    O : A → Set
    α : ∀ {a} → O a → List A

  - [a₁, ⋯, aₙ] ⟹ [b₁, ⋯, bₘ] is isomorphic to a₁⇒b₁ × ⋯ × aₙ⇒bₙ if n=m
  - Otherwise, it is isomorphic to the empty type.
  _⟹_ : List A → List A → Set
  as ⟹ bs = Pointwise _⇒_ as bs

  field
    - The last two fields account for functoriality
    _{_} : ∀ {a} → O a → ∀ {b} (x : a ⇒ b) → O b
    _^_ : ∀ {a b}(x : a ⇒ b)(o : O a) → α o ⟹ α (o { x } )
```

Fig. 6. Generalised binding signatures in Agda

```
MetaContext = List A
data Tm (Γ : MetaContext) (a : A) : Set where
  Rigid : ∀ (o : O a) → (α o ⟶ Γ) → Tm Γ a
  _(_) : ∀ {m} → m ∈ Γ → m ⇒ a → Tm Γ a
```

$$\frac{o \in O_n(a) \qquad \overbrace{\Gamma; \bar{o}_1 \vdash t_1 \qquad \ldots \qquad \Gamma; \bar{o}_n \vdash t_n}^{\text{"}\alpha_o \xrightarrow{\vec{t}} \Gamma\text{"}}}{\Gamma; a \vdash o(t_1, \ldots, t_n)}\textsc{Rig}$$

$$\frac{M : m \in \Gamma \qquad x \in \hom_{\mathcal{A}}(m, a)}{\Gamma; a \vdash M(x)}\textsc{Flex}$$

Fig. 7. Syntax generated by a GB-signature

Agda definition doesn't include properties such as associativity of morphism composition, which are required in the correctness proof, but not in the implementation.

The syntax specified by a GB-signature $(\mathcal{A}, O, \alpha)$ is inductively defined in figure 7, where a context $\Gamma; a$ consists of a variable context a and a metavariable context Γ, as a metavariable arity function from a finite set of metavariable symbols to the set of objects of $\mathcal{A}$. We call a term *rigid* if it is of the shape $o(\ldots)$, *flexible* if it is $M(\ldots)$.

Remark 2.3. As in the previous section, we use a nameless convention for metavariable contexts, which are just lists of variable contexts. As a consequence, the argument of an operation o in the context $\Gamma; a$ can be specified either as a metavariable substitution (defined as in Figure 5) from $\alpha_o = (\bar{o}_1, \ldots, \bar{o}_n)$ to Γ, as in the Agda code, or explicitly as a list of terms $(t_1, \ldots, t_n)$ such that $\Gamma; \bar{o}_i \vdash t_i$, as in the rule Rig.

Remark 2.4. The syntax in the empty metavariable context does not depend on the morphisms in $\mathcal{A}$. In fact, by restricting the morphisms in the category of arities to identity morphisms, any

GB-signature induces an indexed container Altenkirch and Morris [2009] generating the same syntax without metavariables.

Example 2.5. Binding signatures can be compiled into GB-signatures. More specifically, a syntax specified by a binding signature (O, α) is also generated by the GB-signature $(\mathbb{F}_m, O', \alpha')$, where

- $\mathbb{F}_m$ is the category of finite cardinals and injections between them;
- $O'_n(p) = \{v_1, \ldots, v_p\} \sqcup \{o_p | o \in O_n\}$;
- $\alpha'_{v_i} = ()$ and $\alpha'_{o_p} = (p + \overline{o}_1, \ldots, p + \overline{o}_n)$ for any $i, p, n \in \mathbb{N}, o \in O_n$.

Note that variables v_i are explicitly specified as nullary operations and thus do not require a dedicated generating rule, contrary to what happens with binding signatures. Moreover, the choice of renamings (i.e., morphisms in the category of arities) is motivated by the Flex rule. Indeed, if M has arity $m \in \mathbb{N}$, then a choice of arguments in the variable context $a \in \mathbb{N}$ consists of a list of distinct variables in the variable context a, or equivalently, an injection between the cardinal sets m and a, that is, a morphism in $\mathbb{F}_m$ between m and a.

GB-signatures capture multi-sorted binding signatures such as simply-typed λ-calculus, or polymorphic syntax such as System F (see Section §8).

Remark 2.6. In the notion of GB-signature, functoriality ensures that the generated syntax supports renaming: given a morphism $f : a \to b$ in $\mathcal{A}$ and a term $\Gamma; a \vdash t$, we can recursively define a term $\Gamma; b \vdash t\{f\}$. The case of metavariables is simple: $M(x)\{f\} = M(f \circ x)$. For an operation $o(t_1, \ldots, t_n)$, functoriality provides the following components:

(1) an operation symbol $o\{f\} \in O_n(b)$;
(2) a list of morphisms $(f_1^o, \ldots, f_n^o)$ in $\mathcal{A}$ such that $f_i^o : \overline{o}_i \to \overline{o\{f\}}_i$ for each $i \in \{1, \ldots, n\}$.

Then, $o(t_1, \ldots, t_n)\{f\}$ is defined as $o\{f\}(t_1\{f_1^o\}, \ldots, t_n\{f_n^o\})$.

Notation 2.7. If Γ and Δ are two (nameless) metavariable contexts $(a_1, \ldots, a_n)$ and $(b_1, \ldots, b_n)$, we write $\delta : \Gamma \Longrightarrow \Delta$ to mean that δ is a vector of renamings $(\delta_1, \ldots, \delta_n)$ between Γ and Δ, in the sense that each δ_i is a morphism between a_i and b_i. For example, the second component in Remark 2.6 is specified as $f^o : \alpha_o \Longrightarrow \alpha_{o\{f\}}$.

Remark 2.8. A vector of renamings $\delta : \Gamma \Rightarrow \Delta$ canonically induces a metavariable substitution $\overline{\delta} : \Delta \to \Gamma$, mapping M_i to $M_i(\delta_i)$.

The Agda definitions for λ-calculus in Section §2.1 generalise for a syntax generated by a generic signature. The resulting code is usually shorter because the application, λ-abstraction, and variable cases are merged into a single rigid case. Moreover, because of Remark 2.3, we find it more convenient define operations on terms mutually with the corresponding operations on substitutions. For example, composition of substitutions is defined mutually with substitution of terms in Figure 8 (to be compared with Figure 5). We are similarly led to generalise unification of terms to unification of substitutions. Given two substitutions $\delta_1, \delta_2 : \Gamma' \to \Gamma$, we write $\Gamma \vdash \delta_1 = \delta_2 \Rightarrow \sigma \dashv \Delta$ to mean that $\sigma : \Gamma \to \Delta$ unifies δ_1 and δ_2, in the sense that $\delta_1[\sigma] = \delta_2[\sigma]$, and is the most general one, i.e., it uniquely factors any other unifier of δ_1 and δ_2. The unification algorithm is thus split in two functions for single terms and for substitutions:

```
unify : ∀ {Γ a} → Tm Γ a → Tm Γ a → Maybe (Γ ⟶?)
unify-σ : ∀ {Γ Γ'} → (Γ' ⟶ Γ) → (Γ' ⟶ Γ) → Maybe (Γ ⟶?)
```

The function unify-flex-* unifying a metavariable application with a term has the same code as for λ-calculus that we already discussed (Figure **??**). What differ are the implementations of the called functions: the occur-check $u \mathbin{\backslash ?_t} M$, whose definition is adapted to the syntax generated

```
_[_]t : ∀ {Γ a} → Tm Γ a → ∀ {Δ} → (Γ ⟶ Δ) → Tm Δ a
_[_]s : ∀ {Γ₁ Γ₂ Γ₃} → (Γ₁ ⟶ Γ₂) → (Γ₂ ⟶ Γ₃) → (Γ₁ ⟶ Γ₃)

Rigid o δ [ σ ]t = Rigid o (δ [ σ ]s)
M ( x ) [ σ ]t = VecList.nth M σ { x }

δ [ σ ]s = VecList.map (λ _ t → t [ σ ]t) δ
```

$$\frac{\Gamma; n \vdash t \qquad \sigma : \Gamma \to \Delta}{\Delta; n \vdash t[\sigma]}$$

$$\frac{\sigma : \Gamma_1 \to \Gamma_2 \quad \delta : \Gamma_2 \to \Gamma_3}{\underbrace{\sigma[\delta]}_{M \mapsto \sigma_M[\delta]} \quad : \Gamma_1 \to \Gamma_3}$$

Fig. 8. Metavariable substitution for a GB-signature (to be compared with Figure 5)

by a GB-signature, and the functions unify-flex-flex and prune. Regarding the former, recall that unifying two metavariable application involves computing the vector of common positions or value positions of their arguments, as in Figure **??**. Both are characterised as equalisers or pullbacks in the category $\mathbb{F}_m$ defined in Example 2.5, thus providing a natural abstraction in the generic setting. Finally, the generic pruning function is listed in Figure **??**.

