Semantics of pattern unification

ANONYMOUS AUTHOR(S)

It is well-known that first-order unification corresponds to the construction of equalisers in a (multi-sorted) Lawvere theory. We show that Miller's decidable *pattern* fragment of second-order unification can be interpreted similarly; the involved Lawvere theories are no longer freely generated by operations. To illustrate our semantic analysis, we present a generic unification algorithm implemented in Agda. The syntax with metavariables given as input of the algorithm is parameterised by a notion of signature generalising binding signatures, covering a wide range of examples, including ordered λ-calculus, (intrinsic) polymorphic syntax such as System F, and of course Miller's original application, normalised simply-typed λ-calculus.

ACM Reference Format:
Anonymous Author(s). 2025. Semantics of pattern unification. *Proc. ACM Program. Lang.* 1, POPL, Article 1 (January 2025), 32 pages.

1 Introduction

Unification deals with a base syntax enriched with *metavariables*. Given two *enriched* terms, the goal is to find the most general unifier, that is, the most general instantiation of those metavariables that makes the two terms equal.

In this work, we focus on *pattern unification,* a decidable fragment of higher-order unification where in the enriched syntax, metavariables are applied to *pattern*s, that is, to lists of distinct variables. This is especially useful for languages with variable binding, so that metavariables can depend on bound variables. Pattern unification was introduced by Miller [21] for simply-typed λ-calculus modulo β and η. His algorithm either returns the most general unifier of two given terms, or detects that the two terms are not unifiable. It was categorically interpreted by Vezzosi and Abel [31] as computing certain coequalisers in a category of metavariable contexts and substitutions. Their starting point is that patterns correspond to injective renamings can be seen as morphisms in a suitable category of contexts. Like Miller, they get rid of the β and η-equations by working on the syntax of normalised λ-terms (which is notably equation free). In their conclusion they sketched a track to extend their work for a suitable class of languages:

"Instead of [the category of patterns] we can consider a generic category *Ctx* having all the pullbacks and equalisers and whose arrows are monomorphisms. And instead of the grammar of [normal forms of simply-typed λ-calculus] we can use an arbitrary one defined by a family of operators [...] as long as they are functorial with respect to *Ctx*."

The present work proposes a generalisation as envisioned by Vezzosi-Abel: we define a class of languages for which unification problems can be stated and solved by a recursive procedure much similar to Miller's original pattern unification algorithm. Each language is specified by the following data:

- a small category $\mathcal{A}$ with equalisers and pullbacks, whose objects are called *environments*, and whose morphisms, called *patterns*, are all monomorphic;
- for each natural number n,
  - a functorial assignment $O_n(-)$ mapping an environment a to the set of operation symbols available in the context a;

2025. ACM 2475-1421/2025/1-ART1
https://doi.org/

- a functorial assignment α mapping[1] an environment a and an operation symbol $o \in O_n(a)$ to a the list of *input environments* $\alpha_o = (\overline{o}_1, \ldots, \overline{o}_n)$;
- such that O_n and α preserve equalisers and pullbacks.

As we shall see, in practice, types are specified through the notion of environments (first item), while terms are specified through the operation symbols and their arities (second item).

Syntax generated by a specification. The base syntax is generated by the following single rule, where $a \vdash t$ means that the term t is well-formed in the environment a.

$$\forall o \in O_n(a) \frac{\overline{o}_1 \vdash t_1 \quad \ldots \quad \overline{o}_n \vdash t_n}{a \vdash o(t_1, \ldots, t_n)}$$

The syntax is functorial in the sense that given any pattern $f \in \hom(a, b)$ and term $a \vdash t$, we define by recursion on the structure of t a term $b \vdash t\{f\}$, such that $t\{f\}\{g\} = t\{g \circ f\}$ and $t\{id_a\} = t$.

The enriched syntax with metavariables involves a well-formed judgement of the shape $\Gamma; a \vdash t$, where Γ is a *metavariable context* that keeps track of the *metatypes* of metavariables, which are just environments. A metavariable context Γ is a list of metavariable symbols with their associated metatypes. The enriched syntax is generated by two rules, one for operations, and one for metavariables.

$$\forall \Gamma \forall o \in O_n(a) \frac{\Gamma; \overline{o}_1 \vdash t_1 \quad \ldots \quad \Gamma; \overline{o}_n \vdash t_n}{\Gamma; a \vdash o(t_1, \ldots, t_n)} \qquad \frac{M : m \in \Gamma \quad x \in \hom(m, n)}{\Gamma; n \vdash M(x)}$$

Note that contrary to the traditional definition of the pattern fragment, where the notion of pattern is derived from the notion of variable, in our setting, patterns are built-in (they are morphisms in $\mathcal{A}$) and there is no built-in notion of "variables".

The enriched syntax is functorial with respect to the category of renamings – we define $M(x)\{f\}$ as $M(f \circ x)$ – but is moreover functorial with respect to metavariable substitutions. Here, a metavariable substitution σ between two metavariable contexts Γ and Γ' assigns to each metavariable declaration $M : m$ in Γ a term $\Gamma'; m \vdash \sigma_m$. Then, given $\Gamma; a \vdash t$, the substituted term $\Gamma'; a \vdash t[\sigma]$ is defined by recursion on the structure of t. Metavariable substitutions compose and are morphisms in a category whose objects are patterns.

Typed languages. Let us explain how typed languages are accounted, although our notion of specification does not incorporate any explicit notion of types. In Vezzosi-Abel's work, a typing judgement has 4 entities: a metacontext Γ, an object context $\vec{\sigma}$ which is a list of simple types (one for each available free variable), a term t and its type τ. We can handle this type system by merging the context $\vec{\sigma}$ and the type τ into a single entity $\vec{\sigma} \to \tau$, which we call an environment. In other words, in our setting, $\Gamma; \vec{\sigma} \vdash t : \tau$ would be denoted by $\Gamma; \vec{\sigma} \to \tau \vdash t$, see Section §7.2 for more details.

Stating unification. Let us explain how unification is stated in our setting. First, a unifier of two terms t and u in the same metacontext Γ and environment a is a metavariable context Δ and a metavariable substitution $\sigma : \Gamma \to \Delta$ such that $t[\sigma] = u[\sigma]$. It is the most general unifier (abbreviated mgu) if any other unifier δ uniquely factors through σ. The main point of pattern unification (that holds for any of our languages) is that either the mgu exists, or there is no unifier. Moreover, there is a recursive procedure which computes the mgu or detects the impossibility of unification.

We bring a slight change of perspective with respect to the traditional presentation of unification as a partial algorithm that computes the mgu.

[1] See Definition 3.13 for a formal definition, involving the construction of a category of elements.

A slight change of perspective allows us to get rid of this partiality: we add a formal error metavariable context $\bot$ and a single formal term $\bot; a \vdash !$ for all environments a, so that we get a unique metavariable substitution $!_\Gamma$ from any metacontext Γ to $\bot$ which unifies any pair of terms. If two terms are not unifiable in the traditional sense, ! is the mgu. If $\sigma : \Gamma \longrightarrow \Delta$ is the mgu in the traditional sense, then it is still the mgu in this extended setting, because $!_\Gamma$ uniquely factors as $!_\Delta \circ \sigma$. From this, it follows that the unification algorithm is total.

Comparison with Vezzosi-Abel's vision. We mentioned above our implicit treatment of types and failure that makes our abstract setting simpler. On top of that, the key additional ingredient that we introduce is the preservation of equalisers and pullbacks. This is crucial to ensure that the above main point is true.

Scope of our class of languages. The first thing to note is that our class of languages includes any syntax specified by a multi-sorted binding signature [11]: we detail simply-typed λ-calculus (without β and η-equation) in Section §7.1, and the case of mono-sorted binding signature is explained in Example 2.6. Let us discuss a simple example, leaving the above mentioned issue of types aside for simplicity: pure λ-calculus, without β or η equation. Using De Bruijn encoding, an environment is a natural number indicating the number of available free variables. In the environment n, we have n nullary available operation symbols – one for each variable – so that $O_0(n) = \{1, \ldots, n\}$, one unary operation $O_1(n) = \{abs\}$ and one binary operation $O_2(n) = \{app\}$. Following Vezzosi-Abel's analysis, a pattern, as a morphism between environments, n and m is defined as an injective map from $\{1, \ldots, n\}$ and $\{1, \ldots, m\}$. Note that there is an alternative, more restrictive choice of notion of patterns: we could require that the injective map is monotone. In some way, this is equivalent to enforcing that arguments of metavariables are sets of variables. The two choices yield two valid different languages, that share the same base syntax (forgetting about the functoriality). Those alternatives are not specific to pure λ-calculus but are equally possible for any multi-sorted binding signature.

Our class of languages also includes syntaxes that do not support substitution of object variables with terms. This is in particular the case of the syntax of normalised simply-typed λ-terms, detailed in Section §7.2, and taken as a starting point by Vezzosi-Abel. Indeed, replacing an applied variable with a λ-abstracted term does not yield a normal form.

Our class of languages also accounts for languages where terms bind type variables, such as system F (Section §7.4). In another direction, we can handle certain kind of constraints on the variables in the context: in Section §7.3, we detail the calculus underlying ordered linear logic described in Polakow and Pfenning [26], where the context is split in two parts, one of which includes variables that must occur exactly once and in the same order as they occur in that context.

All the examples are summarised in Table 1 in Section §7, where the traditional presentation of each calculus is translated into our notion of specification.

Agda implementation. We illustrate our theoretical analysis with an Agda implementation of a generic unification algorithm (without mechanisation of the correctness proof), of which will show the most relevant parts. 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. (The proper formalisation of category theory in proof assistants remains a significant challenge in its own right.) Furthermore, we disable the termination checker and provide instead a termination proof on paper in Section §6.1. Even used purely as a programming language, dependent types are very helpful in structuring the implementation

Plan of the paper

In section §2, we present our generic pattern unification algorithm, parameterised by our notion of specification. 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. Examples of specifications are given in Section §7, and related work is finally discussed in Section §8 before the conclusion, 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 $A + B$, 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 $A_j \to \coprod_{i \in I} A_i$ are typically denoted by in_j. 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 Section §2.1, we start by describing a pattern unification algorithm for pure λ-calculus, summarised in Figure 4. We claim no originality here; minor variants of the algorithm can be found in the litterature: it serves mainly as an introduction to the generic algorithm presented in Section §2.2 and summarised in Figure 5.

