JFP **35**, e26, 46 pages, 2026. © The Author(s), 2026. Published by Cambridge University Press. This is an Open Access article, distributed under the terms of the Creative Commons Attribution-NonCommercial-ShareAlike licence (http://creativecommons.org/licenses/by-nc-sa/4.0/), which permits non-commercial re-use, distribution, and reproduction in any medium, provided the same Creative Commons licence is used to distribute the re-used or adapted article and the original article is properly cited. The written permission of Cambridge University Press or the rights holder(s) must be obtained prior to any commercial use.

doi:10.1017/S0956796825100130

Semantics of pattern unification

AMBROISE LAFONT
LIX, CNRS, Inria, École polytechnique, Institut Polytechnique de Paris, Palaiseau, France
(*e-mail:* ambroise.lafont@lix.polytechnique.fr)

NEEL KRISHNASWAMI
University of Cambridge, Cambridge, UK
(*e-mail:* nk480@cl.cam.ac.uk)

Abstract

We propose a notion of syntax with metavariables that generalises Miller's decidable *pattern* fragment of second-order unification for simply typed λ-calculus. Using categorical semantics, we show that, under some conditions, a generalisation of Miller's unification algorithm applies. To illustrate our semantic analysis, we implemented our generic unification algorithm in Agda. The syntax with metavariables given as input of the algorithm is specified by a notion of signature generalising binding signatures, covering a wide range of examples, including ordered λ-calculus and (intrinsic) polymorphic syntax such as System F. Although we do not explicitly handle equations, we also tackle simply typed λ-calculus modulo β- and η-equations (Miller's original setting) by working on the syntax of normal forms.

1 Introduction

Unification deals with languages with *metavariables*. Let us assume that a language with metavariables comes with a well-formedness judgement of the shape $\Gamma; a \vdash t$, meaning that the term t is well formed in the *metavariable context* Γ and the *scope a.* What we call a scope depends on the language of interest: for a de Bruijn-encoded untyped syntax, it would be a mere natural number; for a simply typed syntax, it would be a pair of a list of types $\vec{\sigma}$ and a type τ to mean that t has type τ in the base context $\vec{\sigma}$. A *metavariable context,* or *metacontext*, is typically a list of metavariable symbols with their associated *arities*. Metacontexts should form a category whose morphisms are called *metavariable substitutions* or *metasubstitution*s. A metasubstitution σ between Γ and Δ should also induce a mapping $t \mapsto t[\sigma]$ sending terms well-formed in the metacontext Γ and scope a to terms well-formed in the metacontext Δ and same scope a.

Remark 1. *We consider substitutions oriented as in Barr & Wells (*1990*, Section 9.7) or Rydeheard & Burstall (*1988*, Section 8) instead of the common reverse convention as in categories with families (Dybjer,* 1996*), where substitutions from* Γ *to* Δ *induces a mapping from terms over* Δ *to terms over* Γ.

https://doi.org/10.1017/S0956796825100130 Published online by Cambridge University Press

A unification problem is specified by a pair of terms (t_1, t_2) such that $\Gamma; a \vdash t_i$ for $i \in \{1, 2\}$. A unifier for this pair is a metasubstitution $\sigma : \Gamma \to \Delta$ such that $t_1[\sigma] = t_2[\sigma]$, and a most general unifier (abbreviated as mgu) is a unifier σ such that given any other unifier δ, there exists a unique σ' such that $\delta = \sigma' \circ \sigma$. Equality is usually considered up to some equations – typically β/η-equations. In the present work, we avoid dealing with such equations by working with the syntax of normal forms (see, e.g., Section 7.3). The equality is thus completely syntactic, except for the arguments of metavariables, which may be of different nature than lists (they are sets in Section 7.1).

Example: first-order/second-order/pattern unification for an untyped syntax. Let us illustrate different standard versions of unification, starting from the example of a de Bruijn-encoded untyped syntax specified by a binding signature (Aczel, 1978, unpublished data). We take scopes and also metavariable arities to be natural numbers. A metavariable M is always applied to a list of arguments $\vec{t}$, whose length is specified by its arity. We can define three variants of unification by adding one of the following introduction rules for a metavariable M of arity $m \in \mathbb{N}$ in a scope $n \in \mathbb{N}$:

$$\forall (M : m) \in \Gamma \qquad \begin{array}{c|c|c} \text{First-order} & \text{Second-order} & \text{Pattern} \\ & & \overbrace{(t_1,\ldots,t_m) = \text{list of distinct variables}} \\ \dfrac{m = 0}{\Gamma; n \vdash M}\,\textsc{Fo} & \dfrac{\Gamma; n \vdash t_1 \;\ldots\; \Gamma; n \vdash t_m}{\Gamma; n \vdash M(\vec{t})}\,\textsc{So} & \dfrac{\overbrace{\Gamma; n \vdash t_1 \;\ldots\; \Gamma; n \vdash t_m}}{\Gamma; n \vdash M(\vec{t})}\,\textsc{Pat} \end{array}$$

The third *pattern* variant in the rules above was introduced by Miller (1991) as a decidable fragment of second-order unification (for simply typed λ-calculus modulo β- and η-equations): contrary to the latter case, a metavariable can only be applied to a *pattern*, that is, to a list of distinct variables.

In all of these situations, a *metasubstitution* σ between two metacontexts Γ and Δ is defined the same way: it maps each metavariable declaration $M : m$ in Γ to a term $\Delta; m \vdash \sigma_M$. Given a term $\Gamma; n \vdash t$, we define by recursion the substituted term $\Delta; n \vdash t[\sigma]$. Then, composition of metasubstitutions is defined by $(\sigma \circ \delta)_M = \delta_M[\sigma]$.

Motivation

Pattern unification is used in the implementation of various programming languages. As a concrete example, consider Dunfield–Krishnaswami's type inference algorithm for a variant of System F (Dunfield & Krishnaswami, 2019). It only involves first-order unification, but simply adding a monomorphic type with a binder (e.g., a recursive type $\mu a.A[a]$) would require pattern unification. In order to avoid reproving everything for each new type system, pattern unification needs to be formulated generically so that it can be used in a variety of contexts without modification. This is our original motivation for this work. To the best of our knowledge, we are the first to give a general definition of pattern unification that works for a wide class of languages, in the vein of Rydeheard–Burstall's first-order analysis (Rydeheard & Burstall, 1988); see the related work in Section 8 for more details.

https://doi.org/10.1017/S0956796825100130 Published online by Cambridge University Press

First contribution: a class of languages with metavariables

Our first contribution is a class of languages with metavariables. Such a language is specified by a *generalised binding signature*, or GB-signature, consisting of the following data:

- a small category $\mathcal{A}$ of *scopes* (or *metavariable arities)*,[1] and *renamings* between them,
- an endofunctor F on the category $[\mathcal{A}, \mathrm{Set}]$ of the shape

$$F(X)_a = \coprod_{n\in\mathbb{N}} \coprod_{o\in O_n(a)} X_{\overline{o}_1} \times \cdots \times X_{\overline{o}_n}, \tag{1.1}$$

  where
  - $[X, Y]$ denotes the category of functors from the category X to Y;
  - $\coprod$ denotes the coproduct in Set, which is disjoint union;
  - $O_n(a)$ is intuitively the set of available n-ary operation symbols in the scope a;
  - Each $o \in O_n(a)$ comes with a list of scopes $(\overline{o}_1, \ldots, \overline{o}_n)$, one for each argument of o.

The base syntax (in the empty metacontext) is generated by the following single rule:

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

This rule accounts for (possibly simply typed) binding arities (Aczel, 1978, unpublished data; Fiore & Hur 2010) but not only. In particular, in Section 7.3, we handle the syntax of normalised λ-terms, which cannot be specified by a binding signature.

We now present the full syntax with metavariables. Again, a metacontext is a list of metavariable symbols with their associated arities (or scopes). The syntax is generated by two rules, one for operations and one for metavariables:

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

Let us explain how the right rule instantiates to the above metavariable introduction rule PAT for pattern unification. A list of distinct variables $(x_1, \ldots, x_m)$ in the scope n is equivalently given by an injective map from $\{1, \ldots, n\}$ to $\{1, \ldots, m\}$. Therefore, by taking for $\mathcal{A}$ the category $\mathbb{F}_m$ whose objects are natural numbers and whose morphisms from n to m consist of injective maps as above, we recover the above rule PAT. Note that contrary to the traditional definition of pattern unification, 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."

Following the path sketched for the introductory example, we can define metasubstitutions, their action on terms, and their compositions: unification problems can then be stated.

[1] The fact that the notions of scopes and metavariable arities coincide (as in the previous example) allows us to see terms as substitutions and mgus as coequalisers, as we will see. This is not the case in Vezzosi–Abel's presentation of Miller's original setting (Vezzosi & Abel, 2014).

https://doi.org/10.1017/S0956796825100130 Published online by Cambridge University Press

Scope of our class of languages. We account for any syntax specified by a multi-sorted binding signature (Fiore & Hur, 2010): we detail the example of simply typed λ-calculus (without β- and η-equations) in Section 7.2. Note that our framework handles typed settings in such a way that knowing that $M(\vec{x})$ and $M(\vec{y})$ are well formed in the same metacontext and scope is enough to conclude that the types of $\vec{x}$ and $\vec{y}$ are the same.

As already said, our notion of language is more expressive than binding signatures: we mentioned in particular the syntax of normal forms for simply typed λ-calculus (see Section 7.3), which allows us to cover Miller's original setting. Our class also includes languages where terms bind type variables such as System F (Section 7.5.1): the scopes then include information about the available type variables. In another direction, we can handle certain kind of constraints on the variables in the context: in Section 7.4, we give a (novel) unification algorithm for the calculus for ordered linear logic described by Polakow & Pfenning (2000). Their notion of context consists of two components, one of which includes variables that must occur exactly once and in the same order as they occur in that context. Examples of languages that we handle are given in Section 7, where we show some traditional presentation of the calculi alongside with the corresponding GB-signatures.

Let us mention that *dynamic pattern unification (Reed,* 2009*; Abel & Pientka,* 2011*)* does not fit into the scope of this work. This variant deals with general second-order unification problems a priori, but the algorithm defers those which are outside the pattern fragment, hoping that they will eventually become so after solving the other ones. The main obstacle is that the specification is unclear: what does it mean for a dynamic pattern unification algorithm to be complete? This question needs to be solved first if we want to apply our methodology, in the line of Rydeheard and Burstall's account of first-order unification (Rydeheard & Burstall, 1988). Regarding the second-order flavour involved in dynamic pattern unification, the work of Hamana (2004) provides a potentially useful account of syntax with second-order metavariables, in terms of a monad on a presheaf category. In spirit, this is similar to our categorical analysis (see Lemma 30) except that their monad is not free.