As for the unification phase above, we define pruning of terms mutually with pruning of substitutions: given a substitution $\delta : \Gamma' \to \Gamma$ and a vector $x : \Gamma'' \Longrightarrow \Gamma'$ of renamings, the judgement $\Gamma \vdash \delta \mathrel{:>} x \Rightarrow \delta'; \sigma \dashv \Delta$ means that the substitution $\sigma : \Gamma \to \Delta$ extended with $\delta' : \Gamma'' \to \Delta$ is the most general unifier of δ and $\overline{x}$ as substitutions from Γ, Γ' to Δ. The types of the pruning functions are as follows:

```
data _∪_⟶? (Γ Γ' : MetaContext) : Set where
  _◄_„_ : ∀ Δ → (Γ ⟶ Δ) → (Γ' ⟶ Δ) → Γ ∪ Γ' ⟶?
prune : ∀ {Γ a m} → Tm Γ a → m ⇒ a → Maybe (m :: Γ ⟶?)
prune-σ : ∀ {Γ Γₐ Γₘ} → (Γₐ ⟶ Γ) → (Γₘ ⟹ Γₐ) → Maybe (Γₘ ∪ Γ ⟶?)
```

Let us now detail the rigid case: here we want to unify an operation $o(\delta)$ with a fresh metavariable application $M(x)$. Any unifier must replace M with an operation $o'(\delta')$, such that $o'\{x\}(\delta'\{x^{o'}\}) = o(\delta)$. In particular, o must be have a preimage o' for renaming by x. This is precisely the point of the inverse renaming $o\{x\}^{-1}$ in the Agda code: it returns a preimage o' if it exists, or fails. In the λ-calculus case, this check is only explicit for variables, since there is a single version of application and λ-abstraction (as symbols) in any variable context.

As we discussed at the beginning, from GB-signatures we can define the specified syntax with metavariables. In Figure 9 are listed the additional components making a GB-signature friendly, on which the algorithm relies on. To sum up,

- equalisers and pullbacks are used to unify two metavariable applications;
- equality of operation symbols is used when unifying two rigid terms.
- inverse renaming is used when pruning a rigid term.

3 CATEGORICAL SEMANTICS

It remains to prove that each rule is sound, e.g., for the rule U-Split, if the output of the premises are most general unifiers, then so is the conclusion. To do so, the next sections rely on the categorical semantics of pattern unification that we introduce in this section. In Section §3.1, we relate pattern unification to a coequaliser construction, and in Section §3.2, we provide a formal definition of GB-signatures with Initial Algebra Semantics for the generated syntax.

```
record isFriendly (S : Signature) : Set where
  open Signature S
  field
    equaliser : ∀ {a m} → (x y : m ⇒ a) → Σ A (λ p → p ⇒ m)
    pullback : ∀ {m m' a} → (x : m ⇒ a) → (y : m' ⇒ a) → Σ A (λ p → p ⇒ m × p ⇒ m')
    _≟_ : ∀ {a}(o o' : O a) → Dec (o ≡ o')
    _{_}⁻¹ : ∀ {a}(o : O a) → ∀ {b}(x : b ⇒ a) → Maybe-PreImage (_{ x }) o
```

Fig. 9. Friendly GB-signatures in Agda

3.1 Pattern unification as a coequaliser construction

In this section, we assume given a GB-signature $S = (\mathcal{A}, O, \alpha)$ and explain how most general unifiers can be thought of as equalisers in a multi-sorted Lawvere theory, as is well-known in the first-order case Barr and Wells [1990]; Rydeheard and Burstall [1988]. We furthermore provide a formal justification for the error metavariable context $\perp$.

Lemma 3.1. *Metavariable contexts and substitutions (with their composition) between them define a category* $\mathrm{MCon}(S)$.

This relies on functoriality of GB-signatures that we will spell out formally in the next section. There, we will see in Lemma 3.19 that this category fully faithfully embeds in a Kleisli category for a monad generated by S on $[\mathcal{A}, \mathrm{Set}]$.

Remark 3.2. $\mathrm{MCon}(S)$ is the opposite category of a multi-sorted Lawvere theory: the sorts are the objects of $\mathcal{A}$. This theory is not freely generated by operations unless $\mathcal{A}$ is discrete, in which case we recover (multi-sorted) first-order unification. Even the GB-signature induced (as in Example 2.5) by an empty binding signature is not "free".

Since a substitution is precisely a list of terms sharing the same metavariable context Γ, a unification problem for two list of terms is equivalently given by a pair of parallel substitutions

$$\begin{tikzcd}\Gamma \arrow[r, shift left, "\sigma_1"] \arrow[r, shift right, "\sigma_2"'] & \Delta\end{tikzcd} .$$

Lemma 3.3. *The most general unifier of two lists of terms* $\Delta; n_i \vdash t_i, u_i$*, if it exists, is characterised as the coequaliser of* $\vec{t}$ *as* $\vec{u}$ *as substitutions from* $(N_1 : n_1, \dots)$ *to* Δ.

Remark 3.4. This justifies a common interpretation as (co)equalisers of the two variants of the notation $- \vdash - = - \Rightarrow - \dashv -$ involved in Figure ??.

Pattern unification is often stated as the existence of a coequaliser on the condition that there is a unifier. It turns out that we can get rid of this condition by considering the category $\mathrm{MCon}(S)$ freely extended with a terminal object $\perp$, as we now explain.

Definition 3.5. Given a category $\mathscr{B}$, let $\mathscr{B}_\perp$ denote the category $\mathscr{B}$ extended freely with a terminal object $\perp$.

Notation 3.6. We denote by ! any terminal morphism to $\perp$ in $\mathscr{B}_\perp$.

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

Lemma 3.7. *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 $\mathscr{B}_\perp$.

The following result is also useful.

Lemma 3.8. *Given a category $\mathscr{B}$, the canonical embedding functor $\mathscr{B} \to \mathscr{B}_\perp$ creates colimits.*

This ensures in particular that coproducts in $\mathrm{MCon}(S)$, which are computed as union of metavariable contexts, are also coproducts in $\mathrm{MCon}_\perp(S)$.

The main property of this extension for our purposes is the following corollary.

Corollary 3.9. *Any coequaliser in $\mathrm{MCon}(S)$ is also a coequaliser in $\mathrm{MCon}_\perp(S)$. Moreover, whenever there is no unifier of two lists of terms, then the coequaliser of the corresponding parallel arrows in $\mathrm{MCon}_\perp(S)$ exists: it is the terminal cocone on $\perp$.*

Categorically speaking, our pattern unification algorithm provides an explicit proof of the following statement, where the conditions for a signature to be *pattern-friendly* are introduced in the next section (Definition 3.12).

Theorem 3.10. *Given any pattern-friendly signature S, the category $\mathrm{MCon}_\perp(S)$ has coequalisers.*

3.2 Initial Algebra Semantics for GB-signatures

Definition 3.11. A *generalised binding signature*, or *GB-signature*, is a tuple $(\mathcal{A}, O, \alpha)$ consisting of

- a small category $\mathcal{A}$ of arities and renamings between them;
- a functor $O_-(-) : \mathbb{N} \times \mathcal{A} \to \mathrm{Set}$ of operation symbols;
- a functor $\alpha : \int J \to \mathcal{A}$

where $\int J$ denotes the category of elements of $J : \mathbb{N}\times\mathcal{A} \to \mathrm{Set}$ mapping (n, a) to $O_n(a)\times\{1, \dots, n\}$, defined as follows:

- objects are tuples (n, a, o, i) such that $o \in O_n(a)$ and $i \in \{1, \dots, n\}$;
- a morphism between (n, a, o, i) and (n', a', o', i') is a morphism $f : a \to a'$ such that $n = n'$, $i = i'$ and $o\{f\} = o'$ where $o\{f\}$ denotes the image of o by the function $O_n(f) : O_n(a) \to O_n(a')$. introduce the reverse partial notation for the P-RIG rule ?

We now introduce our conditions for the generic unification algorithm to be correct.

Definition 3.12. A GB-signature $S = (\mathcal{A}, O, \alpha)$ is said *pattern-friendly* if

(1) $\mathcal{A}$ has finite connected limits;
(2) all morphisms in $\mathcal{A}$ are monomorphic;
(3) each $O_n(-) : \mathcal{A} \to \mathrm{Set}$ preserves finite connected limits;
(4) α preserves finite connected limits.

Remark 3.13. The first condition is equivalent to the existence of equalisers and pullbacks in $\mathcal{A}$, since any finite connected limit can be constructed from those.

These conditions ensure the following two properties.

Property 3.14. The following properties hold.

(i) The action of $O_n : \mathcal{A} \to \mathrm{Set}$ on any renaming is an injection: given any $o \in O_n(b)$ and renaming $f : a \to b$, there is at most one $o' \in O_n(a)$ such that $o = o'\{f\}$.

(ii) Let $\mathcal{L}$ be the functor $\mathcal{A}^{op} \longrightarrow \mathrm{MCon}(S)$ mapping a morphism $x \in \hom_{\mathcal{A}}(b, a)$ to the substitution $(X : a) \longrightarrow (X : b)$ selecting (by the Yoneda Lemma) the term $X(x)$. Then, $\mathcal{L}$ preserves finite connected colimits: it maps pullbacks and equalisers in $\mathcal{A}$ to pushouts and coequalisers in $\mathrm{MCon}(S)$.

Proof. (i) Since O_n preserves finite connected limits, it preserves monomorphisms because a morphism $f : a \longrightarrow b$ is monomorphic if and only if the following square is a pullback (see Mac Lane [1998, Exercise III.4.4]).

$$\begin{tikzcd}
A \arrow[r, equal] \arrow[d, equal] & A \arrow[d, "f"] \\
A \arrow[r, "f"'] & B
\end{tikzcd}$$

(ii) The proof is deferred to the end of this section. □

The first property is used for soundness of the rules P-Rig and P-Fail. The second one is used to justify unification of two metavariables applications as pullbacks and equalisers in $\mathcal{A}$, in the rules U-Flex and P-Flex.