2.1 An example: pure λ-calculus.

Consider the syntax of pure λ-calculus extended with pattern metavariables. We list the Agda code in Figure 1, together with a corresponding presentation as inductive rules generating the syntax. We write $\Gamma; n \vdash t$ to mean t is a well-formed λ-term in the context $\Gamma; n$, consisting of two parts:

1. a metavariable context (or *metacontext*) Γ, which is either a formal error context $\perp$, or a *proper* context, as a list $(M_1 : m_1, \dots, M_p : m_p)$, of metavariable declarations specifying metavariable symbols M_i together with their arities, i.e, their number of arguments m_i;
2. an environment, 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 [9]. In Agda, variables in the environment n consist of elements of Fin n, the type of natural numbers between[2] 1 and n.

The error metacontext $\perp$ will prove useful to handle failure in the unification algorithm. Traditionally, unification is presented as a partial algorithm, since unifiers may not exist. Instead of modelling partiality with some kind of error monad, we instead make our unification algorithm total by adding a formal error, so that a metacontext is either a proper metacontext or a formal error metacontext, and the unification algorithm either returns a proper substitution or an error substitution. Our approach to failure actually arises from the categorical semantics (see Section §3.1).

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

Fig. 1. Syntax of λ-calculus (Section §2.1)

```
MetaContext· = List ℕ                         hom : ℕ → ℕ → Set
MetaContext = Maybe MetaContext·              hom m n = Vec (Fin n) m

data Tm : MetaContext → ℕ → Set
Tm· Γ n = Tm ⌊ Γ ⌋ n

data Tm where
  App· : ∀ {Γ n} → Tm· Γ n → Tm· Γ n → Tm· Γ n
  Lam· : ∀ {Γ n} → Tm· Γ (1 + n) → Tm· Γ n
  Var· : ∀ {Γ n} → Fin n → Tm· Γ n
  _(_) : ∀ {Γ n m} → m ∈ Γ → hom m n → Tm· Γ n
  ! : ∀ {n} → Tm ⊥ n
```

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

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

$$\frac{}{\bot; a \vdash !}$$

```
App : ∀ {Γ n} → Tm Γ n →               Lam : ∀ {Γ n} → Tm Γ (1 + n) →        Var : ∀ {Γ n} → Fin n →
        Tm Γ n → Tm Γ n                       Tm Γ n                                Tm Γ n
App {⊥} ! ! = !                         Lam {⊥} ! = !                          Var {⊥} i = !
App {⌊ Γ ⌋} t u = App· t u              Lam {⌊ Γ ⌋} t = Lam· t                 Var {⌊ Γ ⌋} i = Var· i
```

In the inductive rules, we use the bold face $\mathbf{\Gamma}$ for any proper metacontext. In the Agda code, we adopt a nameless encoding of proper metacontexts: they are mere lists of metavariable arities, and metavariables are referred to by their index in the list. The type of metacontexts MetaContext is formally defined as Maybe (List ℕ), where Maybe X is an inductive type with an error constructor $\bot$ and a *proper* constructor $\lfloor - \rfloor$ taking as argument an element of type X. Therefore, $\mathbf{\Gamma}$ typically translates into $\lfloor \Gamma \rfloor$ in the implementation. To alleviate notations, we also adopt a dotted convention in Agda to mean that a proper metacontext is involved. For example, MetaContext· and Tm· Γ n are respectively defined as List ℕ and Tm $\lfloor \Gamma \rfloor$ n.

The last term constructor ! builds a well-formed term in any error context $\bot; n$. We call it an *error* term: it is the only one available in such contexts. *Proper* terms, i.e., terms well-formed in a proper metacontext, are built from application, λ-abstraction and variables: they generate the (proper) syntax of λ-calculus. Note that ! cannot occur as a sub-term of a proper term.

Remark 2.1. The names of constructors of λ-calculus for application, λ-abstraction, and variables, are dotted to indicate that they are only available in a proper metacontext. "Improper" versions of those, defined in any metacontext, are also implemented in the obvious way, coinciding with the constructors in a proper context, or returning ! in the error context.

Let us focus on the penultimate constructor, building a metavariable application in the context $\mathbf{\Gamma}; n$. The argument of type $m \in \mathbf{\Gamma}$ is an index of any element m in the list $\mathbf{\Gamma}$. In the pattern fragment, a metavariable of arity m can be applied to a list of size m consisting of distinct variables in the environment n, that is, natural numbers between 1 and n. We denote by $\mathrm{hom}(m, n)$ this set of lists. To make the Agda implementation easier, we did not enforce the uniqueness restriction in the definition of hom m n. However, our unification algorithm is guaranteed to produce correct outputs only if this constraint is satisfied in the inputs.

The Agda implementation of metavariable substitutions for λ-calculus is listed in the first box of Figure 2. We call a substitution *successful* if it targets a proper metacontext, *proper* if the domain is proper. Note that any successful substitution is proper because there is only one metavariable

Fig. 2. Metavariable substitution

```
- Proper substitutions                - Successful substitutions
Γ ·⟶ Δ = ⌊ Γ ⌋ ⟶ Δ                    Γ ·⟶· Δ = ⌊ Γ ⌋ ⟶ ⌊ Δ ⌋
data _⟶_ where
   [] : ∀ {Δ} → ([] ·⟶ Δ )
   _,_ : ∀ {Γ Δ m} → Tm Δ m → (Γ ·⟶ Δ) → (m :: Γ ·⟶ Δ)
   1⊥ : ⊥ ⟶ ⊥
```

λ-calculus (Section §2.1)

```
_[_]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 = nth σ M { x }
! [ 1⊥ ]t = !
```

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

```
_[_]s : ∀ {Γ Δ E} → (Γ ⟶ Δ) → (Δ ⟶ E) → (Γ ⟶ E)
[] [ σ ]s = []
(t , δ) [ σ ]s = t [ σ ]t , δ [ σ ]s
1⊥ [ 1⊥ ]s = 1⊥
```

$$\frac{\delta : \Gamma \to \Delta \qquad \sigma : \Delta \to E}{\underbrace{\delta[\sigma]}_{M \mapsto \delta_M[\sigma]} \; : \Gamma \to E}$$

Generic syntax (Section §2.2)

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

(Rigid· o δ) [ σ ]t = Rigid o (δ [ σ ]s)
M ( x ) [ σ ]t = nth σ M { x }
! [ 1⊥ ]t = !

[] [ σ ]s = []
(t , δ) [ σ ]s = t [ σ ]t , δ [ σ ]s
1⊥ [ 1⊥ ]s = 1⊥
```

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

$$\frac{\delta : \Gamma \to \Delta \qquad \sigma : \Delta \to E}{\underbrace{\delta[\sigma]}_{M \mapsto \delta_M[\sigma]} \; : \Gamma \to E}$$

substitution $1_\perp$ from the error context: it is a formal identity substitution, targeting itself. A *metavariable substitution* $\sigma : \mathbf{\Gamma} \longrightarrow \Delta$ from a proper context assigns to each metavariable M of arity m in $\mathbf{\Gamma}$ a term $\Delta; m \vdash \sigma_M$.

This assignment extends (through a recursive definition) to any term $\mathbf{\Gamma}; n \vdash t$, yielding a term $\Delta; n \vdash t[\sigma]$. Note that the congruence cases involve improper versions of the operations (Remark 2.1), as the target metacontext may not be proper. The base case is $M(x_1, \ldots, x_m)[\sigma] = \sigma_M\{x\}$, where $-\{x\}$ is variable renaming, defined by recursion. Renaming a λ-abstraction requires extending the renaming x : hom $p\ q$ to $x \uparrow$: hom $(p + 1)\ (q + 1)$ to take into account the additional bound variable $\underline{p + 1}$, which is renamed to $\underline{q + 1}$. Then, $(\lambda t)\{x\}$ is defined as $\lambda(t\{x \uparrow\})$. While metavariable substitutions change the metacontext of the substituted term, renamings change the environment.

The identity substitution $1_{\mathbf{\Gamma}} : \mathbf{\Gamma} \to \mathbf{\Gamma}$ is defined by the term $M(1, \ldots, m)$ for each metavariable declaration $M : m \in \mathbf{\Gamma}$. The composition $\delta[\sigma] : \mathbf{\Gamma_1} \to \Gamma_3$ of two substitutions $\delta : \mathbf{\Gamma_1} \to \Gamma_2$ 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* (later abbreviated as mgu) 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 $\delta = \sigma[\delta']$.

Remark 2.2. Given a metacontext Γ, there is a single *terminal* substitution $!_s : \Gamma \to \bot$, which maps any metavariable to the only available term ! if Γ is proper, or is the identity substitution $1_\bot$ otherwise. Any term substituted by $!_s$ yields the error term !, since it is the only one in the metacontext $\bot$. As a consequence,

- $!_s : \Gamma \to \bot$ is uniquely factored by any other substitution $\sigma : \Gamma \to \Delta$ as the composition of σ with $!_s : \Delta \to \bot$
- $!_s$ unifies any pair of terms.

Remark 2.3. Because of the additional error context, our notion of unification differs from the standard presentation, which is recovered by focusing only on successful substitutions. However, it follows from Remark 2.2 that mgus in the standard setting are still mgus in our setting. Moreover, when there is no successful unifier, the terminal substitution is a mgu.

The main property of pattern unification is that the mgu of any pair of terms exists as soon as there exists a unifier. Remark 2.3 shows that we can actually get rid of the latter condition: the non-existence of unifiers (for example, when unifying $t_1\ t_2$ with λu) is restated as $!_s$ being the mgu. Accordingly, our implementation does not explicitly fail. Given two terms $\Gamma; n \vdash t, u$ as input, the return type of the unify is a record with two fields: a context Δ, which is $\bot$ in case there is no successful unifier, and a substitution $\sigma : \Gamma \to \Delta$, which is the mgu of t and u (the latter property is however not explicitly enforced by the type signature in Figure 3). We denote such a situation by $\Gamma \vdash t = u \Rightarrow \sigma \dashv \Delta$, leaving the environment n implicit to alleviate the notation: the symbol $\Rightarrow$ separates the input and the output of the unification algorithm.

This unification function recursively inspects the structure of the given terms until reaching a metavariable at the top-level, as seen in the second box of Figure 4. The last two cases handle unification of two error terms, and unification of two different *rigid* term constructors (application, λ-abstraction, or variables), resulting in failure.

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

(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 $\mathbf{\Gamma}$ without M, denoted by $\mathbf{\Gamma}\backslash M$.

In the first case, the line let $p, z =$ commonPositions $m\ x\ y$ computes 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 $\mathbf{\Gamma}$, and σ maps M to $P(z)$.

[3]The similarity with the above introduced notation is no coincidence: as we will see (Remark 3.11), both are (co)equalisers.

Example 2.4. 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 $\mathbf{\Gamma}$. Accordingly, the implementation keeps M but changes its arity to p, resulting in a context denoted by $\mathbf{\Gamma}[M : p]$.

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 intuition is 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 metacontext Γ, while x is the argument of the metavariable whose arity m is left implicit, as well as its (irrelevant) name. The output is a metacontext Δ, together with a term t' in context $\Delta; m$, and a substitution $\sigma : \Gamma \longrightarrow \Delta$. If Γ is proper, this is precisely the data for the most general unifier of t and $M(x)$, considered in the extended metacontext $M : m, \Gamma$. Following the above pruning intuition, t' is the term t where the outbound variables have been pruned, in case of success. This justifies the type signature of the prune in Figure 3. 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, the line let p, x', y' = commonValues $m\ x\ y$ computes the vectors of *common value positions* $(x'_1, \ldots, x'_p)$ and $(y'_1, \ldots, y'_p)$ between $x_1, \ldots, x_m$ and $y_1, \ldots, y_{m'}$, i.e., the pair of maximal lists $(\vec{x'}, \vec{y'})$ of distinct positions such that $x_{\vec{x'}} = y_{\vec{y'}}$. We denote[4] such a situation by $m \vdash x :> y \Rightarrow y'; x' \dashv p$. 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 $P(x')$ and M' to $P(y')$.

Example 2.5. 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. Assume that those two unification problems yield t' and u' as replacements for t and u, as well as

[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 4.3.

Fig. 3. Type signatures of the functions implemented in Figure 4 and Figure 5