Dynamic pattern unification is especially useful in the implementation of fully dependently typed languages (Gundry, 2013). Such languages, where types can depend on terms, are not supported. Indeed, intuitively, in our notion of specification, types are specified through the set of scopes, which must be given independently and prior to the endofunctor of terms: this sequential splitting is not possible with dependent types, unless we consider the untyped syntax separately from the (dependent) typing judgements. Accordingly, one possible future research direction would be to account for type safety of pattern unification in this situation, possibly exploiting the standard notion of signatures for dependent types theories as *second-order generalised algebraic theories* (Uemura, 2021, Chapter 4).

Second contribution: a unification algorithm for pattern-friendly languages

Our second key contribution consists of working out some conditions ensuring that the main contributions of Miller's work generalise: given two terms $\Gamma; a \vdash t, u$, either their mgu exists, or there is no unifier, and the proof of this statement consists in a recursive procedure (much similar to Miller's original algorithm) which computes a mgu or detects the absence of any unifier.

https://doi.org/10.1017/S0956796825100130 Published online by Cambridge University Press

Those conditions are essentially that renamings are monomorphic, and $\mathcal{A}$ has equalisers and pullbacks, and some additional properties about the functor F related to those limits (see Definition 25). We call one of our languages *pattern-friendly* when it satisfies those properties. All the examples that we already mentioned are pattern-friendly.

Unification as a total algorithm. We use a small trick to avoid the traditional presentation of unification as a partial algorithm computing mgus: we add a formal error metacontext $\bot$ and a single formal error term $\bot; a \vdash !$ for all scopes a, so that we get a unique[2] metasubstitution $!_\Gamma$ from any metacontext Γ to $\bot$. This substitution unifies any pair of terms since $t[!]$ is $!$ for any term t. If two terms are not unifiable in the traditional sense, $!$ is the mgu. If $\sigma : \Gamma \to \Delta$ is the mgu in the traditional sense, then it is still the mgu in this extended setting, because $!_\Gamma$ uniquely factors as $!_\Delta \circ \sigma$. In this way, unification can be seen as a total algorithm that always computes the mgu.

Agda implementation. We implemented our generic unification algorithm (without mechanisation of the correctness proof) in Agda. We show the most important parts; the interested reader can find the full implementation in the Supplementary Material. We used Agda as a programming language rather than a theorem prover. In particular, we did not enforce all the invariants in the definition of the data structures (e.g., associativity of composition in the category of scopes): the user has to check by themselves that the input data is valid for the algorithm to produce valid outputs. Furthermore, we disable the termination checker and provide instead a termination proof on paper in Section 6.1.

Most general unifiers as coequalisers

It is well-known that unification can be formulated categorically (Goguen, 1989). Let us make this formulation explicit in our setting. The set of terms in the metacontext Γ and scope a is recovered as the set of morphisms from the singleton metacontext $(M : a)$ to Γ. With this in mind, a unifier of two terms $\Gamma; a \vdash t, u$ can be interpreted as a cocone, that is, as a morphism $\Gamma \to \Delta$ such that its composition with either of the two terms (interpreted as morphisms) are equal. A mgu is then a coequaliser: this is the characterisation that we use to prove correctness of our unification algorithm.

Let us finally mention that given a specification, we provide in Proposition 33 a direct characterisation of the category of metacontexts and substitutions as a full subcategory of the Kleisli category of the monad T freely generated by the endofunctor F.

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.

[2] This characterises $\bot$ as the terminal object in the category of metacontexts, with $!_\Gamma$ denoting the terminal morphism from Γ.

https://doi.org/10.1017/S0956796825100130 Published online by Cambridge University Press

```
MetaContext· = List ℕ
MetaContext = Maybe MetaContext·

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

- A renaming should be injective
- so that, in particular, m <= n.
hom : ℕ → ℕ → Set
hom m n = Vec (Fin n) m
```

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

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

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

```
App : ∀ {Γ n} → Tm Γ n →
      Tm Γ n → Tm Γ n
App {⊥} ! ! = !
App {⌊ Γ ⌋} t u = App· t u

Lam : ∀ {Γ n} → Tm Γ (1 + n)
    → Tm Γ n
Lam {⊥} ! = !
Lam {⌊ Γ ⌋} t = Lam· t

Var : ∀ {Γ n} → Fin n
    → Tm Γ n
Var {⊥} i = !
Var {⌊ Γ ⌋} i = Var· i
```

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

General notations

Given a list $\vec{x} = (x_1, \ldots, x_n)$ and a list of positions $\vec{p} = (p_1, \ldots, p_m)$ taken in $\{1, \ldots, n\}$, we denote $(x_{p_1}, \ldots, x_{p_m})$ 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 $\coprod_{i \in I} A_i$. Similarly, the morphism $A + B \to A' + B'$ induced by $f : A \to A'$ and $g : B \to B'$ is denoted by $f + g$, and the morphism $\coprod_{i \in I} A_i \to \coprod_{i \in I} A'_i$ induced by a family $(f_i : A_i \to A'_i)_{i \in I}$ is denoted by $\coprod_i f_i$. 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.

2 Presentation of the algorithm

In Section 2.1, we start by describing a pattern unification algorithm for pure λ-calculus. We claim no originality here; minor variants of the algorithm can be found in the literature: it serves mainly as an introduction to the generic algorithm presented in Section 2.2. Both algorithms are summarised side by side at the end of this section in Figures 10 and 11 for comparison.

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

https://doi.org/10.1017/S0956796825100130 Published online by Cambridge University Press

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 $\bot$, or a *proper* context, as a list $(M_1 : m_1, \ldots, M_p : m_p)$, of metavariable declarations specifying metavariable symbols M_i together with their arities, that is, their number of arguments m_i;
2. a scope, which is a mere natural number indicating the highest possible free variable.

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 $\mathbb{N}$), 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$\cdot$ and Tm$\cdot$ Γ n are, respectively, defined as List $\mathbb{N}$ and Tm $\lfloor \Gamma \rfloor$ n.

Free variables are indexed from 1, and we use the *de Bruijn level* convention: the variable bound in $\mathbf{\Gamma}; n \vdash \lambda t$ is $n + 1$, not 0, as it would be using *de Bruijn indices* (De Bruijn, 1972). In Agda, variables in the scope n consist of elements of Fin n, the type of natural numbers between[3] 1 and n.

Remark 2. *De Bruijn levels are one possible convention for interpreting natural numbers as variables, which allows us to view a mere list of variables* $(t_1, \ldots, t_n)$ *as a renaming, replacing the variable i with* t_i*. Moreover, capture-avoiding renaming can be implemented naively as syntactic substitution, contrary to the convention based on de Bruijn indices.*

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, that is, 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.

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 scope n, that is, natural numbers between 1 and n. We denote by $\hom(m, n)$ 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.

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

https://doi.org/10.1017/S0956796825100130 Published online by Cambridge University Press

```
- Proper substitutions
Γ ·⟶ Δ = ⌊ Γ ⌋ ⟶ Δ

- Successful substitutions
Γ ·⟶· Δ = ⌊ Γ ⌋ ⟶ ⌊ Δ ⌋

data _⟶_ where
  []  : ∀ {Δ} → ([] ·⟶ Δ )
  _,_ : ∀ {Γ Δ m} → Tm Δ m → (Γ ·⟶ Δ) → (m :: Γ ·⟶ Δ)
  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
- _{_} : Tm Γ n → hom n p → Tm Γ p
- is renaming (code omitted)
M ( x ) [ σ ]t = nth σ M { x }
! [ 1⊥ ]t = !

_[_]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⊥
```

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

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

Fig. 2. Metavariable substitution for λ-calculus (Section 2.1)

The Agda implementation of metavariable substitutions for λ-calculus is listed in the first box of Figure 2. We call a substitution *proper* if the domain is proper, *successful* if the target is also proper. Note that a substitution is successful if and only if the target is proper, because there is only one metavariable substitution $1_\perp$ from the error context: it is the formal identity substitution, targeting itself. A *metavariable substitution* $\sigma : \mathbf{\Gamma} \to \Delta$ 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 $\Delta; n \vdash t[\sigma]$. The congruence cases involve improper versions of the operations, 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: it replaces each variable i by x_i. Renaming a λ-abstraction requires extending the renaming $x : \mathsf{hom}\ p\ q$ to $x\uparrow : \mathsf{hom}\ (p+1)\ (q+1)$ 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 scope.

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 $M \mapsto \delta_M[\sigma]$.

https://doi.org/10.1017/S0956796825100130 Published online by Cambridge University Press

We write $\Gamma \vdash t = u \Rightarrow \sigma \dashv \Delta$ to mean that σ is a *most general unifier* (mgu) of t and u, as in Section 1. More explicitly, a *unifier* of two terms $\Gamma; n \vdash t, u$ is a substitution $\sigma : \Gamma \to \Delta$ such that $t[\sigma] = u[\sigma]$. We call it successful if the underlying substitution is. A mgu $\sigma : \Gamma \to \Delta$ of t and u is a unifier that uniquely factors any other unifier $\delta : \Gamma \to \Delta'$, in the sense that there exists a unique $\delta' : \Delta \to \Delta'$ such that $\delta = \sigma[\delta']$.

In the notation $\Gamma \vdash t = u \Rightarrow \sigma \dashv \Delta$, the symbol $\Rightarrow$ separates the input and the output of the unification algorithm. Indeed, as can be seen in Figure 3, the unify function takes two terms $\Gamma; n \vdash t, u$ as input and returns 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 a mgu of t and u (the mgu property is however not explicitly enforced by the type signature).

This unification function recursively inspects the structure of the given terms until reaching a metavariable at the top level, as seen in the box of Figure 3. 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 are considered by the auxiliary function unify-flex-*:

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 $x\ y$ computes the *vector of common positions*[4] 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[5] such a situation by $m \vdash x = y \Rightarrow z \dashv p$. The most general unifier σ coincides with the identity substitution except that the declaration $M : m$ is replaced by a metavariable declaration $P : p$ in the context $\mathbf{\Gamma}$, and σ maps M to $P(z)$.

Example 3. *Consider unification of $M(x, y)$ and $M(z, x)$, where x, y, z are three distinct natural numbers. 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 $M \mapsto P(\underline{2})$.*

Recall that in the Agda implementation, metavariables are natural numbers referring to positions in the metacontext, which is just a list of arities. In the Agda code, $\Gamma[M : p]$ denotes the metacontext Γ where the M^{th} element has been replaced with p. Furthermore, $(M : p)(u')$ denotes the metavariable M applied to y', where the annotation $: p$ is an explicit coercion making M a metavariable of $\Gamma[M : p]$ rather than of Γ. Finally, the output

[4] The function commonPositions is defined by recursion on the common size of the vectors.