Remark 3.15. A metavariable application $\Gamma; a \vdash M(x)$ corresponds to the composition $\mathcal{L}x[in_M]$, where in_M is the coproduct injection $(X : m) \cong (M : m) \hookrightarrow \Gamma$.

The rest of this section can be safely skipped at first reading: we provide Initial Algebra Semantics for the generated syntax that we exploit to prove Property 3.14.(ii).

Any GB-signature $S = (\mathcal{A}, O, \alpha)$, generates an endofunctor F_S on $[\mathcal{A}, \mathrm{Set}]$, that we denote by just F when the context is clear, defined by

$$F_S(X)_a = \coprod_{n \in \mathbb{N}} \coprod_{o \in O_n(a)} X_{\bar{o}_1} \times \cdots \times X_{\bar{o}_n}.$$

Lemma 3.16. *F is finitary and generates a free monad T. Moreover, TX is the initial algebra of $Z \mapsto X + FZ$.*

Proof. F is finitary because filtered colimits commute with finite limits Mac Lane [1998, Theorem IX.2.1] and colimits. The free monad construction is due to Reiterman [1977]. □

Lemma 3.17. *The syntax generated by a GB-signature (see Figure 7) is recovered as free algebras for F. More precisely, given a metavariable context $\Gamma = (M_1 : m_1, \ldots, M_p : m_p)$,*

$$T(\underline{\Gamma})_a \cong \{t \mid \Gamma; a \vdash t\}$$

where $\underline{\Gamma} : \mathcal{A} \longrightarrow \mathrm{Set}$ is defined as the coproduct of representable functors $\coprod_i y m_i$, mapping a to $\coprod_i \hom_{\mathcal{A}}(m_i, a)$.

Notation 3.18. Given a metavariable context Γ. We sometimes denote $\underline{\Gamma}$ just by Γ.

If $\Gamma = (M_1 : m_1, ..., M_p : m_p)$ and Δ are metavariable contexts, a Kleisli morphism $\sigma : \Gamma \longrightarrow T\Delta$ is equivalently given (by the Yoneda Lemma and the universal property of coproducts) by a metavariable substitution from Γ to Δ. Moreover, Kleisli composition corresponds to composition of substitutions. This provides a formal link between the category of metavariable contexts $\mathrm{MCon}(S)$ and the Kleisli category of T

Lemma 3.19. *The category $\mathrm{MCon}(S)$ is equivalent to the full subcategory of Kl_T spanned by coproducts of representable functors.*

We will exploit this characterisation to prove various properties of this category when the signature is *pattern-friendly*.

Lemma 3.20. *Given a GB-signature $S = (\mathcal{A}, O, \alpha)$ such that $\mathcal{A}$ has finite connected limits, F_S restricts as an endofunctor on the full subcategory $\mathscr{C}$ of $[\mathcal{A}, \mathrm{Set}]$ consisting of functors preserving finite connected limits if and only if the last two conditions of Definition 3.12 holds.*

Proof. See Appendix §A. □

We now assume given a pattern-friendly signature $S = (\mathcal{A}, O, \alpha)$.

Lemma 3.21. *$\mathscr{C}$ is closed under limits, coproducts, and filtered colimits. Moreover, it is cocomplete.*

Proof. Cocompleteness follows from Adámek and Rosicky [1994, Remark 1.56], since $\mathscr{C}$ is the category of models of a limit sketch, and is thus locally presentable, by Adámek and Rosicky [1994, 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. Coproducts and filtered colimits also commute with finite connected limits Adámek et al. [2002, Example 1.3.(vi)]. □

Corollary 3.22. *T restricts as a monad on $\mathscr{C}$ freely generated by the restriction of F as an endofunctor on $\mathscr{C}$ (Lemma 3.20).*

Proof. The result follows from the construction of T using colimits of initial chains, thanks to the closure properties of $\mathscr{C}$. More specifically, TX can be constructed as the colimit of the chain $\emptyset \longrightarrow H\emptyset \longrightarrow HH\emptyset \longrightarrow \dots$, where $\emptyset$ denotes the constant functor mapping anything to the empty set, and $HZ = FZ + X$. □

We now turn to the proof of Property 3.14.(ii).

By right continuity of the homset bifunctor, any representable functor is in $\mathscr{C}$ and thus the embedding $\mathscr{C} \longrightarrow [\mathcal{A}, \mathrm{Set}]$ factors the Yoneda embedding $\mathcal{A}^{op} \longrightarrow [\mathcal{A}, \mathrm{Set}]$.

Lemma 3.23. *Let $\mathscr{D}$ denote the opposite category of $\mathcal{A}$ and $K : \mathscr{D} \longrightarrow \mathscr{C}$ the factorisation of $\mathscr{C} \longrightarrow [\mathcal{A}, \mathrm{Set}]$ by the Yoneda embedding. Then, $K : \mathscr{D} \longrightarrow \mathscr{C}$ preserves finite connected colimits.*

Proof. This essentially follows from the fact functors in $\mathscr{C}$ preserves finite connected limits. Let us detail the argument: let $y : \mathcal{A}^{op} \longrightarrow [\mathcal{A}, \mathrm{Set}]$ denote the Yoneda embedding and $J : \mathscr{C} \longrightarrow [\mathcal{A}, \mathrm{Set}]$ denote the canonical embedding, so that

$$y = J \circ K. \tag{1}$$

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

$$\begin{aligned}
\mathscr{C}(K \lim F, X) &\cong [\mathcal{A}, \mathrm{Set}](JK \lim F, JX) && (J \text{ is fully faithful})\\
&\cong [\mathcal{A}, \mathrm{Set}](y \lim F, JX) && (\text{By Equation } (1))\\
&\cong JX(\lim F) && (\text{By the Yoneda Lemma.})\\
&\cong \lim(JX \circ F) && (X \text{ preserves finite connected limits})\\
&\cong \lim([\mathcal{A}, \mathrm{Set}](yF-, JX)] && (\text{By the Yoneda Lemma})\\
&\cong \lim([\mathcal{A}, \mathrm{Set}](JKF-, JX)] && (\text{By Equation } (1))\\
&\cong \lim \mathscr{C}(KF-, X) && (J \text{ is full and faithful})\\
&\cong \mathscr{C}(\mathrm{colim}\, KF, X) && (\text{By left continuity of the hom-set bifunctor})
\end{aligned}$$

These isomorphisms are natural in X and thus $K \lim F \cong \mathrm{colim}\, KF$. □

Proof of Property 3.14.(ii). Let $T_{|\mathscr{C}}$ be the monad T restricted to $\mathscr{C}$, following Corollary 3.22. Since $K : \mathscr{D} \to \mathscr{C}$ preserves finite connected colimits (Lemma 3.23), composing it with the left adjoint $\mathscr{C} \to Kl_{T_{|\mathscr{C}}}$ yields a functor $\mathscr{D} \to Kl_{T_{|\mathscr{C}}}$ also preserving those colimits. Since it factors as $\mathscr{D} \xrightarrow{\mathcal{L}} \mathrm{MCon}(S) \hookrightarrow Kl_{T_{|\mathscr{C}}}$, where the right functor is full and faithful, $\mathcal{L}$ also preserves finite connected colimits. □

4 SOUNDNESS OF THE UNIFICATION PHASE

In this section, we assume a pattern-friendly GB-signature S and discuss soundness of the main rules of the main unification phase in Figure ??, which computes a coequaliser in $\mathrm{MCon}_{\perp}(S)$. More specifically, we discuss the rule sequential rule U-Split (Section §4.1), the rule U-Flex unifying metavariable with itself (Section §4.2), and the failing rule U-Cyclic for cyclic unification of a metavariable with a term which includes it deeply (Section §4.3).

4.1 Sequential unification (rule U-Split)

The rule U-Split follows from a stepwise construction of coequalisers valid in any category, as noted by [Rydeheard and Burstall 1988, Theorem 9]: if the first two diagrams below are coequalisers, then the last one as well.

$$\begin{tikzcd}
A_1 \arrow[r, shift left, "t_1"] \arrow[r, shift right, "u_1"'] & \Gamma \arrow[r, dashed, "\sigma_1"] & \Delta_1
\end{tikzcd}
\qquad
\begin{tikzcd}
 & \Gamma \arrow[dr, "\sigma_1"] & & \\
A_2 \arrow[ur, "t_2"] \arrow[dr, "u_2"'] & & \Delta_1 \arrow[r, dashed, "\sigma_2"] & \Delta_2 \\
 & \Gamma \arrow[ur, "\sigma_1"'] & &
\end{tikzcd}$$

$$\begin{tikzcd}
A_1 + A_2 \arrow[r, shift left, "{t_1,t_2}"] \arrow[r, shift right, "{u_1,u_2}"'] & \Gamma \arrow[r, dashed, "\sigma_2 \circ \sigma_1"] & \Delta_2
\end{tikzcd}$$

4.2 Flex-Flex, same metavariable (rule U-Flex)

Here we detail unification of $M(x)$ and $M(y)$, for $x, y \in \hom_{\mathscr{A}}(m, a)$. By Remark 3.15, $M(x) = \mathcal{L}x[in_M]$ and $M(y) = \mathcal{L}y[in_M]$. We exploit the following lemma with $u = \mathcal{L}x$ and $v = \mathcal{L}y$.