```
record _⟶? Γ : Set k' where
  constructor _◄_
  field
    Δ : MetaContext
    σ : Γ ⟶ Δ

record [_]∪_⟶? m Γ : Set k' where
  constructor _◄_
  field
    Δ : MetaContext
    u,σ : (Tm Δ m) × (Γ ⟶ Δ)

record _∪_⟶? (Γ : MetaContext·)(Γ' : MetaContext)
    : Set (i ⊔ j ⊔ k) where
  constructor _◄_
  field
    Δ : MetaContext
    δ,σ : (Γ ·⟶ Δ) × (Γ' ⟶ Δ)

prune : ∀ {Γ a m} → Tm Γ a → hom m a → [ m ]∪ Γ ⟶?
prune-σ : ∀ {Γ Γ' Γ''} → (Γ' ·⟶ Γ) → (Γ'' ⟹ Γ') → Γ'' ∪ Γ ⟶?

unify-flex-* : ∀ {Γ m a} → m ∈ Γ → hom m a → Tm· Γ a → Γ ·⟶?
unify : ∀ {Γ a} → Tm Γ a → Tm Γ a → Γ ⟶?
unify-σ : ∀ {Γ Γ'} → (Γ' ⟶ Γ) → (Γ' ⟶ Γ) → (Γ ⟶?)
```

substitution σ_1 and σ_2, then M should be replaced accordingly with $t'[\sigma_2]\ u'$. Note that this really involves improper application, taking into account the following three subcases at once.

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

$$\frac{\Gamma \vdash t :> x \Rightarrow t'; \sigma_1 \dashv \Delta_1 \qquad \Delta_1 \vdash u[\sigma_1] :> x \Rightarrow !; !_s \dashv \bot}{\Gamma \vdash t\ u :> x \Rightarrow !; !_s \dashv \bot} \qquad \frac{\Gamma \vdash t :> x \Rightarrow !; !_s \dashv \bot \qquad \bot \vdash ! :> x \Rightarrow !; !_s \dashv \bot}{\Gamma \vdash t\ u :> x \Rightarrow !; !_s \dashv \bot}$$

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.

2.2 Generalisation

In this section, we show how to abstract over λ-calculus to get a generic algorithm for pattern unification, parameterised by a new notion of signature to account for syntax with metavariables. We split this notion in two parts:

(1) a notion of generalised binding signature, or GB-signature (formally introduced in Definition 3.13), specifying a syntax with metavariables, for which unification problems can be stated;
(2) some additional structures used in the algorithm to solve those unification problems, as well as properties ensuring its correctness, making the GB-signature *pattern-friendly* (see Definition 3.15).

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

```
prune {⌊ Γ ⌋} (M :  m ( x )) y =
  let p , x' , y' = commonValues m x y
  in Γ [ M :  p ] ·◄ ((M :  p) ( y' ) , M ↦-( x' ))

prune ! y = ⊥ ◄ (! , !s)

prune (App· t u) x =
  let Δ1 ◄ (t' , σ1) = prune t x
      Δ2 ◄ (u' , σ2) = prune (u [ σ1 ]t) x
  in  Δ2 ◄ (App (t' [ σ2 ]t) u' , σ1 [ σ2 ]s)

prune (Lam· t) x =
  let Δ ◄ (t' , σ) = prune t (x ↑)
  in Δ ◄ (Lam t' , σ)

prune {Γ} (Var· i) x with i { x }⁻¹
... | ⊥ = ⊥ ◄ (! , !s)
... | ⌊ PreImage j ⌋ = Γ ◄ (Var j , 1s)
```

$$\frac{m \vdash x :> y \Rightarrow y'; x' \dashv p}{\Gamma[M : m] \vdash M(x) :> y \Rightarrow P(y'); M \mapsto P(x') \dashv \Gamma[P : p]}\ \textsc{P-Flex}$$

$$\frac{}{\bot \vdash\, ! :> x \Rightarrow\, !; !_s \dashv \bot}\ \textsc{P-Fail}$$

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

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

$$\frac{i \notin x}{\Gamma \vdash \underline{i} :> x \Rightarrow\, !; !_s \dashv \bot} \qquad \frac{i = x_j}{\Gamma \vdash \underline{i} :> x \Rightarrow \underline{j}; 1_\Gamma \dashv \Gamma}$$

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

unify-flex-* {Γ} {m} M x t
  with occur-check M t
... | Same-MVar y =
  let p , z = commonPositions m x y
  in Γ [ M :  p ] ·◄ M ↦-( z )
... | Cycle = ⊥ ◄ !s
... | No-Cycle t' =
  let Δ ◄ (u , σ) = prune t' x
  in Δ ◄ M ↦ u , σ

unify (App· t u) (App· t' u') =
  let Δ1 ◄ σ1 = unify t t'
      Δ2 ◄ σ2 = unify (u [ σ1 ]t) (u' [ σ1 ]t)
  in  Δ2 ◄ σ1 [ σ2 ]s
unify (Lam· t) (Lam· t') = unify t t'

unify {Γ} (Var· i) (Var· j) with i Fin.≟ j
... | no _ = ⊥ ◄ !s
... | yes _ = Γ ◄ 1s
```

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

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

$$\frac{M \notin t \qquad \Gamma \backslash M \vdash t :> x \Rightarrow t'; \sigma \dashv \Delta}{\Gamma \vdash M(x) = t \Rightarrow M \mapsto t', \sigma \dashv \Delta}\ \textsc{No-cycle}$$

(+ symmetric rules)

$$\frac{\Gamma \vdash t = t' \Rightarrow \sigma_1 \dashv \Delta_1 \qquad \Delta_1 \vdash u[\sigma_1] = u'[\sigma_2] \Rightarrow \sigma_2 \dashv \Delta_2}{\Gamma \vdash t\ u = t'\ u' \Rightarrow \sigma_1[\sigma_2] \dashv \Delta_2}$$

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

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

```
unify ! ! = ⊥ ◄ !s
```

$$\frac{}{\bot \vdash\, ! = ! \Rightarrow\, !_s \dashv \bot}\ \textsc{U-Fail}$$

```
unify _ _ = ⊥ ◄ !s
```

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

Fig. 5. Our generic pattern unification algorithm

```
prune {⌊ Γ ⌋} (M : m ( x )) y =
  let p , x' , y' = pullback m x y in
  Γ [ M : p ] ·◄ ((M : p) ( y' ) , M ↦-( x' ))
```

Same as the rule P-Flex in Figure 4.

```
prune ! y = ⊥ ◄ (! , !s)
```

Same as the rule P-Fail in Figure 4.

```
prune (Rigid· o δ) x with o { x }⁻¹
... | ⊥ = ⊥ ◄ (! , !s)
... | ⌊ PreImage o' ⌋ =
  let Δ ◄ (δ' , σ) = prune-σ δ (x ^ o')
  in Δ ◄ (Rigid o' δ' , σ)
```

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

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

```
prune-σ {Γ} [] [] = Γ ◄ ([] , 1s)
prune-σ (t , δ) (x₀ :: xs) =
  let Δ₁ ◄ (t' , σ₁) = prune t x₀
      Δ₂ ◄ (δ' , σ₂) = prune-σ (δ [ σ₁ ]s) xs
  in  Δ₂ ◄ ( (t' [ σ₂ ]t , δ') , (σ₁ [ σ₂ ]s) )
```

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

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

unify-flex-* is defined as in Figure 4, 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 4.

```
unify (Rigid· o δ) (Rigid· o' δ') with o ≟ o'
... | no _ = ⊥ ◄ !s
... | yes ≡.refl = unify-σ δ δ'
```

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

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

```
unify ! ! = ⊥ ◄ !s
```

Same as the rule U-Fail in Figure 4.

```
unify-σ {Γ} [] [] = Γ ◄ 1s
unify-σ (t₁ , δ₁) (t₂ , δ₂) =
  let Δ ◄ σ = unify t₁ t₂
      Δ' ◄ σ' = unify-σ (δ₁ [ σ ]s) (δ₂ [ σ ]s)
  in  Δ' ◄ σ [ σ' ]s
unify-σ 1⊥ 1⊥ = ⊥ ◄ !s
```

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

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

$$\frac{}{\bot \vdash 1_\bot = 1_\bot \Rightarrow\ !_s \dashv \bot}\ \textsc{U-Id-Fail}$$

This separation is motivated by the fact that in the case of λ-calculus, the vectors of common (value) positions as well as inverse renaming $-\{-\}^{-1}$ of variables are involved in the algorithm, but not in the definition of the syntax and associated operations (renaming, metavariable substitution).