[5] The similarity with the above introduced notation is no coincidence: as we will see (Remark 21), both are (co)equalisers.

https://doi.org/10.1017/S0956796825100130 Published online by Cambridge University Press

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

unify : ∀ {Γ a} → Tm Γ a → Tm Γ a → Γ ⟶?
```

```
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
```

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

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

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

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

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

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

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

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

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

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

(+ symmetric rules)

```
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. 3. Unification for λ-calculus

https://doi.org/10.1017/S0956796825100130 Published online by Cambridge University Press

metasubstitution σ between Γ and $\Gamma[M : p]$, denoted by $M \mapsto -(x)$ in the code, coincide with the identity substitution of Γ, except for σ_M which is defined as $M(x)$.

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 (successful) 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, listed in Figure 4. 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 an *outbound variable,* that is, a variable that does not occur in x, then 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 variable y.

The pruning phase runs in the metacontext with M removed. We use the notation $\Gamma \vdash t \mathrel{:>} x \Rightarrow t'; \sigma \dashv \Delta$, 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 $\sigma : \Gamma \to \Delta$. 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 prune.

The 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 $x\ y$ 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 $y_1, \ldots, y_{m'}$, that is, the pair of maximal lists $(\vec{x'}, \vec{y'})$ of distinct positions such that $x_{\vec{x'}} = y_{\vec{y'}}$. We denote[6] such a situation by $m \vdash x \mathrel{:>} y \Rightarrow y'; x' \dashv p$. 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 4. *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 implementation merely replaces the metavariable arity of M with p, using the same Agda notations as in the auxiliary function unify-flex-* in Figure 3.

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

[6] 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 50.

https://doi.org/10.1017/S0956796825100130 Published online by Cambridge University Press

```
record [_]∪_ ⟶? m Γ : Set k' where
  constructor _◀_;_
  field
    Δ : MetaContext
    u : Tm Δ m
    σ : Γ ⟶ Δ

prune : ∀ {Γ a m} → Tm Γ a → hom m a → [ m ]∪ Γ ⟶?
```

```
prune {⌊ Γ ⌋} (M ( x )) y =
  let p , x' , y' = commonValues 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
```

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

$$\frac{}{\bot \vdash ! :> x \Rightarrow !; !_s \dashv \bot}\ \text{P-FAIL}$$

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

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

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

Fig. 4. Pruning for λ-calculus

with t and u. We can process those unification problems in order. If the first one yields t' and the substitution σ_1, and then we refine the second unification problem by replacing u with $u[\sigma_1]$. Assuming the output of this second unification is u' 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.

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

https://doi.org/10.1017/S0956796825100130 Published online by Cambridge University Press

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

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 our new notion of specification 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 23), 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 25).

This separation is motivated by the fact that in the case of λ-calculus, the vectors of common (value) positions are involved in the algorithm, but not in the definition of the syntax and associated operations (renaming, metavariable substitution).

A GB-signature consists in a tuple $(\mathcal{A}, O, \alpha)$ consisting of

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

such that O and α are functorial in a suitable sense. In particular, given a morphism $x\colon a \to b$ in $\mathcal{A}$ and operation symbol $o \in O(a)$ with $\alpha_o = (\overline{o}_1, \ldots, \overline{o}_n)$, there is

- an operation symbol $o\{x\}$ in $O(b)$;
- a *vector* $x^o : \alpha_o \dashrightarrow \alpha_{o\{x\}}$ *of renamings*, that is, a list $(x_1^o, \ldots, x_n^o)$ of morphisms in $\mathcal{A}$ such that $x_i^o : \overline{o}_i \to \overline{o\{x\}}_i$.

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 recursively define $\Gamma; b \vdash t\{x\}$ by $M(x \circ y)$ if $t = M(y)$, or by $o\{x\}(t_1\{x_1^o\}, \ldots, t_n\{x_n^o\})$ if $t = o(t_1, \ldots, t_n)$.

Remark 5. *This definition of GB-signatures superficially differs from the notion of specification that we mention in the introduction: here, the set of operation symbols $O(a)$ in a scope a is not indexed by natural numbers. The two descriptions are equivalent: $O_n(a)$*

https://doi.org/10.1017/S0956796825100130 Published online by Cambridge University Press

```
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. 5. Generalised binding signatures in Agda

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. From these data, we can define an endofunctor F as in Equation (1.1).

The Agda implementation in Figure 5 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 6. *We give the signature for pure λ-calculus. As explained in the introduction, we take $\mathcal{A} = \mathbb{F}_m$. In the scope n, we have n nullary available operation symbols (one for each variable), one unary operation abs^n, and one binary operation app^n, so that $O(n) = \{1, \ldots n, abs^n, app^n\}$, with associated arities $\alpha_i = ()$, $\alpha_{abs^n} = (n+1)$ and $\alpha_{app^n} = (n, n)$. The corresponding Agda implementation can be found in Figure 6.*

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 scopes and metavariable types are objects of $\mathcal{A}$ instead of natural numbers.

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

Remark 7. *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 (Altenkirch & Morris,* 2009*) generating the same syntax without metavariables. Note that indexed containers correspond to indexed datatypes in type theory.*

Recall that the Agda code uses a nameless convention for metacontexts: they are just lists of scopes. 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

https://doi.org/10.1017/S0956796825100130 Published online by Cambridge University Press

```
data O n : Set where
  Var : Fin n → O n
  App : O n
  Lam : O n

α : {n : ℕ} → O n → List ℕ
α (Var x) = []
α {n} App = n :: n :: []
α {n} Lam = 1 + n :: []

_{_} : ∀ {a b : ℕ} → O a → hom a b → O b
Var x { s } = Var (Vec.lookup s x)
App { s } = App
Lam { s } = Lam

- Pointwise hom [a₁, ⋯, aₙ] [b₁, ⋯, bₙ] is the type of the
- lists of the shape [c₁, ⋯, cₙ] with cᵢ : hom aᵢ bᵢ
_^_ : {a b : ℕ} (x : hom a b) (o : O a) → Pointwise hom (α o) (α (o { x }))
x ^ Var y = []
x ^ App = x :: x :: []
x ^ Lam = (x ↑) :: []
```

Fig. 6. Implementation of the signature of pure λ-calculus

```
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
```

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

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

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

Fig. 7. Syntax generated by a GB-signature

a metavariable substitution from $\alpha_o = (\overline{o}_1, \ldots, \overline{o}_n)$ to Γ, as in the Agda code, or explicitly as a list of terms $(t_1, \ldots, t_n)$ such that $\Gamma; \overline{o}_i \vdash t_i$, as in the rule RIG. In the following, we will use either interpretation.

Since metasubstitutions occur in the syntax of terms, we define substitution on terms mutually with composition of metasubstitutions in Figure 8. We are similarly led to mutually define unification of terms and unification of metasubstitutions. Given two substitutions $\delta_1, \delta_2 : \Gamma' \to \Gamma$, we write $\Gamma \vdash \delta_1 = \delta_2 \Rightarrow \sigma \dashv \Delta$ to mean that $\sigma : \Gamma \to \Delta$

https://doi.org/10.1017/S0956796825100130 Published online by Cambridge University Press

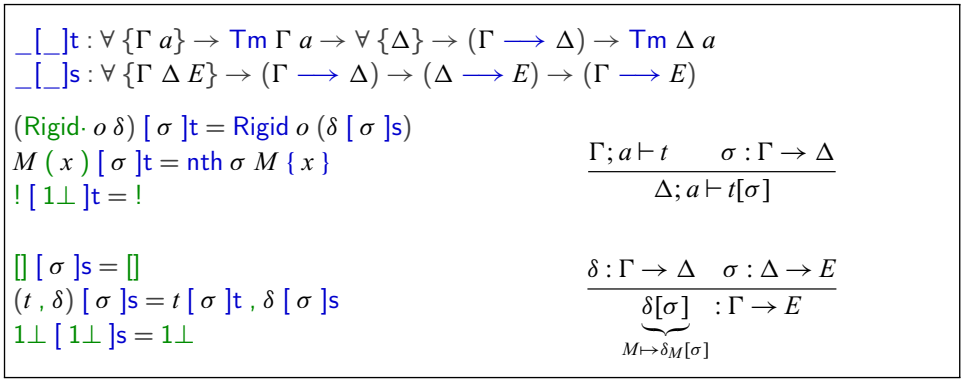


Fig. 8. Metavariable substitution for a GB-signature (Section 2.2)

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

record _∪_⟶? (Γ'' : MetaContext·)(Γ : MetaContext)
    : Set (i ⊔ j ⊔ k) where
  constructor _◀_;_
  field
    Δ : MetaContext
    δ : Γ'' ·⟶ Δ
    σ : 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 ⟶?
```

Fig. 9. Type signatures of unification and pruning

unifies δ_1 and δ_2, in the sense that $\delta_1[\sigma] = \delta_2[\sigma]$, and is the most general one, that is, it uniquely factors any other unifier of δ_1 and δ_2. In Figure 9, we list the type signatures of the functions involved in the algorithm: the main unification function is 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, using the below notion of vector renaming for metacontexts.

Definition 8. *If* $\mathbf{\Gamma}$ *and* $\mathbf{\Delta}$ *are two proper metacontexts* $M_1 : m_1, \ldots, M_p : m_p$ *and* $N_1 : n_1, \ldots, N_p : n_p$ *of the same length, a vector of renamings* $\delta : \mathbf{\Gamma} \dashrightarrow \mathbf{\Delta}$ *between* $\mathbf{\Gamma}$ *and* $\mathbf{\Delta}$ *is a list* $(\delta_1, \ldots, \delta_p)$ *such that each* δ_i *is a morphism between* m_i *and* n_i*. Such a vector canonically induces a metavariable substitution* $\overline{\delta} : \mathbf{\Delta} \to \mathbf{\Gamma}$*, mapping* N_i *to* $M_i(\delta_i)$*.*

This is is compatible with the notion of vector of renamings that we introduced at the beginning of the section, if we take metacontexts to be mere lists of scopes, as in the Agda implementation.

https://doi.org/10.1017/S0956796825100130 Published online by Cambridge University Press

Given a substitution $\delta : \mathbf{\Gamma'} \to \Gamma$ and a vector $x : \mathbf{\Gamma''} \dashrightarrow \mathbf{\Gamma'}$ of renamings, the judgement $\Gamma \vdash \delta \mathbin{:>} x \Rightarrow \delta'; \sigma \dashv \Delta$ means that the substitution $\sigma : \Gamma \to \Delta$ extended with $\delta' : \mathbf{\Gamma''} \to \Delta$ is the most general unifier of δ and $\bar{x}$ as substitutions from $\mathbf{\Gamma''}, \Gamma$ to Δ, where $\mathbf{\Gamma''}, \Gamma$ is the concatenation of contexts if Γ is proper, or $\bot$ otherwise.