Lemma 4.1. *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.

$$\begin{tikzcd}
A \arrow[r, shift left, "u"] \arrow[r, shift right, "v"'] & B \arrow[r, dashed, "h"] & C
\end{tikzcd}
\qquad
\begin{tikzcd}
 & B \arrow[dr, "in_B"] & & \\
A \arrow[ur, "u"] \arrow[dr, "v"'] & & B + D \arrow[r, dashed, "h+1_D"] & C + D \\
 & B \arrow[ur, "in_B"'] & &
\end{tikzcd}$$

It follows that it is enough to compute the coequaliser of $\mathcal{L}x$ and $\mathcal{L}y$. Furthermore, by Property 3.14.(ii), it is the image by $\mathcal{L}$ of the equaliser of x and y, thus justifying the rule U-Flex.

4.3 Flex-rigid, cyclic (rule U-Cyclic)

The rule U-Cyclic handles unification of $M(x)$ and a term u such that u is rigid and M occurs in u. In this section, we show that indeed there is no unifier. More precisely, we prove Corollary 4.6 below, stating that if there is a unifier of a term u and a metavariable application $M(x)$, then either M occurs at top-level in u, or it does not occur at all. The argument follows the basic intuition that $\sigma_M = u[M \mapsto \sigma_M]$ 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.

Definition 4.2. The size[5] $|t| \in \mathbb{N}$ of a term t is recursively defined by $|M(x)| = 0$ and $|o(\vec{t})| = 1+|\vec{t}|$, with $|\vec{t}| = \sum_i t_i$.

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

Definition 4.3. For any term t we define $|t|_M$ recursively by $|M(x)|_M = 1$, $|N(x)|_M = 0$ if $N \neq M$, and $|o(\vec{t})|_M = |\vec{t}|_M$ with the sum convention as above for $|\vec{t}|_M$.

Lemma 4.4. *For any term $\Gamma, M : m; a \vdash t$, if $|t|_M = 0$, then $\Gamma; a \vdash t$. Moreover, for any $\Gamma = (M_1 : m_1, \dots, M_n : m_n)$, well-formed term t in context $\Gamma; a$, and substitution $\sigma : \Gamma \to \Delta$, we have $|t[\sigma]| = |t| + \sum_i |t|_{M_i} \times |\sigma_i|$.*

Corollary 4.5. *For any term t in context $\Gamma, M : m; a$, substitution $\sigma : \Gamma \to \Delta$, morphism $x \in \hom_{\mathcal{A}}(m, a)$ and u in context $\Delta; u$, we have $|t[\sigma, M \mapsto u]| \geq |t| + |u| \times |t|_M$ and $|M(x)[u]| = |u|$.*

Corollary 4.6. *Let t be a term in context $\Gamma, M : m; a$ and $x \in \hom_{\mathcal{A}}(m, a)$ such that $(\sigma, M \mapsto u) : (\Gamma, M : m) \to \Delta$ unifies t and $M(x)$. Then, either $t = M(y)$ for some $y \in \hom_{\mathcal{A}}(m, a)$, or $\Gamma; a \vdash t$.*

Proof. Since $t[\sigma, u] = M(x)[u]$, we have $|t[\sigma, u]| = |M(x)[u]|$. Corollary 4.5 implies $|u| \geq |t| + |u| \times |t|_M$. Therefore, either $|t|_M = 0$ and we conclude by Lemma 4.4, or $|t|_M > 0$ and $|t| = 0$, so that t is $M(y)$ for some y. □

5 SOUNDNESS OF THE PRUNING PHASE

In this section, we assume a pattern-friendly GB-signature S and prove soundness of the main rule of the pruning phase. This phase handles unification of a list of terms $\Gamma; n_i \vdash t_i$ with a list of fresh metavariable applications $M_1(x_1), \dots, M_p(x_p)$, in the extended metavariable context $\Gamma, M_1 : m_1, \dots, M_p : x_p$. Categorically speaking, we are looking at the following coequalising diagram in $\mathrm{MCon}(S)$.

$$\begin{tikzcd}
 & \Gamma \arrow[dr, "in_1"] & \\
\overrightarrow{N : n} \arrow[ur, "\vec{t}"] \arrow[dr, "\overrightarrow{M(x)}"'] & & \Gamma, \overrightarrow{M : m} \\
 & \overrightarrow{M : m} \arrow[ur, "in_2"'] &
\end{tikzcd}$$

The P-Split rule is a straightforward adaption of the U-Split rule specialised to those specific coequaliser diagrams.

Remark 5.1. A unifier $\Gamma, \overrightarrow{M : m} \to \Delta$ splits into two components: a substitution $\sigma : \Gamma \to \Delta$ and a substitution $\vec{u}$ from $\overrightarrow{N : n}$ to $\overrightarrow{M : m}$ such that $t_i[\sigma] = u_i\{x_i\}$ for each $i \in \{1, \dots, p\}$. Moreover, the coequaliser $\sigma, \vec{u} : (\Gamma, \overrightarrow{M : m}) \to \Delta$ is equivalently characterised as a pushout

$$\begin{tikzcd}
\overrightarrow{N : n} \arrow[r, "\overrightarrow{M(x)}"] \arrow[d, "\vec{t}"'] & \overrightarrow{M : m} \arrow[d, "\vec{u}"] \\
\Gamma \arrow[r, "\sigma"'] & \Delta
\end{tikzcd}$$

This justifies a common interpretation as pushouts of the two variants of the notation $- \vdash - :> - \Rightarrow -; -$ involved in Figure ??, in $\mathcal{A}^{op}$ and $\mathrm{MCon}(S)$.

[5] The difference with the notion of size introduced in section §6 is that metavariables are of size 0.

In the following sections, we detail soundness of the rules for the rigid case (Section §5.1) and then for the flex case (Section §5.2).

5.1 Rigid (rules P-Rig and P-Fail)

The rules P-Rig and P-Fail handle non-cyclic unification of $M(x)$ with $\Gamma; a \vdash o(\vec{t})$ in the metavariable context $\Gamma, M : m$ for some $o \in O_n(a)$. By Remark 5.1, a unifier is given by a substitution $\sigma : \Gamma \longrightarrow \Delta$ and a term u such that

$$o(\vec{t}[\sigma]) = u\{x\}. \tag{2}$$

Now, u is either some $M(y)$ or $o'(\vec{v})$. But in the first case, $u\{x\} = M(y)\{x\} = M(x \circ y)$, contradicting Equation (2). Therefore, $u = o'(\vec{v})$ for some $o' \in O_n(m)$ and $\vec{v} = (v_1, \ldots, v_n)$ is a list of terms such that $\Delta; \overline{o'}_i \vdash v_i$. Then, $u\{x\} = (o'\{x\})(v_1\{x_1^{o'}\}, \ldots,)$. It follows from Equation (2) that $o = o'\{x\}$, and $t_i[\sigma] = v_i\{x_i^{o'}\}$.

Note that there is at most one o' such that $o = o'\{x\}$, by Property 3.14.(i). In this case, a unifier is equivalently given by a substitution $\sigma : \Gamma \longrightarrow \Delta$ and a list of terms $\vec{v} = (v_1, \ldots, v_n)$ such that $\Delta; \overline{o'}_i \vdash v_i$ and $t_i[\sigma] = v_i\{x_i^{o'}\}$. But, by Remark 5.1, this is precisely the data for a unifier of $\vec{t}$ and $M_1(x_i^{o'}), \ldots, M_n(x_n^{o'})$. This actually induces an isomorphism between the two categories of unifiers, thus justifying the rules P-Rig and P-Fail.

5.2 Flex (rule P-Flex)

The rule P-Flex handles unification of $\Gamma, N : n; a \vdash N(x)$ and $M(y)$ where M is fresh in $\Gamma, N : n$.

Note that $M(y)$, as a substitution $(A : a) \longrightarrow (M : m)$, is isomorphic to $\mathcal{L}y$, while $N(x) = \mathcal{L}x[in_N]$, by Remark 3.15. Thanks to the following lemma, it is actually enough to compute the pushout of $\mathcal{L}x$ and $\mathcal{L}y$.

Lemma 5.2. *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.

$$\begin{tikzcd}
A \arrow[r, "f"] \arrow[d, "g"'] & B \arrow[d, dashed, "u"] \\
X \arrow[r, dashed, "\sigma"'] & Z
\end{tikzcd}
\qquad
\begin{tikzcd}
A \arrow[r, "f"] \arrow[d, "g"'] & B \arrow[d, dashed, "u"] \\
X \arrow[d, "in_1"'] & Z \arrow[d, dashed, "in_1"] \\
X + Y \arrow[r, dashed, "\sigma+Y"'] & Z + Y
\end{tikzcd}$$

By Property 3.14.(ii), the pushout of $\mathcal{L}x$ and $\mathcal{L}y$ is the image by $\mathcal{L}$ of the pullback of x and y in $\mathcal{A}$, thus justifying the rule P-Flex.

6 TERMINATION

In this section, we sketch an explicit argument to justify termination of our algorithm described in Figure 2. Indeed, it involves three recursive calls in the pruning phase (cf the rules P-Rig and P-Split), as well as in the main unification phase (cf the rules U-Rig and U-Split). In each phase, the second recursive call for splitting is not structurally recursive, making Agda unable to check termination. However, we can devise an adequate notion of input size so that for each recursive call, the inputs are strictly smaller than the inputs of the calling site. First, we define the size $|\Gamma|$

of a metavariable context Γ as its length. We also recursively define the size $||t||$ of a term t by $||M(x)|| = 1$ and $||o(\vec{t})|| = 1 + ||\vec{t}||$, with $||\vec{t}|| = \sum_i ||t_i||$. Note that no term is of empty size.