Let us first focus on the notion of GB-signature, starting from binding signatures [2]: the latter consist in a set of operation symbols, and for each $o \in O$, an arity $\alpha_o = (\overline{o}_1, \ldots, \overline{o}_n)$, i.e., a list of

Fig. 6. Generalised binding signatures in Agda

```
record Signature i j k : Set (lsuc (i ⊔ j ⊔ k)) where
  field
    A : Set i
    hom : A → A → Set j
    id : ∀ {a} → hom a a
    _∘_ : ∀ {a b c} → hom b c → hom a b → hom a c
    O : A → Set k
    α : ∀ {a} → O a → List A

    - Functoriality components
    _{_} : ∀ {a b} → O a → hom a b → O b
    _^_ : ∀ {a b}(x : hom a b)(o : O a) → α o ⟹ α (o { x } )
```

Fig. 7. Syntax generated by a GB-signature

```
MetaContext· = List A
MetaContext = Maybe MetaContext·

data Tm : MetaContext → A
            → Set (i ⊔ j ⊔ k)
Tm· Γ a = Tm ⌊ Γ ⌋ a
```

```
data Tm where
  Rigid· : ∀ {Γ a}(o : O a) → (α o ·⟶· Γ)
         → Tm· Γ a
  _(_) : ∀ {Γ a m} → m ∈ Γ → hom m a
         → Tm· Γ a
  ! : ∀ {a} → Tm ⊥ a
```

natural numbers specifying how many variables are bound in each argument. For example, pure λ-calculus is specified by $O = \{lam, app\}$, with $\alpha_{app} = (0, 0)$, $\alpha_{lam} = (1)$. Now, a GB-signature consists in a tuple $(\mathcal{A}, O, \alpha)$ consisting of

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

such that O and α are functorial in a suitable sense (see Remark 2.10 below).

Intuitively, $O(a)$ is the set of operation symbols available in the environment a. The Agda implementation in Figure 6 does not include properties such as associativity of morphism composition, although they are assumed in the proof of correctness. For example, the latter associativity property ensures that composition of metavariable substitutions is associative.

Example 2.6. 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;
- $O'(p) = \{\underline{1}, \ldots, \underline{p}\} \sqcup \{o_p | o \in O\}$;
- $\alpha'_{\underline{i}} = ()$ and $\alpha'_{o_p} = (p + \overline{o}_1, \ldots, p + \overline{o}_n)$ for any $i, p \in \mathbb{N}$ and n-ary operation symbol $o \in O$.

Note that, contrary to binding signatures, variables $\underline{i}$ are explicitly specified as nullary operations.

The syntax specified by a GB-signature $(\mathcal{A}, O, \alpha)$ is inductively defined in Figure 7, where a context $\Gamma; a$ is defined as in Section §2.1 for λ-calculus, except that variables contexts and metavariable arities are objects of $\mathcal{A}$ instead of natural numbers. We indeed recover the syntax of λ-calculus of Figure 1 by considering the GB-signature generated by the binding signature of λ-calculus (Example 2.6).

We call a term *rigid* if it is of the shape $o(\dots)$, *flexible* if it is some metavariable application $M(\dots)$.

Remark 2.7. Recall that the Agda code uses a nameless convention for metacontexts: they are just lists of environments. Therefore, the arity α_o of an operation o can be considered as a metacontext. It follows that the argument of an operation o in the context $\mathbf{\Gamma}; a$ can be specified either as a metavariable substitution (defined in Figure 2) from $\alpha_o = (\overline{o}_1, \dots, \overline{o}_n)$ to $\mathbf{\Gamma}$, as in the Agda code, or explicitly as a list of terms $(t_1, \dots, t_n)$ such that $\mathbf{\Gamma}; \overline{o}_i \vdash t_i$, as in the rule Rig. In the following, we will use either interpretation.

Remark 2.8. The syntax in the empty metacontext does not depend on the morphisms in $\mathcal{A}$. In fact, by restricting the morphisms in $\mathcal{A}$ to identity morphisms, any GB-signature induces an indexed container [5] generating the same syntax without metavariables.

Example 2.9. GB-signatures capture multi-sorted binding signatures such as simply-typed λ-calculus, or polymorphic syntax such as System F (see Section §7). Although equations are not explicitly supported, simply-typed λ-calculus modulo β- and η- equations can be handled by working on the normalised syntax (see Section §7.2).

Remark 2.10. In the notion of GB-signature, functoriality ensures that the generated syntax supports renaming: given a morphism $x : a \to b$ in $\mathcal{A}$ and a term $\Gamma; a \vdash t$, we can recursively define a term $\Gamma; b \vdash t\{x\}$. The metavariable base case is the same as in Section §2.1: $M(y)\{x\} = M(x \circ y)$. For an operation $o(t_1, \dots, t_n)$, functoriality provides the following components:

(1) a n-ary operation symbol $o\{x\} \in O(b)$;
(2) a list of morphisms $(x_1^o, \dots, x_n^o)$ in $\mathcal{A}$ such that $x_i^o : \overline{o}_i \to \overline{o\{x\}}_i$ for each $i \in \{1, \dots, n\}$.

Then, $o(t_1, \dots, t_n)\{x\}$ is defined as $o\{x\}(t_1\{x_1^o\}, \dots, t_n\{x_n^o\})$.

Notation 2.11. If $\mathbf{\Gamma}$ and $\mathbf{\Delta}$ are two metacontexts $M_1 : m_1, \dots, M_p : m_p$ and $N_1 : n_1, \dots, N_p : n_p$ of the same length, we write $\delta : \mathbf{\Gamma} \Longrightarrow \mathbf{\Delta}$ to mean that δ is a *vector of renamings* $(\delta_1, \dots, \delta_n)$ between $\mathbf{\Gamma}$ and $\mathbf{\Delta}$, in the sense that each δ_i is a morphism between m_i and n_i. The second functoriality component in Remark 2.10 is accordingly specified as a vector of renamings $x^o : \alpha_o \Longrightarrow \alpha_{o\{f\}}$ in Figure 7, considering operation arities as nameless metacontexts (Remark 2.7). We extend the renaming notation to substitutions: given $\delta : \Gamma \to \mathbf{\Delta}$ and $x : \mathbf{\Delta'} \Longrightarrow \mathbf{\Delta}$, we define $\delta\{x\} : \Gamma \to \mathbf{\Delta'}$ as $(\delta_1\{x_1\}, \dots, \delta_n\{x_n\})$ where n is the length of Δ, so that $o(\delta)\{x\}$ can be equivalently defined as $o\{x\}(\delta\{x^o\})$. Note that a vector of renamings $\delta : \mathbf{\Gamma} \Longrightarrow \mathbf{\Delta}$ canonically induces a metavariable substitution $\overline{\delta} : \mathbf{\Delta} \to \mathbf{\Gamma}$, mapping N_i to $M_i(\delta_i)$.

The Agda code adapting the definitions of Section §2.1 to a syntax generated by a generic signature is usually shorter because the application, λ-abstraction, and variable cases are replaced with a single rigid case. Because of Remark 2.7, it is more convenient to define operations on terms mutually with the corresponding operations on substitutions. For example, composition of substitutions is defined mutually with substitution of terms in the second box of Figure 2. The same applies for renaming of terms and substitution as in Notation 2.11.

We are similarly led to generalise unification of terms to unification of proper substitutions, and we extend accordingly the notation. Given two substitutions $\delta_1, \delta_2 : \mathbf{\Gamma'} \to \Gamma$, we write

$\Gamma \vdash \delta_1 = \delta_2 \Rightarrow \sigma \dashv \Delta$ to mean that $\sigma : \Gamma \longrightarrow \Delta$ 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 main unification function is thus split in two functions, unify for single terms, and unify-σ for substitutions. Similarly, we define pruning of terms mutually with pruning of proper substitutions. We thus also extend the pruning notation: given a substitution $\delta : \mathbf{\Gamma'} \longrightarrow \Gamma$ and a vector $x : \mathbf{\Gamma''} \Longrightarrow \mathbf{\Gamma'}$ of renamings, the judgement $\Gamma \vdash \delta \mathrel{:>} x \Rightarrow \delta'; \sigma \dashv \Delta$ means that the substitution $\sigma : \Gamma \longrightarrow \Delta$ extended with $\delta' : \mathbf{\Gamma''} \longrightarrow \Delta$ is the most general unifier of δ and $\overline{x}$ as substitutions from $\Gamma, \mathbf{\Gamma'}$ to Δ. The outputs of unify and unify-σ are gathered as fields of record types (see Figure 3).

In the λ-calculus implementation (Figure 4), unification of two metavariable applications requires computing the vector of common positions or value positions of their arguments, depending on whether the involved metavariables are identical. Both vectors are characterised as equalisers or pullbacks in the category $\mathbb{F}_m$ defined in Example 2.6, thus providing a canonical replacement in the generic algorithm, along with new interpretations of the notations $m \vdash x = y \Rightarrow z \dashv p$ and $m \vdash x \mathrel{:>} y \Rightarrow y'; x' \dashv p$ and as equalisers and pullbacks.

Notation 2.12. We denote an equaliser $p \xrightarrow{z} m \underset{y}{\overset{x}{\rightrightarrows}} \ldots$ in $\mathcal{A}$ by $m \vdash x = y \Rightarrow z \dashv p$. Similarly, $m \vdash x \mathrel{:>} y \Rightarrow y'; x' \dashv p$ denotes a pullback in $\mathcal{A}$ of the shape

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

Let us now comment on pruning rigid terms, when 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 $o'\{x\}(\delta'\{x^{o'}\}) = o(\delta)$, so that, in particular, $o'\{x\} = o$. In other words, 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 symbols in any variable context. Inverse renaming is a function provided by *friendly* GB-signatures, which are GB-signatures with additional components listed in Figure 8 on which the algorithm relies. To sum up,

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

The formal notion of pattern-friendly signatures (Definition 3.15) includes additional properties ensuring correctness of the algorithm.

3 Categorical semantics

To prove that the algorithm is correct, we show in the next sections that the inductive rules describing the implementation are sound. For instance, the rule U-Split is sound on the condition that the output of the conclusion is a most general unifier whenever the output of the premises are most general unifiers. We 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.

Fig. 8. Friendly GB-signatures in Agda

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

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 [6, 28]. We furthermore provide a formal justification for the error metacontext $\perp$.

Lemma 3.1. *Proper metacontexts 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 3.20 that this category fully faithfully embeds in a Kleisli category for a monad generated by S on $[\mathcal{A}, \mathrm{Set}]$.

Remark 3.2. The opposite category of $\mathrm{MCon}(S)$ is equivalent to a multi-sorted Lawvere theory whose sorts are the objects of $\mathcal{A}$. In general, this theory is not freely generated by operations unless $\mathcal{A}$ is discrete, in which case we recover (multi-sorted) first-order unification. Note that even the GB-signature induced (as in Example 2.6) by an empty binding signature is not "free" in this sense.