The unification algorithm is summarised in Figure 10, next to the λ-calculus implementation (Figure 11) for comparison. In that special case, 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 of natural numbers and injective renamings between them, 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 $m \vdash x \mathbin{:>} y \Rightarrow y'; x' \dashv p$ as equalisers and pullbacks.

Notation 9. *We write* $m \vdash x = y \Rightarrow z \dashv p$ *and* $m \vdash x \mathbin{:>} y \Rightarrow y'; x' \dashv p$ *to respectively denote an equaliser and pullback in* $\mathcal{A}$ *as below.*

$$p \xrightarrow{z} m \underset{y}{\overset{x}{\rightrightarrows}} \cdots \qquad \begin{array}{ccc} p & \xrightarrow{x'} & m \\ {\scriptstyle y'}\downarrow & & \downarrow{\scriptstyle x} \\ \cdots & \xrightarrow[y]{} & \cdots \end{array}$$

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 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. The algorithm relies on GB-signatures with additional components listed in Figure 12. We call such GB-signatures *binding-friendly*. 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 25) includes additional properties ensuring that the algorithm is correct. One consequence of those properties is that inverse renaming is unique: there is at most one preimage by renaming.

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.

https://doi.org/10.1017/S0956796825100130 Published online by Cambridge University Press

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

Same as the rule P-FLEX in Figure 11.

```
prune ! y = ⊥ ◀ ! ; !s
```

Same as the rule P-FAIL in Figure 11.

```
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
```

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

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

```
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)
```

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

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

unify-flex-* is defined as in Figure 11, replacing commonPositions with equaliser, and calling the above prune function instead of the one for λ-calculus.

```
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 11.

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

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

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

```
unify ! ! = ⊥ ◀ !s
```

Same as the rule U-FAIL in Figure 11.

```
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
```

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

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

$$\frac{}{\bot \vdash 1_\bot = 1_\bot \Rightarrow !_s \dashv \bot}\text{U-ID-FAIL}$$

Fig. 10. Our generic pattern unification algorithm

https://doi.org/10.1017/S0956796825100130 Published online by Cambridge University Press

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

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

```
prune ! y = ⊥ ◀ ! ; !_s
```

$$\frac{}{\bot \vdash ! :> x \Rightarrow !; !_s \dashv \bot}\ \text{P-FAIL}$$

```
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)
```

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

```
prune (Lam· t) x =
  let Δ ◀ t' ; σ = prune t (x ↑)
  in Δ ◀ Lam t' ; σ
```

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

```
prune {Γ} (Var· i) x with i { x }⁻¹
... | ⊥ = ⊥ ◀ ! ; !_s
... | ⌊ PreImage j ⌋ = Γ ◀ Var j ; 1_s
```

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

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

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

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

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

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

(+ symmetric rules)

```
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
```

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

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

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

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

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

```
unify ! ! = ⊥ ◀ !_s
```

$$\frac{}{\bot \vdash ! = ! \Rightarrow !_s \dashv \bot}\ \text{U-FAIL}$$

```
unify _ _ = ⊥ ◀ !_s
```

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

Fig. 11. Pattern unification for λ-calculus (Section 2.1)

https://doi.org/10.1017/S0956796825100130 Published online by Cambridge University Press