Let us first quickly justify termination of the pruning phase. Consider the above defined size of the input, which is a term t for prune, or a list of terms $\vec{t}$ for prune-σ. It is straightforward to check that the sizes of the inputs of recursive calls are strictly smaller thanks to the following lemmas.

Lemma 6.1. *For any term $\Gamma; a \vdash t$ and substitution $\sigma : \Gamma \longrightarrow \Delta$, if σ is a* metavariable renaming, *i.e., σ_M is a metavariable application for any $M : m \in \Gamma$, then $||t[\sigma]|| = ||t||$.*

Lemma 6.2. *If there is a finite derivation tree of $\Gamma \vdash \vec{t} :> x \Rightarrow \vec{w}; \sigma \dashv \Delta$ and $\Delta \neq \bot$, then $|\Gamma| = |\Delta|$ and σ is a metavariable renaming.*

The size invariance in the above lemma is actually used in the termination proof of the main unification phase, where we consider the size of the input to be the pair $(|\Gamma|, ||t||)$ for unify or $(|\Gamma|, ||\vec{t}||)$ for unify-σ, given as input a term t or a list of terms $\vec{t}$ in the metavariable context Γ. More precisely, it is used in the following lemma that ensures size decreasing (with respect to the lexicographic order).

Lemma 6.3. *If there is a finite derivation tree of $\Gamma \vdash \vec{t} = \vec{u} \Rightarrow \sigma \dashv \Delta$, then $|\Gamma| \geq |\Delta|$, and moreover if $|\Gamma| = |\Delta|$ and $\Delta \neq \bot$, then σ is a metavariable renaming.*

7 COMPLETENESS

In this section, we explain why soundness (Section §5 and Section §4) and termination (Section §6) entail completeness. Intuitively, one may worry that the algorithm fails in cases where it should not. In fact, soundness already ensures that this cannot happen because failure is really handled as a coequaliser, on par with most general unifiers. As explained in Section §3.1, this is done by considering the category $\mathrm{MCon}_\bot(S)$ extending category $\mathrm{MCon}(S)$ with a (dignified) error object $\bot$. Corollary 3.9 implies that since the algorithm terminates and computes the coequaliser in $\mathrm{MCon}_\bot(S)$, it always finds the most general unifier in $\mathrm{MCon}(S)$ if it exists, and otherwise returns failure (i.e., the map to the terminal object $\bot$).

8 APPLICATIONS

In this section, we present various examples of pattern-friendly signatures. We start in Section §8.1 with a variant of pure λ-calculus where metavariable arguments are sets rather than lists. Then, in Section §8.2, we present simply-typed λ-calculus, as an example of syntax specified by a multi-sorted binding signature. Next, we introduce an example of unification for ordered syntax in Section §8.3, and finally we present an example of polymorphic such as System F, in Section §8.4.

8.1 Metavariable arguments as sets

If we think of the arguments of a metavariable as specifying the available variables, then it makes sense to assemble them in a set rather than in a list. This motivates considering the category $\mathcal{A} = \mathbb{I}$ whose objects are natural numbers and a morphism $n \longrightarrow p$ is a subset of $\{1, \ldots, p\}$ 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. Then, a metavariable takes as argument a set of variables, rather than a list of distinct variables. In this approach, unifying two metavariables (see the rules U-Flex and P-Flex) amount to computing a set intersection.

8.2 Simply-typed λ-calculus

In this section, we present the example of simply-typed λ-calculus. Our treatment generalises to any multi-sorted binding signature Fiore and Hur [2010].

Let T denote the set of simple types generated by a set of atomic types and a binary arrow type construction $- \Rightarrow -$. Let us now describe the category $\mathcal{A}$ of arities, or variable contexts, and renamings between them. An arity $\vec{\sigma} \longrightarrow \tau$ consists of a list of input types $\vec{\sigma}$ and an output type τ. A term t in $\vec{\sigma} \longrightarrow \tau$ considered as a variable context is intuitively a well-typed term t of type τ potentially using variables whose types are specified by $\vec{\sigma}$. A valid choice of arguments for a metavariable $M : (\vec{\sigma} \longrightarrow \tau)$ in variable context $\vec{\sigma}' \longrightarrow \tau'$ first requires $\tau = \tau'$, and consists of an injective renaming $\vec{r}$ between $\vec{\sigma} = (\sigma_1, \dots, \sigma_m)$ and $\vec{\sigma}' = (\sigma'_1, \dots, \sigma'_n)$, that is, a choice of distinct positions $(r_1, \dots, r_m)$ in $\{1, \dots, n\}$ such that $\vec{\sigma} = \sigma'_{\vec{r}}$.

This discussion determines the category of arities as $\mathcal{A} = \mathbb{F}_m[T] \times T$, where $\mathbb{F}_m[T]$ is the category of finite lists of elements of T and injective renamings between them. Table 1 summarises the definition of the endofunctor F on $[\mathcal{A}, \mathrm{Set}]$ specifying the syntax, where $|\vec{\sigma}|_\tau$ denotes the number (as a cardinal set) of occurrences of τ in $\vec{\sigma}$.

The induced signature is pattern-friendly and so the generic pattern unification algorithm applies. Equalisers and pullbacks are computed following the same pattern as in pure λ-calculus. For example, to unify $M(\vec{x})$ and $M(\vec{y})$, we first compute the vector $\vec{z}$ of common positions between $\vec{x}$ and $\vec{y}$, thus satisfying $x_{\vec{z}} = y_{\vec{z}}$. Then, the most general unifier maps $M : (\vec{\sigma} \longrightarrow \tau)$ to the term $P(\vec{z})$, where the arity $\vec{\sigma}' \longrightarrow \tau'$ of the fresh metavariable P is the only possible choice such that $P(\vec{z})$ is a valid term in the variable context $\vec{\sigma} \longrightarrow \tau$, that is, $\tau' = \tau$ and $\vec{\sigma}' = \sigma_{\vec{z}}$.

8.3 Ordered λ-calculus

Our setting handles linear ordered λ-calculus, consisting of λ-terms using all the variables in context. In this context, a metavariable M of arity $m \in \mathbb{N}$ can only be used in the variable context m, and there is no freedom in choosing the arguments of a metavariable application, since all the variables must be used, in order. Thus, there is no need to even mention those arguments in the syntax. It is thus not surprising that ordered λ-calculus is already handled by first-order unification, where metavariables do not take any argument, by considering ordered λ-calculus as a multi-sorted Lawvere theory where the sorts are the variable contexts, and the syntax is generated by operations $L_n \times L_m \longrightarrow L_{n+m}$ and abstractions $L_{n+1} \longrightarrow L_n$.

Our generalisation can handle calculi combining ordered and unrestricted variables, such as the calculus underlying ordered linear logic described in Polakow and Pfenning [2000]. In this section we detail this specific example.

The set T of types is generated by a set of atomic types and two binary arrow type constructions $\Rightarrow$ and $\twoheadrightarrow$. The syntax extends pure λ-calculus with a distinct application $t^{>}\ u$ and abstraction $\lambda^{>}u$. Variables contexts are of the shape $\vec{\sigma}|\vec{\omega} \longrightarrow \tau$, where $\vec{\sigma}$, $\vec{\omega}$, and τ are taken in T. The idea is that a term in such a context has type τ and must use all the variables of $\vec{\omega}$ in order, but is free to use any of the variables in $\vec{\sigma}$. Assuming a metavariable M of arity $\vec{\sigma}|\vec{\omega} \longrightarrow \tau$, the above discussion about ordered λ-calculus justifies that there is no need to specify the arguments for $\vec{\omega}$ when applying M. Thus, a metavariable application $M(\vec{x})$ in the variable context $\vec{\sigma}'|\vec{\omega}' \longrightarrow \tau'$ is well-formed if $\tau = \tau'$ and $\vec{x}$ is an injective renaming from $\vec{\sigma}$ to $\vec{\sigma}'$. Therefore, we take $\mathcal{A} = \mathbb{F}_m[T] \times T^* \times T$ for the category of arities, where T^* denote the discrete category whose objects are lists of elements of T. The remaining components of the GB-signature are specified in Table 1: we alternate typing rules for the unrestricted and the ordered fragments (variables, application, abstraction).