Lemma 3.3. *The most general unifier of two parallel substitutions* $\Gamma' \overset{\delta_1}{\underset{\delta_2}{\rightrightarrows}} \Gamma$ *is characterised as their coequaliser.*

This motivates a new interpretation of the unification notation, that we introduce later in Notation 3.10, after explaining how failure is categorically handled. Indeed, pattern unification is typically stated as the existence of a coequaliser on the condition that there is a unifier in this category $\mathrm{MCon}(S)$. But we can get rid of this condition by considering the category $\mathrm{MCon}(S)$ freely extended with a terminal object $\perp$, resulting in the full category of metacontexts and substitutions.

Definition 3.4. Given a category $\mathscr{B}$, let $\mathscr{B}_\perp$ denote the category $\mathscr{B}$ extended freely with a terminal object $\perp$.

Notation 3.5. We denote by $!_s$ any terminal morphism to $\perp$ in $\mathscr{B}_\perp$.

Lemma 3.6. *Metacontexts and substitutions between them define a category which is isomorphic to* $\mathrm{MCon}(S)_\perp$.

In Section §2.1, we already made sense of this extension. Let us rephrase our explanations from a categorical perspective. 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 results are also useful.

Lemma 3.8. *Let $\mathscr{B}$ be a category.*

(i) The canonical embedding functor $\mathscr{B} \to \mathscr{B}_\perp$ creates colimits.
(ii) Any diagram J in $\mathscr{B}_\perp$ such that $\perp$ is in its image has a colimit given by the terminal cocone on $\perp$.

This ensures in particular that coproducts in $\mathrm{MCon}(S)$, which are computed as union of metacontexts, are also coproducts in $\mathrm{MCon}(S)_\perp$. It also justifies defining the union of a proper metacontext with $\perp$ as $\perp$.

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

This justifies the following interpretation to the unification notation.

Notation 3.10. $\Gamma \vdash \delta_1 = \delta_2 \Rightarrow \sigma \dashv \Delta$ denotes a coequaliser $\ldots \overset{\delta_1}{\underset{\delta_2}{\rightrightarrows}} \Gamma \xrightarrow{\sigma} \Delta$ in $\mathrm{MCon}(S)_\perp$.

Remark 3.11. This is the same interpretation as in Notation 2.12 for equaliser, taking $\mathcal{A}$ to be the opposite category of $\mathrm{MCon}(S)_\perp$.

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 3.15).

Theorem 3.12. *Given any pattern-friendly signature S, the category $\mathrm{MCon}(S)_\perp$ has coequalisers.*

3.2 Initial Algebra Semantics for GB-signatures

The proofs of various statements presented in this section are detailed in the appendices found in the supplemental material.

Definition 3.13. 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 $O_n(a)\times\{1, \ldots., n\}$, defined as follows:

- objects are tuples (n, a, o, i) such that $o \in O_n(a)$ and $i \in \{1, \ldots, n\}$;
- 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 $O_n(f) : O_n(a) \to O_n(a')$.

Remark 3.14. This definition of GB-signatures superficially differs from the one we informally introduced in Section §2.2, in the sense that the set of operation symbols $O(a)$ in an environment a

was not indexed by natural numbers. The two descriptions are equivalent: $O_n(a)$ is recovered as the subset of n-ary operation symbols in $O(a)$, and conversely, $O(a)$ is recovered as the union of all the $O_n(a)$ for every natural number n.

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

Definition 3.15. A GB-signature $S = (\mathcal{A}, O, \alpha)$ is said to be *pattern-friendly* if

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

These conditions ensure the following two properties.

Property 3.16 (proved in §A.1). The following properties hold for pattern-friendly signatures.

(i) The action of $O_n : \mathcal{A} \longrightarrow \mathrm{Set}$ 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 $\mathcal{A}^{op} \longrightarrow \mathrm{MCon}(S)_\perp$ mapping a morphism $x \in \hom_{\mathcal{A}}(b, a)$ to the substitution $(X : a) \to (X : b)$ 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 $\mathrm{MCon}(S)_\perp$.

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

Remark 3.17. A metavariable application $\Gamma; a \vdash M(x)$ corresponds to the composition $\mathcal{L}x[in_M]$ as a substitution from $X : a$ to Γ, where in_M is the coproduct injection $(X : m) \cong (M : m) \hookrightarrow \Gamma$ mapping M to $M(1_m)$.

The rest of this section, we provide Initial Algebra Semantics for the generated syntax (this is used in the proof of Property 3.16.(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

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

Lemma 3.18 (proved in §A.2). *F is finitary and generates a free monad T. Moreover, TX is the initial algebra of $Z \mapsto X + FZ$.*

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

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

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

Notation 3.19. Given a proper metacontext Γ. We sometimes denote $\underline{\Gamma}$ just by Γ.

If $\Gamma = (M_1 : m_1, ..., M_p : m_p)$ and Δ are metacontexts, a Kleisli morphism $\sigma : \Gamma \to T\Delta$ is equivalently given (by combining the above lemma, 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 metacontexts $\mathrm{MCon}(S)$ and the Kleisli category of T.

Lemma 3.20. *The category* $\mathrm{MCon}(S)$ *is equivalent to the full subcategory of* Kl_T *spanned by coproducts of representable functors.*

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

Lemma 3.21 (proved in §A.3). *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 3.15 holds.*

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

Lemma 3.22 (proved in §A.4). $\mathscr{C}$ *is closed under limits, coproducts, and filtered colimits. Moreover, it is cocomplete.*

Corollary 3.23 (proved in §A.5). T *restricts as a monad on* $\mathscr{C}$ *freely generated by the restriction of* F *as an endofunctor on* $\mathscr{C}$ *(Lemma 3.21).*

4 Soundness of the pruning phase

In this section, we assume a pattern-friendly GB-signature S and discuss soundness of the main rules of the two mutually recursive functions prune and prune-σ listed in Figure 5, which handles unification of two substitutions $\delta : \mathbf{\Gamma}'_1 \to \Gamma$ and $\overline{x} : \mathbf{\Gamma}'_1 \to \mathbf{\Gamma}'_2$ where $\overline{x}$ is induced by a vector of renamings $x : \mathbf{\Gamma}'_2 \Longrightarrow \mathbf{\Gamma}'_1$. Strictly speaking, this is not unification as we introduced it because δ and $\overline{x}$ do not target the same context, but it is straightforward to adapt the definition: a unifier is given by two substitutions $\sigma : \Gamma \to \Delta$ and $\sigma' : \mathbf{\Gamma}'_2 \to \Delta$ such that the following equation holds

$$\delta[\sigma] = \overline{x}[\sigma'] \tag{1}$$

As usual, the mgu is defined as the unifier uniquely factoring any other unifier.

Remark 4.1. The right hand-side $\overline{x}[\sigma']$ in (1) is actually equal to $\sigma'\{x\}$. Indeed, $\overline{x} = (\dots, M_i(x_i), \dots)$ and $M_i(x_i)[\sigma'] = \sigma'_i\{x_i\}$.

From a categorical point of view, such a mgu is characterised as a pushout.

Notation 4.2. Given $\delta : \mathbf{\Gamma}'_1 \to \Gamma$, $x : \mathbf{\Gamma}'_2 \Longrightarrow \mathbf{\Gamma}'_1$, $\sigma : \Gamma \to \Delta$, and $\sigma' : \mathbf{\Gamma}'_2 \to \Delta$, the notation $\Gamma \vdash \delta :> x \Rightarrow \sigma'; \sigma \dashv \Delta$ means that the square
$$\begin{tikzcd} \mathbf{\Gamma}'_1 \arrow[r, "\overline{x}"] \arrow[d, "\delta"'] & \mathbf{\Gamma}'_2 \arrow[d, "\sigma'"] \\ \Gamma \arrow[r, "\sigma"'] & \Delta \end{tikzcd}$$
is a pushout in $\mathrm{MCon}(S)_\perp$.

Remark 4.3. This justifies the similarity between the pruning notation $- \vdash - :> - \Rightarrow -; -$ and the pullback notation of Notation 2.12, since pushouts in a category are nothing but pullbacks in the opposite category.

In the following subsections, we detail soundness of the rules for the rigid case (Section §4.1) and then for the flex case (Section §4.2).

The rules P-Empty and P-Split are straightforward adaptions specialised to those specific unification problems of the rules U-Empty and U-Split described later in Section §5.1. The failing rule P-Fail is justified by Lemma 3.8.(ii).

4.1 Rigid (rules P-RIG and P-RIG-FAIL)

The rules P-RIG and P-RIG-FAIL handle non-cyclic unification of $M(x)$ with $\Gamma; a \vdash o(\delta)$ for some $o \in \mathcal{O}_n(a)$, where $M \notin \Gamma$. By Remark 4.1, a unifier is given by a substitution $\sigma : \Gamma \longrightarrow \Delta$ and a term u such that

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

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, $u = o'(\delta')$ for some $o' \in \mathcal{O}_n(m)$ and δ' is a substitution from $\alpha_{o'}$ to Δ. Then, $u\{x\} = o'\{x\}(\delta\{x^{o'}\})$. It follows from Equation (2) that $o = o'\{x\}$, and $\delta[\sigma] = \delta'\{x^{o'}\}$.

Note that there is at most one o' such that $o = o'\{x\}$, by Property 3.16.(i). In this case, a unifier is equivalently given by substitutions $\sigma : \Gamma \longrightarrow \Delta$ and $\sigma' : \alpha_{o'} \longrightarrow \Delta$ such that $\delta[\sigma] = \sigma'\{x^{o'}\}$. But, by Remark 4.1, this is precisely the data for a unifier of δ and $x^{o'}$. This actually induces an isomorphism between the two categories of unifiers, thus justifying the rules P-RIG and P-RIG-FAIL.

4.2 Flex (rule P-FLEX)

The rule P-FLEX handles unification of $M(x)$ with $N(y)$ where $M \neq N$ in an environment a. More explicitly, this is about computing the pushout of $(X : a) \xrightarrow{\mathcal{L}x} (X : m) \cong (M : m) \xhookrightarrow{in_M} \Gamma$ and $(X : a) \xrightarrow{\mathcal{L}x} (X : n) \cong (N : n)$.

Thanks to the following lemma, it is actually enough to compute the pushout of $\mathcal{L}x$ and $\mathcal{L}y$, taking $A = (X : a)$, $B = (X : m)$, $C = (X : N)$, $Y = \Gamma \backslash M$, so that $B + Y \cong \Gamma$.

LEMMA 4.4. *In any category, if the square below left is a pushout, then so is the square below right.*

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

By Property 3.16.(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.

5 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 two mutually recursive functions unify and unify-σ listed in Figure 5, which compute coequalisers in $\mathrm{MCon}(S)_\perp$.