```
record isFriendly {i j k}(S : Signature i j k) : Set (i ⊔ j ⊔ k) where
field
  equaliser : ∀ {m a} → (x y : hom m a) → Σ A (λ p → hom p m)
  pullback : ∀ {m m' a} → hom m a → 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)
```

Fig. 12. Pattern-friendly GB-signatures in Agda

3.1 Pattern unification as a coequaliser construction

In this section, we assume given a GB-signature S 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 (Rydeheard & Burstall, 1988; Barr & Wells, 1990). We furthermore provide a formal justification for the error metacontext $\bot$.

Lemma 10. *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 Proposition 33 that this category fully faithfully embeds in a Kleisli category for a monad generated by S on $[\mathcal{A}, \mathrm{Set}]$.

Remark 11. *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.*

Lemma 12. *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 20, after explaining how failure is handled categorically. 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 $\bot$, resulting in the full category of metacontexts and substitutions.

Definition 13. *Given a category* $\mathcal{B}$, *let* $\mathcal{B}_\bot$ *denote the category* $\mathcal{B}$ *extended freely with a terminal object* $\bot$.

Notation 14. *We denote by* $!_s$ *any terminal morphism to* $\bot$ *in* $\mathcal{B}_\bot$.

https://doi.org/10.1017/S0956796825100130 Published online by Cambridge University Press

Lemma 15. *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 16. *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 17. *Let $\mathscr{B}$ be a category.*

(i) The canonical embedding functor $\mathscr{B} \to \mathscr{B}_\perp$ preserves 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 $\perp$.

Corollary 18. *Coproducts in* $\mathrm{MCon}(S)$*, computed as concatenation of metacontexts, are also coproducts in* $\mathrm{MCon}(S)_\perp$*. Coproduct of any metacontext with $\perp$ is $\perp$.*

Accordingly, one can define informally "concatenation" of a metacontext with $\perp$ as $\perp$. The main property of this extension for our purposes is the following corollary.

Corollary 19. *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 20. $\Gamma \vdash \delta_1 = \delta_2 \Rightarrow \sigma \dashv \Delta$ *denotes a coequaliser* $\ldots \overset{\delta_1}{\underset{\delta_2}{\rightrightarrows}} \Gamma \xrightarrow{\sigma} \Delta$ *in* $\mathrm{MCon}(S)_\perp$.

Remark 21. *This is the same interpretation as in Notation 9 for equaliser, taking $\mathscr{A}$ 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 25).

Theorem 22. *Given any pattern-friendly signature S, the category* $\mathrm{MCon}(S)_\perp$ *has coequalisers.*

https://doi.org/10.1017/S0956796825100130 Published online by Cambridge University Press

3.2 Initial algebra semantics for GB-signatures

The proofs of various statements presented in this section are detailed in Section 3.3.

Definition 23. *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 family of functors* $(\alpha_{n,i} : \int O_n \to \mathcal{A})_{n,i \leq n}$ *indexed by natural numbers* i, n *such that* $1 \leq i \leq n$,

where $\int O_n$ *denotes the category of elements of* $O_n : \mathcal{A} \to \mathrm{Set}$*, defined as follows:*

- *objects are pairs* (a, o) *such that* $o \in O_n(a)$
- *a morphism between* (a, o) *and* (a', o') *is a morphism* $f : a \to a'$ *such that* $o\{f\} = o'$ *where* $o\{f\}$ *denotes the image of* o *by the function* $O_n(f) : O_n(a) \to O_n(a')$.

Notation 24. *Given a GB-signature* $(\mathcal{A}, O, \alpha)$ *and* $o \in O_n(a)$*, we write* $\overline{o}_j$ *for* $\alpha_{n,j}(o)$ *and* α_o *for the tuple* $(\overline{o}_1, \ldots, \overline{o}_n)$.

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

Definition 25. *A GB-signature* $S = (\mathcal{A}, O, \alpha)$ *is said to be pattern-friendly if*

1. $\mathcal{A}$ *has finite connected limits (or equivalently,* $\mathcal{A}$ *has pullbacks and equalisers);*
2. *all morphisms in* $\mathcal{A}$ *are monomorphic;*
3. *each* $O_n(-) : \mathcal{A} \to \mathrm{Set}$ *preserves finite connected limits;*
4. *each* $\alpha_{n,i}$ *preserves finite connected limits.*

Remark 26. *As a counter-example to the third condition, take* $\mathcal{A}$ *to be the category* $a \xrightarrow{f} b$ *consisting of two objects and one non-identity morphism between them, and consider the syntax generated by two nullary operations in scope* a *and one nullary operation* $*$ *in scope* b*. The first two conditions of Definition 25 are met, and the fourth one vacuously holds. The third condition is not satisfied for* $n = 0$ *and the following pullback, essentially because* $O_0(f)$ *is not injective.*

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

Note that $M(f) \stackrel{?}{=} *$ *has two unifiers given by the two operations in scope* a*, but none of them factors the other and so there is no most general unifier.*

These conditions ensure the following two properties.

https://doi.org/10.1017/S0956796825100130 Published online by Cambridge University Press

Property 27 (proved in Section 3.3.1)**.** *The following properties hold for pattern-friendly signatures.*

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

(ii) Let $\mathcal{L}$ *be the functor* $\mathcal{A}^{op} \to \mathrm{MCon}(S)_\perp$ *mapping a morphism* $x \in \hom_{\mathcal{A}}(b, a)$ *to the substitution from* $(X : a)$ *to* $(X : b)$ *consisting of the single 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. It is not satisfied in the counter-example of Remark 26. 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.

Lemma 28. *A metavariable application* $\mathbf{\Gamma}; a \vdash M(x)$ *corresponds to the composition* $\mathcal{L}x[in_M]$ *as a substitution from* $X : a$ *to* $\mathbf{\Gamma}$*, where* in_M *is the coproduct injection* $(X : m) \xrightarrow{\cong} (M : m) \hookrightarrow \mathbf{\Gamma}$ *mapping* M *to* $M(1_m)$*.*

In the rest of this section, we provide Initial Algebra Semantics for the generated syntax (this is used in the proof of Assumption 27.(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 29 (proved in Section 3.3.2)**.** *F is finitary and generates a free monad T. Moreover, TX is the initial algebra of $Z \mapsto X + FZ$.*

Lemma 30. *The proper syntax generated by a GB-signature (see Figure 7) is recovered as free algebras for F. More precisely, given a metacontext $\mathbf{\Gamma} = (M_1 : m_1, \ldots, M_p : m_p)$,*

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

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

Notation 31. *Given a proper metacontext $\mathbf{\Gamma}$. We sometimes denote $\underline{\mathbf{\Gamma}}$ just by $\mathbf{\Gamma}$.*

If $\mathbf{\Gamma} = (M_1 : m_1, ..., M_p : m_p)$ and $\mathbf{\Delta}$ are metacontexts, a Kleisli morphism $\sigma : \mathbf{\Gamma} \to T\mathbf{\Delta}$ is equivalently given (by combining the above lemma, the Yoneda Lemma, and the universal property of coproducts) by a metavariable substitution from $\mathbf{\Gamma}$ to $\mathbf{\Delta}$. 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 (see Section 1 for a definition of the latter category).

https://doi.org/10.1017/S0956796825100130 Published online by Cambridge University Press

Notation 32. *We denote the Kleisli category of a monad T on $\mathscr{B}$ 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 $f[g] : A \to TC$.*

Proposition 33. *The category* MCon(S) *is equivalent to the full subcategory of Kl_T spanned by coproducts of representable functors.*

Remark 34. *It follows from Proposition 33 and (Mac Lane, 1998, Exercise VI.5.1) that* MCon(S) *fully faithfully embeds in the category of algebras of T, by mapping a metacontext $\mathbf{\Gamma}$ to the free algebra $T\mathbf{\Gamma}$. In fact,* MCon(S)$_\perp$ *also fully faithfully embeds in the category of algebras by mapping $\perp$ to the terminal algebra, whose underlying functor maps any object of $\mathscr{A}$ to a singleton set.*

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

Notation 35. *Given a GB-signature $S = (\mathscr{A}, \mathcal{O}, \alpha)$, we denote the full subcategory of* [$\mathscr{A}$, Set] *consisting of functors preserving finite connected limits by $\mathscr{C}_S$, or sometimes by $\mathscr{C}$, leaving S implicit.*

Lemma 36 (proved in Section 3.3.3)**.** *Given a GB-signature $S = (\mathscr{A}, \mathcal{O}, \alpha)$ such that $\mathscr{A}$ has finite connected limits, F_S restricts as an endofunctor on $\mathscr{C}_S$ if and only if the last two conditions of Definition 25 hold.*

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

Lemma 37 (proved in Section 3.3.4)**.** *$\mathscr{C}$ is closed under limits, coproducts, and filtered colimits. Moreover, it is cocomplete.*

Corollary 38 (proved in Section 3.3.5)**.** *T restricts as a monad on $\mathscr{C}$ freely generated by the restriction of F as an endofunctor on $\mathscr{C}$ (Lemma 36).*

3.3 Proofs of statements in Section 3.2

3.3.1 Property 27

We use the notations and definitions of Section 3.2.

Let us first prove the first item.

Proof of Property 27.(i)

We show that given any $o \in \mathcal{O}_n(b)$ and renaming $f : a \to b$, there is at most one $o' \in \mathcal{O}_n(a)$ such that $o = o'\{f\}$.

Note that a morphism $f : a \to b$ is monomorphic if and only if the following square is a pullback (Mac Lane, 1998, Exercise III.4.4), as shown by unfolding the universal property

https://doi.org/10.1017/S0956796825100130 Published online by Cambridge University Press

of this particular limit.

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

Therefore, since O_n preserves finite connected limits, it also preserves monomorphisms. ■

The rest of this section is devoted to the proof of Property 27.(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 39. *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

$$y = J \circ K. \tag{3.1}$$

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

$$\begin{aligned}
\mathscr{C}(K \lim F, X) &\cong [\mathcal{A}, \mathrm{Set}](JK \lim F, JX) && (J \text{ is fully faithful}) \\
&\cong [\mathcal{A}, \mathrm{Set}](y \lim F, JX) && (\text{By Equation (3.1)}) \\
&\cong JX(\lim F) && (\text{By the Yoneda Lemma.}) \\
&\cong \lim(JX \circ F) && (X \text{ preserves finite connected limits}) \\
&\cong \lim([\mathcal{A}, \mathrm{Set}](yF-, JX)] && (\text{By the Yoneda Lemma}) \\
&\cong \lim([\mathcal{A}, \mathrm{Set}](JKF-, JX)] && (\text{By Equation (3.1)}) \\
&\cong \lim \mathscr{C}(KF-, X) && (J \text{ is full and faithful}) \\
&\cong \mathscr{C}(\mathrm{colim}\, KF, X) && (\text{By left continuity of the hom-set bifunctor})
\end{aligned}$$

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

Proof of Assumption 27.(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 17.(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 38. Since $K : \mathscr{D} \to \mathscr{C}$ preserves finite connected colimits (Lemma 39), 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. ■

https://doi.org/10.1017/S0956796825100130 Published online by Cambridge University Press

3.3.2 Lemma 29

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