Table 1. Examples of generalised binding signatures (Definition 3.11)

	Typing rule	$O_n(\vec{\sigma} \to \tau) = \ldots +$	$\alpha_o = (\ldots)$
Simply-typed λ-calculus (Section §8.2)	$\dfrac{x : \tau \in \Gamma}{\Gamma \vdash x : \tau}$	$\{v_i \vert i \in \vert\vec{\sigma}\vert_\tau\}$	$()$
	$\dfrac{\Gamma \vdash t : \tau' \Rightarrow \tau \quad \Gamma \vdash u : \tau'}{\Gamma \vdash t\ u : \tau}$	$\{a_{\tau'} \vert \tau' \in T\}$	$\begin{pmatrix} \vec{\sigma} \to (\tau' \Rightarrow \tau) \\ \vec{\sigma} \to \tau' \end{pmatrix}$
	$\dfrac{\Gamma, x : \tau_1 \vdash t : \tau_2}{\Gamma \vdash \lambda x.t : \tau_1 \Rightarrow \tau_2}$	$\{l_{\tau_1,\tau_2} \vert \tau = (\tau_1 \Rightarrow \tau_2)\}$	$(\vec{\sigma}, \tau_1 \to \tau_2)$

	Typing rule	$O_n(\vec{\sigma}\vert\vec{\omega} \to \tau) = \ldots +$	$\alpha_o = (\ldots)$
Ordered λ-calculus (Section §8.3)	$\dfrac{x : \tau \in \Gamma}{\Gamma\vert\cdot \vdash x : \tau}$	$\{v_i \vert i \in \vert\vec{\sigma}\vert_\tau$ and $\vec{\omega} = ()\}$	$()$
	$\dfrac{}{\Gamma\vert x : \tau \vdash x : \tau}$	$\{v^> \vert \vec{\omega} = ()\}$	$()$
	$\dfrac{\Gamma\vert\Omega \vdash t : \tau' \Rightarrow \tau \quad \Gamma\vert\cdot \vdash u : \tau'}{\Gamma\vert\Omega \vdash t\ u : \tau}$	$\{a_{\tau'} \vert \tau' \in T\}$	$\left(\begin{array}{l} \vec{\sigma}\vert\vec{\omega} \to (\tau' \Rightarrow \\ \vec{\sigma}\vert() \to \tau' \end{array} \right.$
	$\dfrac{\Gamma\vert\Omega_1 \vdash t : \tau' \twoheadrightarrow \tau \quad \Gamma\vert\Omega_2 \vdash u : \tau'}{\Gamma\vert\Omega_1, \Omega_2 \vdash t^>\ u : \tau}$	$\{a_{\tau'}^{\vec{\omega}_1,\vec{\omega}_2} \vert \tau' \in T$ and $\vec{\omega} = \vec{\omega}_1, \vec{\omega}_2\}$	$\left(\begin{array}{l} \vec{\sigma}\vert\vec{\omega}_1 \to (\tau' \Rightarrow \\ \vec{\sigma}\vert\vec{\omega}_2 \to \tau' \end{array} \right.$
	$\dfrac{\Gamma, x : \tau_1\vert\Omega \vdash t : \tau_2}{\Gamma\vert\Omega \vdash \lambda x.t : \tau_1 \Rightarrow \tau_2}$	$\{l_{\tau_1,\tau_2} \vert \tau = (\tau_1 \Rightarrow \tau_2)\}$	$(\vec{\sigma}, \tau_1\vert\vec{\omega} \to \tau$
	$\dfrac{\Gamma\vert\Omega, x : \tau_1 \vdash t : \tau_2}{\Gamma\vert\Omega \vdash \lambda^> x.t : \tau_1 \twoheadrightarrow \tau_2}$	$\{l^>_{\tau_1,\tau_2} \vert \tau = (\tau_1 \twoheadrightarrow \tau_2)\}$	$(\vec{\sigma}, \tau_1\vert\vec{\omega} \to \tau$

	Typing rule	$O_n(p\vert\vec{\sigma} \vdash \tau) = \ldots +$	$\alpha_o = (\ldots)$
System F (Section §8.4)	$\dfrac{x : \tau \in \Gamma}{n\vert\Gamma \vdash x : \tau}$	$\{v_i \vert i \in \vert\vec{\sigma}\vert_\tau\}$	$()$
	$\dfrac{n\vert\Gamma \vdash t : \tau' \Rightarrow \tau \quad n\vert\Gamma \vdash u : \tau'}{n\vert\Gamma \vdash t\ u : \tau}$	$\{a_{\tau'} \vert \tau' \in S_n\}$	$\begin{pmatrix} n\vert\vec{\sigma} \to \tau' \Rightarrow \tau \\ n\vert\vec{\sigma} \to \tau' \end{pmatrix}$
	$\dfrac{n\vert\Gamma, x : \tau_1 \vdash t : \tau_2}{n\vert\Gamma \vdash \lambda x.t : \tau_1 \Rightarrow \tau_2}$	$\{l_{\tau_1,\tau_2} \vert \tau = (\tau_1 \Rightarrow \tau_2)\}$	$(n\vert\vec{\sigma}, \tau_1 \to \tau_2)$
	$\dfrac{n\vert\Gamma \vdash t : \forall\tau_1 \quad \tau_2 \in S_n}{n\vert\Gamma \vdash t \cdot \tau_2 : \tau_1[\tau_2]}$	$\{A_{\tau_1,\tau_2} \vert \tau = \tau_1[\tau_2]\}$	$(n\vert\vec{\sigma} \to \forall\tau_1)$
	$\dfrac{n+1\vert wk(\Gamma) \vdash t : \tau}{n\vert\Gamma \vdash \Lambda t : \forall\tau}$	$\{\Lambda_{\tau'} \vert \tau = \forall\tau'\}$	$(n+1\vert wk(\vec{\sigma}) \to \tau')$

Pullbacks and equalisers are computed essentially as in Section §8.2. For example, the most general unifier of $M(\vec{x})$ and $M(\vec{y})$ maps M to $P(\vec{z})$ where $\vec{z}$ is the vector of common positions of $\vec{x}$ and $\vec{y}$, and P is a fresh metavariable of arity $\sigma_{\vec{z}}|\vec{\omega} \to \tau$.

8.4 Intrinsic polymorphic syntax

We present intrinsic System F, in the spirit of Hamana [2011]. The syntax of types in type variable context n is inductively generated as follows, following the De Bruijn level convention.

$$\frac{1 \leq i \leq n}{n \vdash i} \qquad \frac{n \vdash t \quad n \vdash u}{n \vdash t \Rightarrow u} \qquad \frac{n+1 \vdash t}{n \vdash \forall t}$$

Let $S : \mathbb{F}_m \longrightarrow \mathrm{Set}$ be the functor mapping n to the set S_n of types for system F taking free type variables in $\{1, \ldots, n\}$. In other words, $S_n = \{\tau | n \vdash \tau\}$. Intuitively, a metavariable arity $n|\vec{\sigma} \to \tau$ 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. This provides the underlying set of objects of the category $\mathcal{A}$ of arities. A term t in $n|\vec{\sigma} \to \tau$ considered as a variable context is intuitively a well-typed term of type τ potentially involving ground variables of type $\vec{\sigma}$ and type variables in $\{1, \ldots, n\}$.

A metavariable $M : (n|\sigma_1, \ldots, \sigma_p \to \tau)$ in the variable context $n'|\vec{\sigma}' \to \tau'$ must be supplied with

- a choice $(\eta_1, \ldots, \eta_n)$ of n distinct type variables among $\{1, \ldots n'\}$, such that $\tau[\vec{\eta}] = \tau'$, and
- an injective renaming $\vec{\sigma}[\vec{\eta}] \to \vec{\sigma}'$, i.e., a list of distinct positions $r_1, \ldots, r_p$ such that $\vec{\sigma}[\vec{\eta}] = \sigma'_{\vec{r}}$.

This defines the data for a morphism in $\mathcal{A}$ between $(n|\vec{\sigma} \to \tau)$ and $(n'|\vec{\sigma}' \to \tau')$. The intrinsic syntax of system F can then be specified as in Table 1. The induced GB-signature is pattern-friendly. For example, morphisms in $\mathcal{A}$ are easily seen to be monomorphic; we detail in Appendix §B the proof of the following statement.

Lemma 8.1. *$\mathcal{A}$ has finite connected limits.*

Pullbacks and equalisers in $\mathcal{A}$ are essentially computed as in Section §8.2, by computing the vector of common (value) positions. For example, given a metavariable M of arity $m|\vec{\sigma} \to \tau$, to unify $M(\vec{w}|\vec{x})$ with $M(\vec{y}|\vec{z})$, we compute the vector of common positions $\vec{p}$ between $\vec{w}$ and $\vec{y}$, and the vector of common positions $\vec{q}$ between $\vec{x}$ and $\vec{z}$. Then, the most general unifier maps M to the term $P(\vec{p}|\vec{q})$, where P is a fresh metavariable. Its arity $m'|\vec{\sigma}' \to \tau'$ is the only possible one for $P(\vec{p}|\vec{q})$ to be well-formed in the variable context $m|\vec{\sigma} \to \tau$, that is, m' is the size of $\vec{p}$, while $\tau' = \tau[p_i \mapsto i]$ and $\vec{\sigma}' = \sigma_{\vec{q}}[p_i \mapsto i]$.

9 RELATED WORK

First-order unification has been explained from a lattice-theoretic point of view by Plotkin Plotkin [1970], and later categorically analysed in Barr and Wells [1990]; Goguen [1989]; Rydeheard and Burstall [1988, Section 9.7] as coequalisers. However, there is little work on understanding pattern unification algebraically, with the notable exception of Vezzosi and Abel [2014], working with normalised terms of simply-typed λ-calculus. The present paper can be thought of as a generalisation of their work.

Although our notion of signature has a broader scope since we are not specifically focusing on syntax where variables can be substituted, our work is closer in spirit to the presheaf approach Fiore et al. [1999] to binding signatures than to the nominal approach Gabbay and Pitts [1999] in that everything is explicitly scoped: terms come with their support, metavariables always appear with their scope of allowed variables.

Nominal unification Urban et al. [2003] is an alternative to pattern unification where metavariables are not supplied with the list of allowed variables. Instead, substitution can capture variables.

Nominal unification explicitly deals with α-equivalence as an external relation on the syntax, and as a consequence deals with freshness problems in addition to unification problems.