The failing rules U-FAIL and U-ID-FAIL are justified by Lemma 3.8.(ii). Both rules CLASH and U-RIG handle unification of two rigid terms $o(\delta)$ and $o'(\delta')$. If $o \neq o'$, they do not have any unifier: this is the rule CLASH. If $o = o'$, then a substitution is a unifier if and only if it unifies δ and δ', thus justifying the U-RIG.

In the next subsections, we discuss the rule sequential rules U-EMPTY and U-SPLIT (Section §5.1), the rule NO-CYCLE transitioning to the pruning phase (Section §5.2), the rule SAME-MVAR unifying metavariable with itself (Section §5.3), and the failing rule CYCLE for cyclic unification of a metavariable with a term which includes it deeply (Section §5.4).

5.1 Sequential unification (rules U-EMPTY and U-SPLIT)

The rule U-EMPTY is a direct application of the following general lemma.

LEMMA 5.1. *If A is initial in a category, then any diagram of the shape $A \rightrightarrows B \xrightarrow{1_B} B$ is a coequaliser.*

The rule U-Split is a direct application of a stepwise construction of coequalisers valid in any category, as noted by [28, Theorem 9]: if the first two diagrams below are coequalisers, then the last one as well.

$$
\begin{tikzcd}
\Gamma'_1 \arrow[r, shift left, "t_1"] \arrow[r, shift right, "u_1"'] & \Gamma \arrow[r, dashed, "\sigma_1"] & \Delta_1
\end{tikzcd}
\qquad
\begin{tikzcd}
 & \Gamma \arrow[dr, "\sigma_1"] & & \\
\Gamma'_2 \arrow[ur, "t_2"] \arrow[dr, "u_2"'] & & \Delta_1 \arrow[r, dashed, "\sigma_2"] & \Delta_2 \\
 & \Gamma \arrow[ur, "\sigma_1"'] & &
\end{tikzcd}
$$

$$
\begin{tikzcd}
\Gamma'_1 + \Gamma'_2 \arrow[r, shift left, "t_1, t_2"] \arrow[r, shift right, "u_1, u_2"'] & \Gamma \arrow[r, dashed, "\sigma_2 \circ \sigma_1"] & \Delta_2
\end{tikzcd}
$$

5.2 Flex-Flex, no cycle (rule No-Cycle)

The rule No-Cycle transitions from unification to pruning. While unification is a coequaliser construction, in Section §4, we explained that pruning is a pushout construction. The rule is justified by the following well-known connection between those two notions, taking B to be $\mathbf{\Gamma} \backslash M$ and C to be the singleton context $M : m$, so that the coproduct of those two contexts in $\mathrm{MCon}(S)_\perp$ is their disjoint union $\mathbf{\Gamma}$.

Lemma 5.2. *Consider a commuting square*
$$
\begin{tikzcd}
A \arrow[r, "u"] \arrow[d, "v"'] & B \arrow[d, "f"] \\
C \arrow[r, "g"'] & D
\end{tikzcd}
$$
in any category. If the coproduct $B + C$ of B and C exists, then this is a pushout if and only if $B + C \xrightarrow{f,g} D$ is the coequaliser of $in_1 \circ u$ and $in_2 \circ v$.

5.3 Flex-Flex, same metavariable (rule Same-MVar)

Here we detail unification of $M(x)$ and $M(y)$, for $x, y \in \hom_{\mathcal{A}}(m, a)$. By Remark 3.17, $M(x) = \mathcal{L}x[in_M]$ and $M(y) = \mathcal{L}y[in_M]$. We exploit the following lemma with $u = \mathcal{L}x$ and $v = \mathcal{L}y$.

Lemma 5.3. *In any category, 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 3.16.(ii), it is the image by $\mathcal{L}$ of the equaliser of x and y, thus justifying the rule Same-MVar.

5.4 Flex-rigid, cyclic (rule Cycle)

The rule Cycle handles unification of $M(x)$ and a term t such that t is rigid and M occurs in t. In this section, we show that indeed there is no successful unifier. More precisely, we prove Corollary 5.8 below, stating that if there is a unifier of a term t and a metavariable application $M(x)$, then either M occurs at top-level in t, or it does not occur at all. The argument follows the basic intuition that $\sigma_M = t[M \mapsto \sigma_M]$ 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 5.4. The size $|t| \in \mathbb{N}$ of a proper 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 5.5. 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 5.6. *For any term $\Gamma; a \vdash t$, if $|t|_M = 0$, then $\Gamma \backslash M; a \vdash t$. Moreover, for any $\Gamma = (M_1 : m_1, \ldots, M_n : m_n)$, well-formed term t in context $\Gamma; a$, and successful substitution $\sigma : \Gamma \longrightarrow \Delta$, we have $|t[\sigma]| = |t| + \sum_i |t|_{M_i} \times |\sigma_i|$.*

Corollary 5.7. *For any term t in context $\Gamma; a$ with $(M : m) \in \Gamma$, successful substitution $\sigma : \Gamma \longrightarrow \Delta$, 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 5.8. *Let t be a term in context $\Gamma; a$ with $(M : m) \in \Gamma$ and $x \in \hom_{\mathcal{A}}(m, a)$ such that $(M \mapsto u, \sigma) : \Gamma \longrightarrow \Delta$ 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 $t[\sigma, M \mapsto u] = M(x)[u]$, we have $|t[\sigma, M \mapsto u]| = |M(x)[u]|$. Corollary 5.7 implies $|u| \geq |t| + |u| \times |t|_M$. Therefore, either $|t|_M = 0$ and we conclude by Lemma 5.6, or $|t|_M > 0$ and $|t| = 0$, so that t is $M(y)$ for some y. □

6 Termination and completeness

6.1 Termination

In this section, we sketch an explicit argument to justify termination of our algorithm described in Figure 5. 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 proper metacontext Γ as its length, while $|\bot| = 0$ by definition. We also recursively define the size[5] $||t||$ of a proper 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 proper term $\Gamma; a \vdash t$ and successful 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 $\Gamma \vdash \vec{t} :> x \Rightarrow \vec{w}; \sigma \dashv \Delta$ 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 metacontext 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 Δ is proper, then σ is a metavariable renaming.*

[5] The difference with the notion of size introduced in Definition 5.4 is that metavariable applications are now of size 1 instead of 0.

6.2 Completeness

In this section, we explain why soundness (Section §4 and Section §5) and termination (Section §6.1) entail completeness. Intuitively, one may worry that the algorithm fails in cases where it should not. In fact, we already checked in the previous sections that failure only occurs when there is no unifier, as expected. Indeed, failure is treated as a free "terminal" unifier, as explained in Section §3.1, by considering the category $\mathrm{MCon}(S)_\perp$ extending category $\mathrm{MCon}(S)$ with an error metacontext $\perp$. Corollary 3.9 implies that since the algorithm terminates and computes the coequaliser in $\mathrm{MCon}(S)_\perp$, 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 $\perp$).

7 Applications

In this section, we present various examples of pattern-friendly signatures summarised in Table 1.

In Section §7.1, we present simply-typed λ-calculus, as an example of syntax specified by a multi-sorted binding signature. We then explain in Section §7.2 how we can handle β and η equations by working on the normalised syntax. Next, we introduce an example of unification for ordered syntax in Section §7.3, and finally we present an example of polymorphic such as System F, in Section §7.4.

7.1 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 [11].

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 environments, and renamings between them. An arity $\vec{\sigma} \to \tau$ consists of a list of input types $\vec{\sigma}$ and an output type τ. A term t in $\vec{\sigma} \to \tau$ considered as an environment 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 $M : (\vec{\sigma} \to \tau)$ in environment $\vec{\sigma}' \to \tau'$ 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 $M : (\vec{\sigma} \to \tau)$ to the term $P(\vec{z})$, where the arity $\vec{\sigma}' \to \tau'$ of the fresh metavariable P is the only possible choice such that $P(\vec{z})$ is a valid term in the environment $\vec{\sigma} \to \tau$, that is, $\tau' = \tau$ and $\vec{\sigma}' = \sigma_{\vec{z}}$.

7.2 Simply-typed λ-calculus modulo $\beta\eta$

Higher-order pattern unification was originally introduced for closed simply-typed lambda-terms with metavariables applied to distinct variables. Lambda-terms are considered in β-short η-long normal forms. Although we do not explicitly cover equations, the syntax of those normal formas is equation free and can be specified by a GB-signature: we take the same category of arities as in Section §7.1, and we consider the operations as specified in Table 1.

Table 1. Examples of (pattern-friendly) GB-signatures (Definition 3.13)

Simply-typed λ-calculus (Section §7.1)

Typing rule	$O(\vec{\sigma} \to \tau) = \ldots +$	$\alpha_o = (\ldots)$
$\dfrac{x : \tau \in \Gamma}{\Gamma \vdash x : \tau}$	$\{v_i \mid i \in \lvert\vec{\sigma}\rvert_\tau\}$	$()$
$\dfrac{\Gamma \vdash t : \tau' \Rightarrow \tau \quad \Gamma \vdash u : \tau'}{\Gamma \vdash t\ u : \tau}$	$\{a_{\tau'} \mid \tau' \in T\}$	$\begin{pmatrix} \vec{\sigma} \to (\tau' \Rightarrow \tau) \\ \vec{\sigma} \to \tau' \end{pmatrix}$
$\dfrac{\Gamma, x : \tau_1 \vdash t : \tau_2}{\Gamma \vdash \lambda x.t : \tau_1 \Rightarrow \tau_2}$	$\{l_{\tau_1,\tau_2} \mid \tau = (\tau_1 \Rightarrow \tau_2)\}$	$(\vec{\sigma}, \tau_1 \to \tau_2)$

Simply-typed λ-calculus modulo $\beta\eta$ (Section §7.2)

Typing rule	$O(\vec{\sigma} \to \tau) = \ldots +$	$\alpha_o = (\ldots)$
$\dfrac{x : \vec{\tau}' \Rightarrow \tau \in \Gamma \quad \tau \text{ is a base type} \quad \Gamma \vdash \vec{t} : \vec{\tau}'}{\Gamma \vdash x\vec{t} : \tau}$	$\{a_{i,\tau'_1,\ldots,\tau'_n} \mid i \in \lvert\vec{\sigma}\rvert_{\vec{\tau}' \Rightarrow \tau}$ and τ is a base type$\}$	$\begin{pmatrix} \vec{\sigma} \to \tau'_1 \\ \ldots \\ \vec{\sigma} \to \tau'_n \end{pmatrix}$
$\dfrac{\Gamma, x : \tau_1 \vdash t : \tau_2}{\Gamma \vdash \lambda x.t : \tau_1 \Rightarrow \tau_2}$	$\{l_{\tau_1,\tau_2} \mid \tau = (\tau_1 \Rightarrow \tau_2)\}$	$(\vec{\sigma}, \tau_1 \to \tau_2)$