3.3.3 Lemma 36

Notation 40. *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 $\operatorname{colim} F$) 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 41. *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 $\alpha_{n,i} : \int O_n \to \mathcal{A}$ preserves finite connected limits.*

We first introduce a bunch of intermediate lemmas.

Lemma 42. *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 43. *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 42 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 39. 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 44. *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 limits. Then, $\int F$*

https://doi.org/10.1017/S0956796825100130 Published online by Cambridge University Press

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 43, 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 45. *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 41 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*

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

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

Proof Note that, by definition, $\overline{o}_j = \alpha_{n,j}(\lim d, o)$, and $\overline{o\{p_i\}}_j = \alpha_{n,j}(d_i, o\{p_i\})$. This is a direct application of Corollary 44. ■

Lemma 46 (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. ■

Lemma 47. *A coproduct* $\coprod_i G_i$ *of functors from a small category* $\mathcal{B}$ *with finite connected limits to Set preserves those limits if and only if each* G_i *does.*

Proof This is a consequence of the following statement, which is a direct application of Adámek *et al*. (2002, Theorem 2.4 and Example 2.3.(iii)): 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 $\int G$ is a coproduct of filtered categories. ■

Proof of Lemma 41 Let $d : I \to \mathcal{A}$ be a finite connected diagram and X be a functor preserving finite connected limits. Then,

https://doi.org/10.1017/S0956796825100130 Published online by Cambridge University Press

$$\int_{i:I} F(X)_{d_i} = \int_{i:I} \coprod_n \coprod_{o \in \mathcal{O}_n(d_i)} X_{\overline{o}_1} \times \cdots \times X_{\overline{o}_n}$$

$$\cong \coprod_n \int_{i:I} \coprod_{o \in \mathcal{O}_n(d_i)} X_{\overline{o}_1} \times \cdots \times X_{\overline{o}_n}$$

(Coproducts commute with connected limits)

$$\cong \coprod_n \coprod_{o \in \int_i \mathcal{O}_n(d_i)} \int_{i:I} X_{\overline{p_i(o)}_1} \times \cdots \times X_{\overline{p_i(o)}_n} \qquad \text{(By Lemma 46)}$$

$$\cong \coprod_n \coprod_{o \in \int_i \mathcal{O}_n(d_i)} \int_{i:I} X_{\overline{p_i(o)}_1} \times \cdots \times \int_{i:I} X_{\overline{p_i(o)}_n} \qquad \text{(By commutation of limits)}$$

Thus, since X preserves finite connected limits by assumption,

$$\int_i F(X)_{d_i} = \coprod_n \coprod_{o \in \int_i \mathcal{O}_n(d_i)} X_{\int_{i:I} \overline{p_i(o)}_1} \times \cdots \times X_{\int_{i:I} \overline{p_i(o)}_n} \tag{3.2}$$

Now, let us prove the only if statement first. Assume that each $\mathcal{O}_n$ and $\alpha_{n,i} : \int \mathcal{O}_n \to \mathcal{A}$ preserve finite connected limits. Then,

$$\int_i F(X)_{d_i} \cong \coprod_n \coprod_{o \in \int_i \mathcal{O}_n(d_i)} X_{\int_{i:I} \overline{p_i(o)}_1} \times \cdots \times X_{\int_{i:I} \overline{p_i(o)}_n} \qquad \text{(By Equation (3.2))}$$

$$\cong \coprod_n \coprod_{o \in \mathcal{O}_n(\lim d)} X_{\int_{i:I} \overline{o\{p_i\}}_1} \times \cdots \times X_{\int_{i:I} \overline{o\{p_i\}}_n} \qquad \text{(By assumption on } \mathcal{O}_n)$$

$$\cong \coprod_n \coprod_{o \in \mathcal{O}_n(\lim d)} X_{\overline{o}_1} \times \cdots \times X_{\overline{o}_n} \qquad \text{(By Corollary 45)}$$

$$= F(X)_{\lim d}$$

Conversely, let us assume that F restricts to an endofunctor on $\mathscr{C}$. Then, $F(1) = \coprod_n \mathcal{O}_n$ preserves finite connected limits. By Lemma 47, each $\mathcal{O}_n$ preserves finite connected limits. By Corollary 45, it is enough to prove that given a finite connected diagram $d : D \to \mathcal{A}$ and element $o \in \mathcal{O}_n(\lim d)$, the following canonical morphism is an isomorphism

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

Now, we have

$$\int_{i:I} F(X)_{d_i} \cong F(X)_{\lim d} \qquad \text{(By assumption)}$$

$$= \coprod_n \coprod_{o \in \mathcal{O}_n(\lim d)} X_{\overline{o}_1} \times \cdots \times X_{\overline{o}_n}$$

On the other hand,

$$\int_{i:I} F(X)_{d_i} \cong \coprod_n \coprod_{o \in \int_i \mathcal{O}_n(d_i)} X_{\int_{i:I} \overline{p_i(o)}_1} \times \cdots \times X_{\int_{i:I} \overline{p_i(o)}_n} \qquad \text{(By Equation (3.2))}$$

$$= \coprod_n \coprod_{o \in \mathcal{O}_n(\lim d)} X_{\int_{i:I} \overline{o\{p_i\}}_1} \times \cdots \times X_{\int_{i:I} \overline{o\{p_i\}}_n}$$

($\mathcal{O}_n$ preserves finite connected limits)

https://doi.org/10.1017/S0956796825100130 Published online by Cambridge University Press

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. ■

3.3.4 Lemma 37

Cocompleteness follows from Adámek & Rosicky (1994, Remark 1.56), since $\mathscr{C}$ is the category of models of a limit sketch, and is thus locally presentable, by Adámek & Rosicky (1994, Proposition 1.51).

For the claimed closure property, all we have to check is that limits, coproducts, and filtered colimits of functors preserving finite connected limits still preserve finite connected limits. The case of limits is clear, since limits commute with limits. Coproducts and filtered colimits also commute with finite connected limits (Adámek *et al*., 2002, Example 1.3.(vi)).

3.3.5 Corollary 38

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

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 10, which handle 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 $x : \mathbf{\Gamma}'_2 \dashrightarrow \mathbf{\Gamma}'_1$ of renamings (Definition 8). 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

$$\delta[\sigma] = \overline{x}[\sigma'] \tag{4.1}$$

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

Lemma 48. *The right-hand side* $\overline{x}[\sigma']$ *in (4.1) 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 49. *Given*

- $\delta : \mathbf{\Gamma}'_1 \to \Gamma$,
- $x : \mathbf{\Gamma}'_2 \dashrightarrow \mathbf{\Gamma}'_1$,

https://doi.org/10.1017/S0956796825100130 Published online by Cambridge University Press

- $\sigma : \Gamma \to \Delta$,
- $\sigma' : \mathbf{\Gamma'_2} \to \Delta$,

the notation $\Gamma \vdash \delta$ **:>** $x \Rightarrow \sigma'; \sigma \dashv \Delta$ *means that the square* $\begin{array}{ccc} \mathbf{\Gamma'_1} & \xrightarrow{\bar{x}} & \mathbf{\Gamma'_2} \\ {\scriptstyle\delta}\downarrow & & \downarrow{\scriptstyle\sigma'} \\ \Gamma & \xrightarrow[\sigma]{} & \Delta \end{array}$ *is a pushout in* $\mathrm{MCon}(S)_\perp$.

Remark 50. *This justifies the similarity between the pruning notation* $- \vdash - $ **:>** $- \Rightarrow - ; -$ *and the pullback notation of Notation 9, 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 17.(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 $\mathbf{\Gamma}; a \vdash o(\delta)$ for some $o \in \mathcal{O}_n(a)$, where $M \notin \mathbf{\Gamma}$. By Lemma 48, a unifier is given by a substitution $\sigma : \Gamma \to \Delta$ and a term u such that

$$o(\delta[\sigma]) = u\{x\}. \tag{4.2}$$

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 (4.2). 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 (4.2) that $o = o'\{x\}$, and $\delta[\sigma] = \delta'\{x^{o'}\}$.

Note that there is at most one o' such that $o = o'\{x\}$, by Assumption 27.(i). In this case, a unifier is equivalently given by substitutions $\sigma : \Gamma \to \Delta$ and $\sigma' : \alpha_{o'} \to \Delta$ such that $\delta[\sigma] = \sigma'\{x^{o'}\}$. But, by Lemma 48, 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.

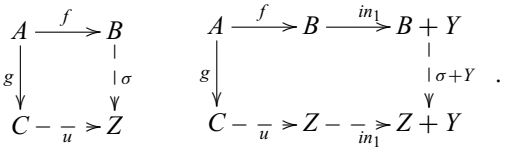
4.2 Flex (rule P-Flex)

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

Thanks to the following lemma, it is actually enough to compute the pushout of $\mathcal{L}x$ and $\mathcal{L}y$, taking $A = (X : a)$, $B = (X : m)$, $C = (X : n)$, $Y = \mathbf{\Gamma} \backslash M$, so that $B + Y \cong \mathbf{\Gamma}$, since coproduct is concatenation of metacontext by Corollary 18.

https://doi.org/10.1017/S0956796825100130 Published online by Cambridge University Press

Lemma 51. *In any category, if the square below left is a pushout, then so is the square below right.*



By Assumption 27.(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 10, which compute coequalisers in $\mathrm{MCon}(S)_\perp$.

The failing rules U-FAIL and U-ID-FAIL are justified by Lemma 17.(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 rule.

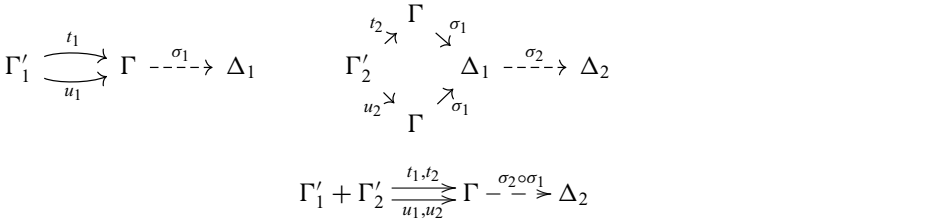
In the next subsections, we discuss the rule sequential rules U-EMPTY and U-SPLIT (Section 5.1), the rule NO-CYCLE transitioning to the pruning phase (Section 5.2), the rule SAME-MVAR unifying metavariable with itself (Section 5.3), and the failing rule CYCLE for cyclic unification of a metavariable 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, since the empty metavariable context is initial: the only morphism to any metacontext Γ is the empty metasubstitution ().

Lemma 52. *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 Rydeheard & Burstall (1988, Theorem 9): if the first two diagrams below are coequalisers, then the last one as well.



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.

Lemma 53. *Consider a commuting square* $\begin{array}{ccc} A & \xrightarrow{u} & B \\ {\scriptstyle v}\downarrow & & \downarrow{\scriptstyle f} \\ C & \xrightarrow[g]{} & D \end{array}$ *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* $in_2 \circ v$.

We take B to be $M : m$ and C to be $\mathbf{\Gamma} \backslash M$. The premise $\mathbf{\Gamma} \backslash M \vdash t :> x \Rightarrow t'; \sigma \dashv \Delta$ tells us that we have a pushout as below left. The conclusion $\mathbf{\Gamma} \vdash M(x) = t \Rightarrow M \mapsto t', \sigma \dashv \Delta$ states that we have a coequaliser as below right, in accordance with the above lemma.

$$\begin{array}{ccc} \ldots & \xrightarrow{\bar{x}} & M : m \\ {\scriptstyle (t)}\downarrow & & \downarrow{\scriptstyle (t')} \\ \mathbf{\Gamma} \backslash M & \xrightarrow[\sigma]{} & \Delta \end{array} \qquad \ldots \underset{in_M \circ \bar{x}}{\overset{t_1}{\rightrightarrows}} \mathbf{\Gamma} \xrightarrow{M \mapsto t', \sigma} \Delta$$

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 Lemma 28, $M(x) = \mathcal{L}x[in_M]$ and $M(y) = \mathcal{L}y[in_M]$. We exploit the following lemma with $u = \mathcal{L}x$, $v = \mathcal{L}y$, $D = \mathbf{\Gamma} \backslash M$ so that $B + D \cong \mathbf{\Gamma}$ since coproduct is concatenation of metacontext, by Corollary 18.

Lemma 54. *In any category, if the below left diagram is a coequaliser, then so is the below right diagram.*

$$A \underset{v}{\overset{u}{\rightrightarrows}} B \xrightarrow{h} C \qquad \begin{array}{ccccc} & u & B & {\scriptstyle in_B} & \\ A & \nearrow & & \searrow & B + D \xrightarrow{h+1_D} C + D \\ & {\scriptstyle v}\searrow & & \nearrow{\scriptstyle in_B} & \\ & & B & & \end{array}$$