Cheney Cheney [2005] shows that nominal unification and pattern unification problems are inter-translatable. As he notes, this result indirectly provides semantic foundations for pattern unification based on the nominal approach. In this respect, the present work provides a more direct semantic analysis of pattern unification, leading us to the generic algorithm we present, parameterised by a general notion of signature for the syntax.

10 CONCLUSION

We presented a generic unification algorithm for Miller's pattern fragment with its associated categorical semantics, parameterised by a new notion of signature for syntax with metavariables. In the future, we plan to a implement a reusable library based on this work. We also plan to see how this work applies to dependently-typed languages, going beyond polymorphic syntax. Finally, we are interesting in further extending the setting to cover unification modulo equations, or linear syntax without restriction on the order the variables are used.

REFERENCES

Peter Aczel. 2016. A general church-rosser theorem, 1978. *Unpublished note. http://www. ens-lyon. fr/LIP/REWRITING/MISC/AGeneralChurch-RosserTheorem. pdf. Accessed* (2016), 10–07.

Jiri Adámek, Francis Borceux, Stephen Lack, and Jirí Rosicky. 2002. A classification of accessible categories. *Journal of Pure and Applied Algebra* 175, 1 (2002), 7–30. https://doi.org/10.1016/S0022-4049(02)00126-3 Special Volume celebrating the 70th birthday of Professor Max Kelly.

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

Thorsten Altenkirch and Peter Morris. 2009. Indexed Containers. In *Proceedings of the 24th Annual IEEE Symposium on Logic in Computer Science, LICS 2009, 11-14 August 2009, Los Angeles, CA, USA*. IEEE Computer Society, 277–285. https://doi.org/10.1109/LICS.2009.33

Michael Barr and Charles Wells. 1990. *Category Theory for Computing Science.* Prentice-Hall, Inc., USA.

R. Blackwell, G.M. Kelly, and A.J. Power. 1989. Two-dimensional monad theory. *Journal of Pure and Applied Algebra* 59, 1 (1989), 1–41. https://doi.org/10.1016/0022-4049(89)90160-6

James Cheney. 2005. Relating nominal and higher-order pattern unification. In *Proceedings of the 19th international workshop on Unification (UNIF 2005)*. LORIA research report A05, 104–119.

N. G. De Bruijn. 1972. Lambda-Calculus Notation with Nameless Dummies, a Tool for Automatic Formula Manipulation, with Application to the Church-Rosser Theorem. *Indagationes Mathematicae* 34 (1972), 381–392.

Jana Dunfield and Neelakantan R. Krishnaswami. 2019. Sound and complete bidirectional typechecking for higher-rank polymorphism with existentials and indexed types. *Proc. ACM Program. Lang.* 3, POPL (2019), 9:1–9:28. https://doi.org/10.1145/3290322

Marcelo Fiore, Gordon Plotkin, and Daniele Turi. 1999. Abstract Syntax and Variable Binding. In *Proc. 14th Symposium on Logic in Computer Science* IEEE.

M. P. Fiore and C.-K. Hur. 2010. Second-order equational logic. In *Proceedings of the 19th EACSL Annual Conference on Computer Science Logic (CSL 2010)*.

Murdoch J. Gabbay and Andrew M. Pitts. 1999. A New Approach to Abstract Syntax Involving Binders. In *Proc. 14th Symposium on Logic in Computer Science* IEEE.

Joseph A. Goguen. 1989. What is Unification? - A Categorical View of Substitution, Equation and Solution. In *Resolution of Equations in Algebraic Structures, Volume 1: Algebraic Techniques*. Academic, 217–261.

Warren D. Goldfarb. 1981. The undecidability of the second-order unification problem. *Theoretical Computer Science* 13, 2 (1981), 225–230. https://doi.org/10.1016/0304-3975(81)90040-2

John W. Gray. 1966. Fibred and Cofibred Categories. In *Proceedings of the Conference on Categorical Algebra*, S. Eilenberg, D. K. Harrison, S. MacLane, and H. Röhrl (Eds.). Springer Berlin Heidelberg, Berlin, Heidelberg, 21–83.

Makoto Hamana. 2011. Polymorphic Abstract Syntax via Grothendieck Construction.

Gérard P. Huet. 1975. A Unification Algorithm for Typed lambda-Calculus. *Theor. Comput. Sci.* 1, 1 (1975), 27–57. https://doi.org/10.1016/0304-3975(75)90011-0

André Joyal and Ross Street. 1993. Pullbacks equivalent to pseudopullbacks. *Cahiers de Topologie et Géométrie Différentielle Catégoriques* XXXIV, 2 (1993), 153–156.

Saunders Mac Lane. 1998. *Categories for the Working Mathematician* (2nd ed.). Number 5 in Graduate Texts in Mathematics. Springer.

Dale Miller. 1991. A Logic Programming Language with Lambda-Abstraction, Function Variables, and Simple Unification. *J. Log. Comput.* 1, 4 (1991), 497–536. https://doi.org/10.1093/logcom/1.4.497

Gordon D. Plotkin. 1970. A Note on Inductive Generalization. *Machine Intelligence* 5 (1970), 153–163.

Jeff Polakow and Frank Pfenning. 2000. Properties of Terms in Continuation-Passing Style in an Ordered Logical Framework. In *2nd Workshop on Logical Frameworks and Meta-languages (LFM'00)*, Joëlle Despeyroux (Ed.). Santa Barbara, California. Proceedings available as INRIA Technical Report.

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

J. A. Robinson. 1965. A Machine-Oriented Logic Based on the Resolution Principle. *J. ACM* 12, 1 (jan 1965), 23–41. https://doi.org/10.1145/321250.321253

David E. Rydeheard and Rod M. Burstall. 1988. *Computational category theory.* Prentice Hall.

Christian Urban, Andrew Pitts, and Murdoch Gabbay. 2003. Nominal Unification. In *Computer Science Logic*, Matthias Baaz and Johann A. Makowsky (Eds.). Springer Berlin Heidelberg, Berlin, Heidelberg, 513–527.

Andrea Vezzosi and Andreas Abel. 2014. A Categorical Perspective on Pattern Unification. *RISC-Linz* (2014), 69.

Jinxu Zhao, Bruno C. d. S. Oliveira, and Tom Schrijvers. 2019. A mechanical formalization of higher-ranked polymorphic type inference. *Proc. ACM Program. Lang.* 3, ICFP (2019), 112:1–112:29. https://doi.org/10.1145/3341716

A PROOF OF LEMMA 3.20

Notation A.1. Given a functor $F : I \longrightarrow \mathscr{B}$, we denote the limit (resp. colimit) of F by $\int_{i:I} F(i)$ or $\lim F$ (resp. $\int^{i:I} F(i)$ or colim F) and the canonical projection $\lim F \longrightarrow Fi$ by p_i for any object i of I.

This section is dedicated to the proof of the following lemma.

Lemma A.2. *Given a GB-signature $S = (\mathcal{A}, O, \alpha)$ such that $\mathcal{A}$ has finite connected limits, F_S restricts as an endofunctor on the full subcategory $\mathscr{C}$ of $[\mathcal{A}, \mathrm{Set}]$ consisting of functors preserving finite connected limits if and only if each $O_n \in \mathscr{C}$, and $\alpha : \int J \longrightarrow \mathcal{A}$ preserves finite limits.*

We first introduce a bunch of intermediate lemmas.