Ordered λ-calculus (Section §7.3)

Typing rule	$O(\vec{\sigma} \mid \vec{\omega} \to \tau) = \ldots +$	$\alpha_o = (\ldots)$
$\dfrac{x : \tau \in \Gamma}{\Gamma \mid \cdot \vdash x : \tau}$	$\{v_i \mid i \in \lvert\vec{\sigma}\rvert_\tau$ and $\vec{\omega} = ()\}$	$()$
$\dfrac{}{\Gamma \mid x : \tau \vdash x : \tau}$	$\{v^{>} \mid \vec{\omega} = ()\}$	$()$
$\dfrac{\Gamma \mid \Omega \vdash t : \tau' \Rightarrow \tau \quad \Gamma \mid \cdot \vdash u : \tau'}{\Gamma \mid \Omega \vdash t\ u : \tau}$	$\{a_{\tau'} \mid \tau' \in T\}$	$\begin{pmatrix} \vec{\sigma} \mid \vec{\omega} \to (\tau' \Rightarrow \tau) \\ \vec{\sigma} \mid () \to \tau' \end{pmatrix}$
$\dfrac{\Gamma \mid \Omega_1 \vdash t : \tau' \twoheadrightarrow \tau \quad \Gamma \mid \Omega_2 \vdash u : \tau'}{\Gamma \mid \Omega_1, \Omega_2 \vdash t^{>}\ u : \tau}$	$\{a_{\tau'}^{\vec{\omega}_1,\vec{\omega}_2} \mid \tau' \in T$ and $\vec{\omega} = \vec{\omega}_1, \vec{\omega}_2\}$	$\begin{pmatrix} \vec{\sigma} \mid \vec{\omega}_1 \to (\tau' \Rightarrow \tau) \\ \vec{\sigma} \mid \vec{\omega}_2 \to \tau' \end{pmatrix}$
$\dfrac{\Gamma, x : \tau_1 \mid \Omega \vdash t : \tau_2}{\Gamma \mid \Omega \vdash \lambda x.t : \tau_1 \Rightarrow \tau_2}$	$\{l_{\tau_1,\tau_2} \mid \tau = (\tau_1 \Rightarrow \tau_2)\}$	$(\vec{\sigma}, \tau_1 \mid \vec{\omega} \to \tau_2)$
$\dfrac{\Gamma \mid \Omega, x : \tau_1 \vdash t : \tau_2}{\Gamma \mid \Omega \vdash \lambda^{>} x.t : \tau_1 \twoheadrightarrow \tau_2}$	$\{l^{>}_{\tau_1,\tau_2} \mid \tau = (\tau_1 \twoheadrightarrow \tau_2)\}$	$(\vec{\sigma}, \tau_1 \mid \vec{\omega} \to \tau_2)$

7.3 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 environment m, and there is no freedom in choosing the arguments of a metavariable application, since all the

System F (Section §7.4)

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

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 environments, and the syntax is generated by operations $L_n \times L_m \to L_{n+m}$ and abstractions $L_{n+1} \to L_n$.

Our generalisation can handle calculi combining ordered and unrestricted variables, such as the calculus underlying ordered linear logic described in Polakow and Pfenning [26]. In this section we detail this specific example. Note that this does not fit into Schack-Nielsen and Schürman's pattern unification algorithm Schack-Nielsen and Schürmann [29] for linear types where exchange is allowed (the order of their variables does not matter).

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 $\vec{\sigma}|\vec{\omega} \to \tau$, 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 $\vec{\sigma}|\vec{\omega} \to \tau$, 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 environment $\vec{\sigma}'|\vec{\omega}' \to \tau'$ 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).

Pullbacks and equalisers are computed essentially as in Section §7.1. 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 $\sigma_{\vec{z}}|\vec{\omega} \to \tau$.

7.4 Intrinsic polymorphic syntax

We present intrinsic System F, in the spirit of Hamana [16]. The Agda implementation of the friendly GB-signature can be found in the supplemental material.

The syntax of types in type environment 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 $S : \mathbb{F}_m \to \mathrm{Set}$ 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 an environment 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 environment $n'|\vec{\sigma}' \to \tau'$ must be supplied with

- a choice $(\eta_1, \ldots, \eta_n)$ of n distinct type variables among the set $\{1, \ldots n'\}$, 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 that $\mathcal{A}$ has finite connected limits. Pullbacks and equalisers in $\mathcal{A}$ are essentially computed as in Section §7.1, 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 environment $m|\vec{\sigma} \to \tau$, that is, m' is the size of $\vec{p}$, while $\tau' = \tau[p_i \mapsto i]$ and $\vec{\sigma}' = \sigma_{\vec{q}}[p_i \mapsto i]$.

8 Related work

First-order unification has been explained from a lattice-theoretic point of view by Plotkin [25], and later categorically analysed by Barr and Wells [6], Goguen [13], Rydeheard and Burstall [28, Section 9.7] as coequalisers. However, there is little work on understanding pattern unification algebraically, with the notable exception of Vezzosi and Abel [31], 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 [10] to binding signatures than to the nominal approach [12] in that everything is explicitly scoped: terms come with their support, metavariables always appear with their scope of allowed variables.

Nominal unification [30] 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.

Nominal unification and pattern unification problems are inter-translatable [8, 19]. As Cheney 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.

Pattern unification has also been studied from the viewpoint of logical frameworks [1, 22–24] using contextual types to characterise metavariables. LF-style signatures handle type dependency, but there are also GB-signatures which cannot be encoded with an LF signature. For example, GB-signatures allow us to express pattern unification for ordered lambda terms (Section §7.3).

Our semantics for metavariables has been engineered so that it can *only* interpret metavariable instantiations in the pattern fragment, and cannot interpret full metavariable instantiations, contrary to prior semantics of metavariables (e.g., Hu et al. [17] or Hamana [15]). This restriction gives our model much stronger properties, enabling us to characterise each part of the pattern unification algorithm in terms of universal properties. This lets us extend Rydeheard and Burstall's proof to the pattern case.

9 Conclusion

We presented a generic unification algorithm for Miller's pattern fragment with its associated initial-algebra semantics, parameterised by a new notion of signature for syntax with metavariables. Our setting does not handle type dependency, which notably implies that we do not deal with type metavariables (even in our system F example). An interesting question is to which extent this work could be adapted to provide semantics of higher-order pattern unification for dependently-typed languages.

References

[1] Andreas Abel and Brigitte Pientka. 2011. Higher-Order Dynamic Pattern Unification for Dependent Types and Records. In *Typed Lambda Calculi and Applications - 10th International Conference, TLCA 2011, Novi Sad, Serbia, June 1-3, 2011. Proceedings (Lecture Notes in Computer Science, Vol. 6690)*, C.-H. Luke Ong (Ed.). Springer, 10–26. https://doi.org/10.1007/978-3-642-21691-6_5

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

[3] 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.

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

[5] 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

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

[7] 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

[8] 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.

[9] 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.

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

[11] 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)*.

[12] 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.

[13] 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.

[14] 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.

[15] Makoto Hamana. 2004. Free Σ-Monoids: A Higher-Order Syntax with Metavariables. In *Proc. 2nd Asian Symposium on Programming Languages and Systems (LNCS, Vol. 3302)*, Wei-Ngan Chin (Ed.). Springer, 348–363. https://doi.org/10.1007/978-3-540-30477-7_23

[16] Makoto Hamana. 2011. Polymorphic Abstract Syntax via Grothendieck Construction.

[17] Jason Z. S. Hu, Brigitte Pientka, and Ulrich Schöpp. 2022. A Category Theoretic View of Contextual Types: From Simple Types to Dependent Types. *ACM Trans. Comput. Log.* 23, 4 (2022), 25:1–25:36. https://doi.org/10.1145/3545115

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

[19] Jordi Levy and Mateu Villaret. 2012. Nominal Unification from a Higher-Order Perspective. *ACM Trans. Comput. Log.* 13, 2 (2012), 10:1–10:31. https://doi.org/10.1145/2159531.2159532

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

[21] 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

[22] Aleksandar Nanevski, Frank Pfenning, and Brigitte Pientka. 2008. Contextual modal type theory. *ACM Trans. Comput. Log.* 9, 3 (2008), 23:1–23:49. https://doi.org/10.1145/1352582.1352591

[23] Aleksandar Nanevski, Brigitte Pientka, and Frank Pfenning. 2003. A modal foundation for meta-variables. In *Eighth ACM SIGPLAN International Conference on Functional Programming, Workshop on Mechanized reasoning about languages with variable binding, MERLIN 2003, Uppsala, Sweden, August 2003*. ACM. https://doi.org/10.1145/976571.976582

[24] Brigitte Pientka. 2003. *Tabled higher-order logic programming*. Carnegie Mellon University.

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

[26] 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.

[27] Jan Reiterman. 1977. A left adjoint construction related to free triples. *Journal of Pure and Applied Algebra* 10, 1 (1977), 57–71. https://doi.org/10.1016/0022-4049(77)90028-7

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

[29] Anders Schack-Nielsen and Carsten Schürmann. 2010. Pattern Unification for the Lambda Calculus with Linear and Affine Types. In *Proceedings 5th International Workshop on Logical Frameworks and Meta-languages: Theory and Practice, LFMTP 2010, Edinburgh, UK, 14th July 2010 (EPTCS, Vol. 34)*, Karl Crary and Marino Miculan (Eds.). 101–116. https://doi.org/10.4204/EPTCS.34.9

[30] 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.

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

A Proofs of statements in Section 3.2

A.1 Property 3.16

We use the notations and definitions of Section §3.2.

Let us first prove the first item.

Proof of Property 3.16.(i). We show that 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 $o = o'\{f\}$.

Since O_n preserves finite connected limits, it preserves monomorphisms because a morphism $f : a \to b$ is monomorphic if and only if the following square is a pullback (see [20, Exercise III.4.4]).

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

□

The rest of this section is devoted to the proof of Property 3.16.(ii).