It follows that it is enough to compute the coequaliser of $\mathcal{L}x$ and $\mathcal{L}y$. Furthermore, by Assumption 27.(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 59 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

https://doi.org/10.1017/S0956796825100130 Published online by Cambridge University Press

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 55. *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 56. *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 57. *For any term $\mathbf{\Gamma}; a \vdash t$, if $|t|_M = 0$, then $\mathbf{\Gamma} \backslash M; a \vdash t$. Moreover, for any $\mathbf{\Gamma} = (M_1 : m_1, \ldots, M_n : m_n)$, well-formed term t in context $\mathbf{\Gamma}; a$, and successful substitution $\sigma : \mathbf{\Gamma} \to \mathbf{\Delta}$, we have $|t[\sigma]| = |t| + \sum_i |t|_{M_i} \times |\sigma_{M_i}|$.*

Corollary 58. *For any term t in context $\mathbf{\Gamma}; a$ with $(M : m) \in \mathbf{\Gamma}$, successful substitution $\sigma : \mathbf{\Gamma} \to \mathbf{\Delta}$, morphism $x \in \hom_{\mathcal{A}}(m, a)$, we have $|t[\sigma]| \geq |t| + |\sigma_M| \times |t|_M$ and $|M(x)[\sigma]| = |\sigma_M|$.*

Corollary 59. *Let t be a term in context $\mathbf{\Gamma}; a$ with $(M : m) \in \mathbf{\Gamma}$ and $x \in \hom_{\mathcal{A}}(m, a)$ such that $\sigma : \mathbf{\Gamma} \to \mathbf{\Delta}$ unifies t and $M(x)$. Then, either $t = M(y)$ for some $y \in \hom_{\mathcal{A}}(m, a)$, or $\mathbf{\Gamma}; a \vdash t$.*

Proof Since $t[\sigma] = M(x)[u]$, we have $|t[\sigma]| = |M(x)[\sigma]|$. Corollary 58 implies $|\sigma_M| \geq |t| + |\sigma_M| \times |t|_M$. Therefore, either $|t|_M = 0$ and we conclude by Lemma 57, 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 10. Note that the pruning and the unification phases are not mutually recursive: the latter depends on the former, but not conversely. Therefore, we can first show that the pruning phase terminates, and then that the unification does. Both phases involve three recursive calls (cf. the rules P-RIG, P-SPLIT, 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 $|\mathbf{\Gamma}|$ of a proper metacontext $\mathbf{\Gamma}$ as its length, while $|\bot| = 0$ by definition. We also recursively define the size[7] $||t||$ of a proper term t by $||M(x)|| = 1$ and $||o(\vec{t})|| = 1 + ||\vec{t}||$, with $||\vec{t}|| = \sum_i ||t_i||$. We also define the size of the error term $||!||$ as 1. Note that no term is of size 0.

[7] The difference with the notion of size introduced in Definition 55 is that metavariable applications are now of size 1 instead of 0.

https://doi.org/10.1017/S0956796825100130 Published online by Cambridge University Press

Definition 60. *We say that a substitution $\sigma : \Gamma \to \Delta$ is monotone if $||t[\sigma]|| \leq ||t||$ for any term well-formed in the metacontext Γ.*

Example 61. *The error substitution ! is monotone since there is no term of size 0. A substitution $\sigma : \mathbf{\Gamma} \to \mathbf{\Delta}$ such that σ_M is a metavariable application for any $(M : m) \in \mathbf{\Gamma}$ is monotone (as in the output of the rules* P-FLEX *and* SAME-MVAR*).*

Let us first quickly justify termination of the pruning phase.

Lemma 62. *If there is a derivation tree of $\Gamma \vdash \vec{t} :> x \Rightarrow \vec{w}; \sigma \dashv \Delta$, then σ is monotone.*

Corollary 63. *The pruning phase always terminates.*

Proof 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 previous lemma. Let us detail the case of the rule P-SPLIT. We show that the input $\delta[\sigma_1]$ of the second recursive call is smaller than the original input t, δ. Then,

$$\begin{aligned} ||\delta[\sigma_1]|| &\leq ||\delta|| && \text{(By Lemma 62)} \\ &< ||\delta|| + ||t|| && \text{(No term is of size 0)} \\ &= ||t, \delta|| \end{aligned}$$

■

For termination of the main unification phase, we consider the size of the input to be the (lexicographic) pair $(|\Gamma|, ||t||)$ for unify or $(|\Gamma|, ||\vec{t}||)$ for unify-σ, given as input a pair of terms (t, t') or lists of terms $(\vec{t}, \vec{t'})$ in the metacontext Γ.

Lemma 64. *If there is a derivation tree of $\Gamma \vdash \vec{t} :> x \Rightarrow \vec{w}; \sigma \dashv \Delta$ or $\Gamma \vdash \vec{t} = \vec{u} \Rightarrow \sigma \dashv \Delta$, then $|\Gamma| \geq |\Delta|$.*

Lemma 65. *If there is a derivation tree of $\Gamma \vdash \vec{t} = \vec{u} \Rightarrow \sigma \dashv \Delta$ such that $|\Gamma| = |\Delta|$, then σ is monotone.*

Corollary 66. *The unification algorithm as defined in Figure 10 always terminates.*

Proof It is straightforward to check that the sizes of the inputs of recursive calls are strictly smaller thanks to the previous lemmas. Let us detail the case of the rule U-SPLIT. We show that the size $(|\Delta|, ||\delta_1[\sigma]||)$ of the second recursive call is strictly smaller than the size $(|\Gamma|, ||t_1, \delta_1||)$ of the original input.

If $|\Delta| < |\Gamma|$, then we are done. Otherwise, by Lemma 64, we have $|\Delta| = |\Gamma|$, and then,

$$\begin{aligned} ||\delta_1[\sigma]|| &\leq ||\delta_1|| && \text{(By Lemma 65)} \\ &< ||\delta_1|| + ||t_1|| && \text{(No term is of size 0)} \\ &= ||t_1, \delta_1|| \end{aligned}$$

■

https://doi.org/10.1017/S0956796825100130 Published online by Cambridge University Press

6.2 Completeness

In this section, we explain why soundness (Sections 4 and 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 19 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.

We start in Section 7.1 with a variant of pure λ-calculus where metavariable arguments are sets rather than lists. In Section 7.2, we present simply typed λ-calculus, as an example of syntax specified by a multi-sorted binding signature. We then explain in Section 7.3 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 7.4, and finally we present an example of polymorphic language such as System F, in Section 7.5.1.

7.1 Metavariable arguments as sets

If we think of the arguments of a metavariable as specifying the available variables in any order (as in Section 2.1), then it makes sense to assemble them in a set rather than in a list. This motivates considering the category $\mathcal{A} = \mathbb{I}$ whose objects are natural numbers and a morphism $n \to p$ is a subset of $\{1, \ldots, p\}$ of cardinality n. Equivalently, $\mathbb{I}$ can be taken as subcategory of $\mathbb{F}_m$ consisting of strictly increasing injections, or as the subcategory of the augmented simplex category consisting of injective functions. Then, a metavariable takes as argument a set of variables, rather than a list of distinct variables. In this approach, unifying two metavariables (see the rules U-FLEX and P-FLEX) amount to computing a set intersection.

7.2 Simply typed λ-calculus

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

Let T denote the set of simple types generated by a set of base types and a binary arrow type construction $- \Rightarrow -$. Let us now describe the category $\mathcal{A}$ of arities, or scopes, 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 a scope 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 scope $\vec{\sigma}' \to \tau'$ first requires $\tau = \tau'$ and consists of an injective renaming $\vec{r}$ between $\vec{\sigma} = (\sigma_1, \ldots, \sigma_m)$ and $\vec{\sigma}' = (\sigma'_1, \ldots, \sigma'_n)$, that is, a choice of distinct positions $(r_1, \ldots, r_m)$ in $\{1, \ldots, n\}$ such that $\vec{\sigma} = \sigma'_{\vec{r}}$.

https://doi.org/10.1017/S0956796825100130 Published online by Cambridge University Press

Table 1. Simply typed λ-calculus (Section 7.2)

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

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 GB-signature specifying the syntax, where $\lvert\vec{\sigma}\rvert_\tau$ denotes the number (as a cardinal set) of occurrences of τ in $\vec{\sigma}$. The middle column associates a subset of the operation symbols to each typing rule: the total set of operation symbols is recovered by computing the (disjoint) union of them. The last column gives the list $\alpha_o = (\overline{o}_1, \ldots, \overline{o}_n)$ for each operation symbol o.

Proposition 67. *The induced signature is pattern-friendly.*

Proof Because we consider injective renamings, it is easy to check that every morphism in $\mathcal{A}$ is monomorphic. Moreover, the projection functor $\mathcal{A} \to \mathbb{F}_m[T]$ creates finite connected limits. For instance, consider an equaliser diagram in $\mathcal{A}$, that is, two parallel morphisms $\vec{x}, \vec{y}$ between $\vec{\sigma} \to \tau$ and $\vec{\sigma}' \to \tau'$. First, $\tau = \tau'$, and then we compute the equaliser of $\vec{\sigma}$ and $\vec{\sigma}'$ in $\mathbb{F}_m[T]$ following the same process as in pure λ-calculus: we construct the vector $\vec{z}$ of common positions between $\vec{x}$ and $\vec{y}$, thus satisfying $x_{\vec{z}} = y_{\vec{z}}$. Then, $\vec{z}$ defines a morphism from $\sigma_{\vec{z}}$ to $\vec{\sigma}$, which is the equaliser of $\vec{x}$ and $\vec{y}$ in $\mathbb{F}_m[T]$. It is straightforward to check that it is also the equaliser in $\mathcal{A} = \mathbb{F}_m[T] \times T$.

Next, every O_n preserve finite connected limits by Lemma 47 because they are all coproduct of representable functors (as shown below), which preserve limits by Mac Lane (1998, Theorem V.4.1):

$$
\begin{aligned}
O_0(-) &\cong \coprod_{\tau \in T} \hom_{\mathcal{A}}(\tau \to \tau, -) \\
O_1(-) &\cong \coprod_{\tau_1, \tau_2 \in T} \hom_{\mathcal{A}}(\cdot \to \tau_1 \Rightarrow \tau_2, -) \\
O_2(-) &\cong \coprod_{\tau, \tau' \in T} \hom_{\mathcal{A}}(\cdot \to \tau, -) \\
O_{3+n}(-) &\cong \emptyset
\end{aligned}
$$

Note that $\alpha_{3+n,i} : \emptyset \to \mathcal{A}$ vacuously preserves finite connected limits. It remains to show that $\alpha_{1,1}$, $\alpha_{2,1}$, and $\alpha_{2,2}$ also preserve them. First, we observe that the domains of those

https://doi.org/10.1017/S0956796825100130 Published online by Cambridge University Press

functors are isomorphic to $\mathbb{F}_m[T] \times T^2$. More specifically,

$$\int O_i \cong \mathbb{F}_m[T] \times T^2 \qquad i \in \{1,2\}$$
$$(\vec{\sigma} \to \tau_1 \Rightarrow \tau_2, l_{\tau_1,\tau_2}) \mapsto (\vec{\sigma}, \tau_1, \tau_2) \qquad i = 1$$
$$(\vec{\sigma} \to \tau, a_{\tau'}) \mapsto (\vec{\sigma}, \tau, \tau') \qquad i = 2$$

Precomposed with the reverse isomorphisms, the functors $\alpha_{1,1}, \alpha_{2,1}, \alpha_{2,2} : \int O_n \to \mathcal{A}$ induce the three below left functors $\beta_{1,1}$, $\beta_{2,1}$, and $\beta_{2,2}$.

$$\begin{aligned} \mathbb{F}_m[T] \times T^2 &\to \mathbb{F}_m[T] \times T \\ (\vec{\sigma}, \tau_1, \tau_2) &\mapsto ((\vec{\sigma}, \tau_1), \tau_2) \\ (\vec{\sigma}, \tau, \tau') &\mapsto (\vec{\sigma}, (\tau' \Rightarrow \tau)) \\ (\vec{\sigma}, \tau, \tau') &\mapsto (\vec{\sigma}, \tau') \end{aligned} \qquad \begin{aligned} \mathbb{F}_m[T] &\to \mathbb{F}_m[T] \\ \vec{\sigma} &\mapsto (\vec{\sigma}, \tau_1) \\ \vec{\sigma} &\mapsto \vec{\sigma} \\ \vec{\sigma} &\mapsto \vec{\sigma} \end{aligned}$$