Lemma A.3. *If $\mathscr{B}$ is a small category with finite connected limits, then a functor $G : \mathscr{B} \longrightarrow \mathrm{Set}$ preserves those limits if and only if $\int \mathscr{B}$ is a coproduct of filtered categories.*

Proof. This is a direct application of Adámek et al. [2002, Theorem 2.4 and Example 2.3.(iii)]. □

Corollary A.4. *Assume $\mathcal{A}$ has finite connected limits. Then $J : \mathbb{N} \times \mathcal{A} \longrightarrow \mathrm{Set}$ preserves finite connected limits if and only if each $O_n : \mathcal{A} \longrightarrow \mathrm{Set}$ does.*

Proof. This follows from $\int J \cong \coprod_{n \in \mathbb{N}} \coprod_{j \in \{1,\ldots,n\}} \int O_n$. □

Lemma A.5. *Let $F : \mathscr{B} \longrightarrow \mathrm{Set}$ be a functor. For any functor $G : I \longrightarrow \int F$, denoting by H the composite functor $I \xrightarrow{G} \int F \longrightarrow \mathscr{B}$, there exists a unique $x \in \lim(F \circ H)$ such that $Gi = (Hi, p_i(x))$.*

Proof. $\int F$ is isomorphic to the opposite of the comma category y/F, where $y : \mathscr{B}^{op} \longrightarrow [\mathscr{B}, \mathrm{Set}]$ is the Yoneda embedding. The statement follows from the universal property of a comma category. □

Lemma A.6. *Let $F : \mathscr{B} \longrightarrow \mathrm{Set}$ and $G : I \longrightarrow \int F$ such that F preserves the limit of $H : I \xrightarrow{G} \int F \longrightarrow \mathscr{B}$. Then, there exists a unique $x \in F \lim H$ such that $Gi = (Hi, Fp_i(x))$ and moreover, $(\lim H, x)$ is the limit of G.*

Proof. The unique existence of $x \in F \lim H$ such that $Gi = (Hi, Fp_i(x))$ follows from Lemma A.5 and the fact that F preserves $\lim H$. Let $\mathscr{C}$ denote the full subcategory of $[\mathscr{B}, \mathrm{Set}]$ of functors preserving $\lim G$. Note that $\int F$ is isomorphic to the opposite of the comma category K/F, where $K : \mathscr{B}^{op} \longrightarrow \mathscr{C}$ is the Yoneda embedding, which preserves colim G, by an argument similar to the proof of Lemma 3.23. We conclude from the fact that the forgetful functor from a comma category L/R to the product of the categories creates colimits that L preserve. □

Corollary A.7. *Let I be a small category, $\mathscr{B}$ and $\mathscr{B}'$ be categories with I-limits (i.e., limits of any diagram over I). Let $F : \mathscr{B} \longrightarrow \mathrm{Set}$ be a functor preserving those colimits. Then, $\int F$ has I-limits, preserved by the projection $\int F \longrightarrow \mathscr{B}$. Moreover, a functor $G : \int F \longrightarrow \mathscr{B}'$ preserves them if and only if for any $d : I \longrightarrow \mathscr{B}$ and $x \in F \lim d$, the canonical morphism $G(\lim d, x) \longrightarrow \int_{i:I} G(d_i, Fp_i(x))$ is an isomorphism.*

Proof. By Lemma A.6, a diagram $d' : I \longrightarrow \int F$ is equivalently given by $d : I \longrightarrow \mathscr{B}$ and $x \in F \lim d$, recovering d' as $d'_i = (d_i, Fp_i(x))$, and moreover $\lim d' = (\lim d, x)$. □

Corollary A.8. *Assuming that $\mathcal{A}$ has finite connected limits and each O_n preserves finite connected limits, the finite limit preservation on $\alpha : \int J \longrightarrow \mathcal{A}$ of Lemma A.2 can be reformulated as follows: given a finite connected diagram $d : D \longrightarrow \mathcal{A}$ and element $o \in O_n(\lim d)$, the following canonical morphism is an isomorphism*

$$\bar{o}_j \longrightarrow \int_{i:D} \overline{o\{p_i\}}_j$$

for any $j \in \{1, \dots, n\}$.

Proof. This is a direct application of Corollary A.7 and Corollary A.4. □

Lemma A.9 (Limits commute with dependent pairs). *Given functors $K : I \longrightarrow \mathrm{Set}$ and $G : \int K \longrightarrow \mathrm{Set}$, the following canonical morphism is an isomorphism*

$$\int_{i:I} \coprod_{x \in Ki} G(i, x) \longrightarrow \coprod_{\alpha \in \lim K} \int_{i:I} G(i, p_i(\alpha))$$

Proof. It is straightforward to check that both sets share the same universal property. □

Proof of Lemma A.2. Let $d : I \longrightarrow \mathcal{A}$ be a finite connected diagram and X be a functor preserving finite connected limits. Then,

$$\begin{aligned}
\int_{i:I} F(X)_{d_i} &= \int_{i:I} \coprod_n \coprod_{o \in O_n(d_i)} X_{\bar{o}_1} \times \cdots \times X_{\bar{o}_n} \\
&\cong \coprod_n \int_{i:I} \coprod_{o \in O_n(d_i)} X_{\bar{o}_1} \times \cdots \times X_{\bar{o}_n} && \text{(Coproducts commute with connected limits)} \\
&\cong \coprod_n \coprod_{o \in \int_i O_n(d_i)} \int_{i:I} X_{\overline{p_i(o)}_1} \times \cdots \times X_{\overline{p_i(o)}_n} && \text{(By Lemma A.9)} \\
&\cong \coprod_n \coprod_{o \in \int_i O_n(d_i)} \int_{i:I} X_{\overline{p_i(o)}_1} \times \cdots \times \int_{i:I} X_{\overline{p_i(o)}_n} && \text{(By commutation of limits)}
\end{aligned}$$

Thus, since X preserves finite connected limits by assumption,

$$\int_i F(X)_{d_i} = \coprod_n \coprod_{o \in \int_i O_n(d_i)} X_{\int_{i:I} \overline{p_i(o)}_1} \times \cdots \times X_{\int_{i:I} \overline{p_i(o)}_n} \tag{3}$$

Now, let us prove the only if statement first. Assuming that $\alpha : \int J \longrightarrow \mathcal{A}$ and each O_n preserves finite connected limits. Then,

$$\begin{aligned}
\int_i F(X)_{d_i} &\cong \coprod_n \coprod_{o \in \int_i O_n(d_i)} X_{\int_{i:I} \overline{p_i(o)}_1} \times \cdots \times X_{\int_{i:I} \overline{p_i(o)}_n} && \text{(By Equation (3))} \\
&\cong \coprod_n \coprod_{o \in O_n(\lim d)} X_{\int_{i:I} \overline{o\{p_i\}}_1} \times \cdots \times X_{\int_{i:I} \overline{o\{p_i\}}_n} && \text{(By assumption on } O_n\text{)} \\
&\cong \coprod_n \coprod_{o \in O_n(\lim d)} X_{\overline{o}_1} \times \cdots \times X_{\overline{o}_n} && \text{(By Corollary A.8)} \\
&= F(X)_{\lim d}
\end{aligned}$$

Conversely, let us assume that F restricts to an endofunctor on $\mathscr{C}$. Then, $F(1) = \coprod_n O_n$ preserves finite connected limits. By Lemma A.3, each O_n preserves finite connected limits. By Corollary A.8, it is enough to prove that given a finite connected diagram $d : D \longrightarrow \mathcal{A}$ and element $o \in O_n(\lim d)$, the following canonical morphism is an isomorphism

$$\overline{o}_j \longrightarrow \int_{i:D} \overline{o\{p_i\}}_j$$

Now, we have

$$\begin{aligned}
\int_{i:I} F(X)_{d_i} &\cong F(X)_{\lim d} && \text{(By assumption)} \\
&= \coprod_n \coprod_{o \in O_n(\lim d)} X_{\overline{o}_1} \times \cdots \times X_{\overline{o}_n}
\end{aligned}$$

On the other hand,

$$\begin{aligned}
\int_{i:I} F(X)_{d_i} &\cong \coprod_n \coprod_{o \in \int_i O_n(d_i)} X_{\int_{i:I} \overline{p_i(o)}_1} \times \cdots \times X_{\int_{i:I} \overline{p_i(o)}_n} && \text{(By Equation (3))} \\
&= \coprod_n \coprod_{o \in O_n(\lim d)} X_{\int_{i:I} \overline{o\{p_i\}}_1} \times \cdots \times X_{\int_{i:I} \overline{o\{p_i\}}_n} && \text{(}O_n\text{ preserves finite connected limits)}
\end{aligned}$$

It follows from those two chains of isomorphisms that each function $X_{\overline{o}_j} \longrightarrow X_{\int_{i:I} \overline{o\{p_i\}}_j}$ is a bijection, or equivalently (by the Yoneda Lemma), that $\mathscr{C}(K\overline{o}_j, X) \longrightarrow \mathscr{C}(K \int_{i:I} \overline{o\{p_i\}}_j, X)$ is an isomorphism. Since the Yoneda embedding is fully faithful, $\overline{o}_j \longrightarrow \int_{i:D} \overline{o\{p_i\}}_j$ is an isomorphism. □

B PROOF OF LEMMA 8.1

In this section, we show that the category $\mathcal{A}$ of arities for System F (Section §8.4) has finite connected limits. First, note that $\mathcal{A}$ is the op-lax colimit of the functor from $\mathbb{F}_m$ to the category of small categories mapping n to $\mathbb{F}_m[S_n] \times S_n$. Let us introduce the category $\mathcal{A}'$ whose definition follows that of $\mathcal{A}$, but without the output types: objects are pairs of a natural number n and an element of S_n. Formally, this is the op-lax colimit of $n \mapsto \mathbb{F}_m[S_n]$.

Lemma B.1. *$\mathcal{A}'$ has finite connected limits, and the projection functor $\mathcal{A}' \longrightarrow \mathbb{F}_m$ preserves them.*

Proof. The crucial point is that $\mathcal{A}'$ is not only op-fibred over $\mathbb{F}_m$ by construction, it is also fibred over $\mathbb{F}_m$. Intuitively, if $\vec{\sigma} \in \mathbb{F}_m[S_n]$ and $f : n' \longrightarrow n$ is a morphism in $\mathbb{F}_m$, then $f_! \vec{\sigma} \in \mathbb{F}_m[S_{n'}]$ is essentially $\vec{\sigma}$ restricted to elements of S_n that are in the image of S_f. We can now apply Gray [1966, Corollary 4.3], since each $\mathbb{F}_m[S_n]$ has finite connected limits. □

We are now ready to prove that $\mathcal{A}$ has finite connected limits.

Proof of Lemma 8.1. Since $S : \mathbb{F}_m \to \mathrm{Set}$ preserves finite connected limits, $\int S$ has finite connected limits and the projection functor to $\mathbb{F}_m$ preserves them by Corollary A.7.

Now, the 2-category of small categories with finite connected limits and functors preserving those between them is the category of algebras for a 2-monad on the category of small categories Blackwell et al. [1989]. Thus, it includes the weak pullback of $\mathcal{A}' \longrightarrow \mathbb{F}_m \longleftarrow \int S$. But since $\int S \longrightarrow \mathbb{F}_m$ is a fibration, and thus an isofibration, by Joyal and Street [1993] this weak pullback can be computed as a pullback, which is $\mathcal{A}$. □