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

Lemma A.1. *Let $\mathscr{D}$ denote the opposite category of $\mathcal{A}$ and $K : \mathscr{D} \to \mathscr{C}$ the factorisation of $\mathscr{C} \to [\mathcal{A}, \mathrm{Set}]$ by the Yoneda embedding. Then, $K : \mathscr{D} \to \mathscr{C}$ 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 $y : \mathcal{A}^{op} \to [\mathcal{A}, \mathrm{Set}]$ denote the Yoneda embedding and $J : \mathscr{C} \to [\mathcal{A}, \mathrm{Set}]$ denote the canonical embedding, so that

$$y = J \circ K. \tag{3}$$

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

$$\begin{aligned}
\mathscr{C}(K \lim F, X) &\cong [\mathcal{A}, \mathrm{Set}](JK \lim F, JX) && (J \text{ is fully faithful}) \\
&\cong [\mathcal{A}, \mathrm{Set}](y \lim F, JX) && (\text{By Equation (3)}) \\
&\cong JX(\lim F) && (\text{By the Yoneda Lemma.}) \\
&\cong \lim(JX \circ F) && (X \text{ preserves finite connected limits}) \\
&\cong \lim([\mathcal{A}, \mathrm{Set}](yF-, JX)] && (\text{By the Yoneda Lemma}) \\
&\cong \lim([\mathcal{A}, \mathrm{Set}](JKF-, JX)] && (\text{By Equation (3)}) \\
&\cong \lim \mathscr{C}(KF-, X) && (J \text{ is full and faithful}) \\
&\cong \mathscr{C}(\mathrm{colim}\, KF, X) && (\text{By left continuity of the hom-set bifunctor})
\end{aligned}$$

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

Proof of Property 3.16.(ii). Note that $\mathcal{L}$ factors as

$$\mathscr{D} \xrightarrow{\mathcal{L}^\bullet} \mathrm{MCon}(S) \hookrightarrow \mathrm{MCon}(S)_\perp,$$

where the right embedding preserves colimits by Lemma 3.8.(i), so it is enough to show that $\mathcal{L}^\bullet$ preserves finite connected colimits. Let $T_{|\mathscr{C}}$ be the monad T restricted to $\mathscr{C}$, following Corollary 3.23. Since $K : \mathscr{D} \to \mathscr{C}$ preserves finite connected colimits (Lemma A.1), 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

$\mathscr{D} \xrightarrow{\mathcal{L}^\bullet} \mathrm{MCon}(S) \hookrightarrow Kl_{T_{|\mathscr{C}}}$, where the right functor is full and faithful, $\mathcal{L}^\bullet$ also preserves finite connected colimits. □

A.2 Lemma 3.18

F is finitary because filtered colimits commute with finite limits [20, Theorem IX.2.1] and colimits. The free monad construction is due to Reiterman [27].

A.3 Lemma 3.21

Notation A.2. Given a functor $F : I \to \mathscr{B}$, 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 $\lim F \to Fi$ by p_i for any object i of I.

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

Lemma A.3. *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 $\alpha : \int J \to \mathcal{A}$ preserves finite limits.*

We first introduce a bunch of intermediate lemmas.

Lemma A.4. *If $\mathscr{B}$ is a small category with finite connected limits, then a functor $G : \mathscr{B} \to \mathrm{Set}$ 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. [3, Theorem 2.4 and Example 2.3.(iii)]. □

Corollary A.5. *Assume $\mathcal{A}$ has finite connected limits. Then $J : \mathbb{N} \times \mathcal{A} \to \mathrm{Set}$ preserves finite connected limits if and only if each $O_n : \mathcal{A} \to \mathrm{Set}$ does.*

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

Lemma A.6. *Let $F : \mathscr{B} \to \mathrm{Set}$ be a functor. For any functor $G : I \to \int F$, denoting by H the composite functor $I \xrightarrow{G} \int F \to \mathscr{B}$, 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 $y : \mathscr{B}^{op} \to [\mathscr{B}, \mathrm{Set}]$ is the Yoneda embedding. The statement follows from the universal property of a comma category. □

Lemma A.7. *Let $F : \mathscr{B} \to \mathrm{Set}$ and $G : I \to \int F$ such that F preserves the limit of $H : I \xrightarrow{G} \int F \to \mathscr{B}$. 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.6 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 $K : \mathscr{B}^{op} \to \mathscr{C}$ is the Yoneda embedding, which preserves colim G, by an argument similar to the proof of Lemma A.1. 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.8. *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 $F : \mathscr{B} \to \mathrm{Set}$ be a functor preserving those colimits. Then, $\int F$ has I-limits, preserved by the projection $\int F \to \mathscr{B}$. Moreover, a functor $G : \int F \to \mathscr{B}'$ preserves them if and only if for any $d : I \to \mathscr{B}$ and $x \in F \lim d$, the canonical morphism $G(\lim d, x) \to \int_{i:I} G(d_i, Fp_i(x))$ is an isomorphism.*

Proof. By Lemma A.7, a diagram $d' : I \to \int F$ is equivalently given by $d : I \to \mathscr{B}$ 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.9. *Assuming that $\mathcal{A}$ has finite connected limits and each O_n preserves finite connected limits, the finite limit preservation on $\alpha : \int J \to \mathcal{A}$ of Lemma A.3 can be reformulated as follows: given a finite connected diagram $d : D \to \mathcal{A}$ and element $o \in O_n(\lim d)$, the following canonical morphism is an isomorphism*

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

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

Proof. This is a direct application of Corollary A.8 and Corollary A.5. □

Lemma A.10 (Limits commute with dependent pairs). *Given functors $K : I \to \mathrm{Set}$ and $G : \int K \to \mathrm{Set}$, the following canonical morphism is an isomorphism*

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

Proof. The domain consists of a family $(\alpha_i)_{i \in I}$ where $\alpha_i \in K_i$ together with a family $(g_i)_{i \in I}$ where $g_i \in G(i, \alpha_i)$, such that that for each morphism $i \xrightarrow{u} j$ in I, we have $Ku(\alpha_i) = \alpha_j$ and $(Gu)(g_i) = g_j$.

The codomain consists of a family $(x_i, g_i)_{i \in I}$ where $x_i \in Ki$ and $g_i \in G(i, x_i)$, such that for each morphism $i \xrightarrow{u} j$ in I, we have $Ku(x_i) = x_j$ and $(Gu)(g_i) = g_j$.

The canonical morphism maps $((x_i)_{i \in I}, (g_i)_{i \in I})$ to the family $(x_i, g_i)_{i \in I}$. It is clearly a bijection. □

Proof of Lemma A.3. Let $d : I \to \mathcal{A}$ be a finite connected diagram and X be a functor preserving finite connected limits. Then,

$$\begin{aligned}
\int_{i:I} F(X)_{d_i} &= \int_{i:I} \coprod_n \coprod_{o \in O_n(d_i)} X_{\bar{o}_1} \times \cdots \times X_{\bar{o}_n} \\
&\cong \coprod_n \int_{i:I} \coprod_{o \in O_n(d_i)} X_{\bar{o}_1} \times \cdots \times X_{\bar{o}_n} && \text{(Coproducts commute with connected limits)} \\
&\cong \coprod_n \coprod_{o \in \int_i O_n(d_i)} \int_{i:I} X_{\overline{p_i(o)}_1} \times \cdots \times X_{\overline{p_i(o)}_n} && \text{(By Lemma A.10)} \\
&\cong \coprod_n \coprod_{o \in \int_i O_n(d_i)} \int_{i:I} X_{\overline{p_i(o)}_1} \times \cdots \times \int_{i:I} X_{\overline{p_i(o)}_n} && \text{(By commutation of limits)}
\end{aligned}$$

Thus, since X preserves finite connected limits by assumption,

$$\int_i F(X)_{d_i} = \coprod_n \coprod_{o \in \int_i O_n(d_i)} X_{\int_{i:I} \overline{p_i(o)}_1} \times \cdots \times X_{\int_{i:I} \overline{p_i(o)}_n} \tag{4}$$

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

$$
\begin{aligned}
\int_i F(X)_{d_i} &\cong \coprod_n \coprod_{o \in \int_i O_n(d_i)} X_{\int_{i:I} \overline{p_i(o)}_1} \times \cdots \times X_{\int_{i:I} \overline{p_i(o)}_n} && \text{(By Equation (4))} \\
&\cong \coprod_n \coprod_{o \in O_n(\lim d)} X_{\int_{i:I} \overline{o\{p_i\}}_1} \times \cdots \times X_{\int_{i:I} \overline{o\{p_i\}}_n} && \text{(By assumption on } O_n\text{)} \\
&\cong \coprod_n \coprod_{o \in O_n(\lim d)} X_{\overline{o}_1} \times \cdots \times X_{\overline{o}_n} && \text{(By Corollary A.9)} \\
&= F(X)_{\lim d}
\end{aligned}
$$

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.4, each O_n preserves finite connected limits. By Corollary A.9, it is enough to prove that given a finite connected diagram $d : D \to \mathcal{A}$ and element $o \in O_n(\lim d)$, the following canonical morphism is an isomorphism

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

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,

$$
\begin{aligned}
\int_{i:I} F(X)_{d_i} &\cong \coprod_n \coprod_{o \in \int_i O_n(d_i)} X_{\int_{i:I} \overline{p_i(o)}_1} \times \cdots \times X_{\int_{i:I} \overline{p_i(o)}_n} && \text{(By Equation (4))} \\
&= \coprod_n \coprod_{o \in O_n(\lim d)} X_{\int_{i:I} \overline{o\{p_i\}}_1} \times \cdots \times X_{\int_{i:I} \overline{o\{p_i\}}_n} && (O_n \text{ preserves finite connected limits})
\end{aligned}
$$

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

A.4 Lemma 3.22

Cocompleteness follows from Adámek and Rosicky [4, Remark 1.56], since $\mathscr{C}$ is the category of models of a limit sketch, and is thus locally presentable, by Adámek and Rosicky [4, 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 [3, Example 1.3.(vi)].

A.5 Corollary 3.23

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 $\emptyset \to H\emptyset \to HH\emptyset \to \ldots$, where $\emptyset$ denotes the constant functor mapping anything to the empty set, and $HZ = FZ + X$.

B Proof that $\mathcal{A}$ has finite connected limits (Section 7.4 on system F)

In this section, we show that the category $\mathcal{A}$ of arities for System F (Section §7.4) 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 $\mathcal{A}' \to \mathbb{F}_m$ 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 $f : n' \to n$ 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 [14, 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.

Lemma B.2. *$\mathcal{A}$ has finite connected limits.*

Proof. 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.8.

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 [7]. Thus, it includes the weak pullback of $\mathcal{A}' \to \mathbb{F}_m \leftarrow \int S$. But since $\int S \to \mathbb{F}_m$ is a fibration, and thus an isofibration, by [18] this weak pullback can be computed as a pullback, which is $\mathcal{A}$. □