Now, finite connected limits in $\mathbb{F}_m[T] \times T^n$ are computed in $\mathbb{F}_m[T]$. Therefore, to conclude that the functors $\beta_{n,i}$ listed above left preserve those limits, it is enough to show that given $\tau_1, \tau_2 \in T$, the compositions $\mathbb{F}_m[T] \xrightarrow{(-,\tau_1,\tau_2)} \mathbb{F}_m[T] \times T^2 \xrightarrow{\beta_{n,i}} \mathbb{F}_m[T] \times T \to \mathbb{F}_m[T]$ listed above right do so. It is obvious for the last two functors, and it is straightforward to check for the first functor, by investigating the constructions of equalisers and pullbacks. ■

It follows that 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 scope $\vec{\sigma} \to \tau$, that is, $\tau' = \tau$ and $\vec{\sigma}' = \sigma_{\vec{z}}$.

7.3 Simply typed λ-calculus modulo βη

In this section, we explain how we account for Miller's original setting: simply typed λ-calculus modulo β and η-equations. We follow Cheney's presentation of the equation-free syntax of β-short η-long normal forms with metavariables (Cheney, 2005, Section 2.1). The normals forms have the following shape, where z denotes a variable, $\vec{x}$ denotes a vector of distinct variables and M denotes a metavariable:

$$t ::= \lambda\vec{y}.z\vec{t} \mid \lambda\vec{y}.M\vec{x}$$

This syntax is closed under substitution of metavariables, modulo β-normalisation. For example, replacing M by $\lambda ww'.z\vec{t}$ in $\lambda\vec{y}.Mxx'$ yields a β-reducible term $\lambda\vec{y}.(\lambda ww'.z\vec{t})xx'$ which yields a normal form after two β-reductions.

Remark 68. *Hereditary substitutions (Watkins et al., 2003; Abel & Pientka, 2011) is a refined implementation of substitution of normal forms that perform β-normalisation on the fly so that the output is a normal form. It works for second-order metavariables,*

https://doi.org/10.1017/S0956796825100130 Published online by Cambridge University Press

where arguments of metavariables are arbitrary terms. The pattern fragment is a simple case: the only β-redices introduced by substituting a metavariable are the application of a λ-abstraction to variables.

We now explain how this syntax of normal forms can be specified by a GB-signature. The associated metavariable substitution is then *hereditary* in the sense of Remark 68, since the syntax generated by any GB-signature is always stable under metavariable substitution.

Notation 69. *We denote a type $\sigma_1 \Rightarrow \cdots \Rightarrow \sigma_n \Rightarrow \iota$ by $\vec{\sigma} \Rightarrow \iota$, where ι is a base type. Note that any type can be written in this way, uniquely.*

We take the same notion of scope as in the previous section: a term well-formed in scope $\vec{\sigma} \to \tau$ will correspond to a normal form of type τ with free variables of types $\vec{\sigma}$.

Definition 70. *We say that a metavariable of arity $\vec{\sigma} \to (\vec{\tau} \Rightarrow \iota)$ has type τ' when $\tau'$is $\vec{\sigma}, \vec{\tau} \Rightarrow \iota$.*

We choose a notion of scope morphism different from that of the previous section, so that the introduction rule of metavariables matches the below typing rule for a metavariable term $\lambda\vec{y}.M\vec{x}$, given a metavariable M of type $\vec{\tau} \Rightarrow \iota$.

$$\frac{\vec{z} : \vec{\sigma}, \vec{y} : \vec{\sigma'} \vdash \overbrace{\vec{x}}^{\text{distinct variables}} : \vec{\tau}}{\vec{z} : \vec{\sigma} \vdash \lambda\vec{y}.M\vec{x} : \vec{\sigma'} \Rightarrow \iota}$$

Assuming that the arity of M is $\vec{\tau}_1 \to \vec{\tau}_2 \Rightarrow \iota$ with $\vec{\tau} = \vec{\tau}_1, \vec{\tau}_2$, this means that a scope morphism from $\vec{\tau}_1 \to (\vec{\tau}_2 \Rightarrow \iota)$ to $\vec{\sigma} \to (\vec{\sigma'} \Rightarrow \iota')$ should be empty unless $\iota = \iota'$. In that case, a morphism is given by a choice of distinct variables of types $\vec{\tau} = \vec{\tau}_1, \vec{\tau}_2$ in $\vec{\sigma}, \vec{\sigma}'$, that is, a morphism in $\mathbb{F}_m[T]$ between those two lists of types. We therefore arrive to the following definition.

Definition 71. *The category of scopes $\mathcal{A}$ has pairs consisting of an object $\vec{\sigma}$ of $\mathbb{F}_m[T]$ and an element τ of T as objects. We denote such a pair by $\vec{\sigma} \to \tau$. A morphism between $\vec{\sigma} \to (\vec{\tau} \Rightarrow \iota)$ and $\vec{\sigma}' \to (\vec{\tau}' \Rightarrow \iota')$ is a morphism between $\vec{\sigma}, \vec{\tau} \to \iota$ and $\vec{\sigma}', \vec{\tau}' \to \iota'$ in $\mathbb{F}_m[T] \times B$. Composition and identities are defined as in $\mathbb{F}_m[T] \times B$.*

This definition readily entails that $\mathcal{A}$ is equivalent to $\mathbb{F}_m[T] \times B$.

We have specified the category of scopes so that we get a suitable introduction rule for metavariables. It remains to provide a GB-signature on top of $\mathcal{A}$ that accounts for the construction $\lambda\vec{y}.x\vec{t}$, with the corresponding below typing rule:

$$\frac{x : (\vec{\tau} \Rightarrow \iota) \in (\vec{z} : \vec{\sigma}, \vec{y} : \vec{\sigma'}) \qquad \vec{z} : \vec{\sigma}, \vec{y} : \vec{\sigma'} \vdash \vec{t} : \vec{\tau}}{\vec{z} : \vec{\sigma} \vdash \lambda\vec{y}.x\vec{t} : \vec{\sigma'} \Rightarrow \iota}$$

https://doi.org/10.1017/S0956796825100130 Published online by Cambridge University Press

The following components induce a GB-signature that generates the same base syntax:

$$O(\vec{\sigma} \to \tau) = \{l_{x,\tau_1,\ldots,\tau_n,\vec{\sigma'},\iota} | \tau = (\vec{\sigma'} \Rightarrow \iota), x \in |\vec{\sigma}, \vec{\sigma'}|_{\vec{\tau} \Rightarrow \iota}\}$$

$$\alpha_{l_{x,\tau_1,\ldots,\tau_n,\vec{\sigma'},\iota}} = \begin{pmatrix} \vec{\sigma}, \vec{\sigma'} \to \tau_1 \\ \ldots \\ \vec{\sigma}, \vec{\sigma'} \to \tau_n \end{pmatrix}$$

7.4 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 scope m, and there is no freedom in choosing the arguments of a metavariable application, since all the variables must be used, in order. Thus, there is no need to even mention those arguments in the syntax. It is thus not surprising that ordered λ-calculus is already handled by first-order unification, where metavariables do not take any argument, by considering ordered λ-calculus as a multi-sorted Lawvere theory where the sorts are the scopes, 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 & Pfenning (2000). 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 & Schürmann, 2010) 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 scope $\vec{\sigma}'|\vec{\omega}' \to \tau'$ 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 $\mathbb{F}_m[T]$ accounts for the unrestricted variables, while T^* accounts for the linear ones: it denotes the discrete category whose objects are lists of elements of T. The remaining components of the GB-signature are specified in Table 2, following the same convention as 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 7.2. For example, the most general unifier of $M(\vec{x})$ and $M(\vec{y})$ maps M to $P(\vec{z})$ where $\vec{z}$ is the vector of common positions of $\vec{x}$ and $\vec{y}$, and P is a fresh metavariable of arity $\sigma_{\vec{z}}|\vec{\omega} \to \tau$.

7.5 Intrinsic polymorphic syntax

7.5.1 Syntactic System F

We present intrinsic System F, in the spirit of Hamana (2011). The Agda implementation of the pattern-friendly GB-signature can be found in the Supplementary Material.

https://doi.org/10.1017/S0956796825100130 Published online by Cambridge University Press

Table 2. Ordered λ-calculus (Section 7.4)

Typing rules	Partition of $O(\vec{\sigma}\|\vec{\omega} \to \tau)$	α_o
$\dfrac{x:\tau \in \Gamma}{\Gamma\|\cdot \vdash x:\tau}$	$\{v_i \| i \in \|\vec{\sigma}\|_\tau \text{ and } \vec{\omega} = ()\}$	$()$
$\dfrac{}{\Gamma\|x:\tau \vdash x:\tau}$	$\{v^> \| \vec{\omega} = ()\}$	$()$
$\dfrac{\Gamma\|\Omega \vdash t:\tau' \Rightarrow \tau \quad \Gamma\|\cdot \vdash u:\tau'}{\Gamma\|\Omega \vdash t\ u:\tau}$	$\{a_{\tau'} \| \tau' \in T\}$	$\begin{pmatrix} \vec{\sigma}\|\vec{\omega} \to (\tau' \Rightarrow \tau) \\ \vec{\sigma}\|() \to \tau' \end{pmatrix}$
$\dfrac{\Gamma\|\Omega_1 \vdash t:\tau' \twoheadrightarrow \tau \quad \Gamma\|\Omega_2 \vdash u:\tau'}{\Gamma\|\Omega_1, \Omega_2 \vdash t^>\ u:\tau}$	$\{a^{\vec{\omega}_1,\vec{\omega}_2}_{\tau'} \| \tau' \in T \text{ and } \vec{\omega} = \vec{\omega}_1, \vec{\omega}_2\}$	$\begin{pmatrix} \vec{\sigma}\|\vec{\omega}_1 \to (\tau' \twoheadrightarrow \tau) \\ \vec{\sigma}\|\vec{\omega}_2 \to \tau' \end{pmatrix}$
$\dfrac{\Gamma, x:\tau_1\|\Omega \vdash t:\tau_2}{\Gamma\|\Omega \vdash \lambda x.t:\tau_1 \Rightarrow \tau_2}$	$\{l_{\tau_1,\tau_2} \| \tau = (\tau_1 \Rightarrow \tau_2)\}$	$(\vec{\sigma}, \tau_1\|\vec{\omega} \to \tau_2)$
$\dfrac{\Gamma\|\Omega, x:\tau_1 \vdash t:\tau_2}{\Gamma\|\Omega \vdash \lambda^> x.t:\tau_1 \twoheadrightarrow \tau_2}$	$\{l^>_{\tau_1,\tau_2} \| \tau = (\tau_1 \twoheadrightarrow \tau_2)\}$	$(\vec{\sigma}, \tau_1\|\vec{\omega} \to \tau_2)$

The syntax of types in type scope n is inductively generated as follows, following the de Bruijn level convention:

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

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, $S_n = \{\tau \mid n \vdash \tau\}$. Intuitively, a metavariable arity $n|\vec{\sigma} \to \tau$ specifies the number n of free type variables, the list of input types $\vec{\sigma}$, and the output type τ, all living in S_n. This provides the underlying set of objects of the category $\mathcal{A}$ of arities. A term t in $n|\vec{\sigma} \to \tau$ considered as a scope 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 scope $n'|\vec{\sigma}' \to \tau'$ must be supplied with a choice $(\eta_1, \ldots, \eta_n)$ of n distinct type variables among the set $\{1, \ldots n'\}$ such that $\tau[\vec{\eta}] = \tau'$, as well as an injective renaming $\vec{\sigma}[\vec{\eta}] \to \vec{\sigma}'$, that is, 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 3, following the same convention 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 Section 7.5.3 the proof that $\mathcal{A}$ has finite connected limits.

Pullbacks and equalisers in $\mathcal{A}$ are essentially computed as in Section 7.2, by computing the vector of common (value) positions. For example, given a metavariable M of arity $m|\vec{\sigma} \to \tau$, to unify $M(\vec{w}|\vec{x})$ with $M(\vec{y}|\vec{z})$, we compute the vector of common positions $\vec{p}$ between $\vec{w}$ and $\vec{y}$, and the vector of common positions $\vec{q}$ between $\vec{x}$ and $\vec{z}$. Then, the

https://doi.org/10.1017/S0956796825100130 Published online by Cambridge University Press

Table 3. The (pattern-friendly) GB-signature of (syntactic) System F (Section 7.5.1)

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

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 scope $m|\vec{\sigma} \to \tau$, that is, m' is the size of $\vec{p}$, while $\tau' = \tau[p_i \mapsto i]$ and $\vec{\sigma}' = \sigma_{\vec{q}}[p_i \mapsto i]$.

7.5.2 System F modulo $\beta\eta$

In this section, we sketch how we can handle System F modulo $\beta\eta$ in the spirit of Section 7.3, by devising a signature for normal forms. To make the syntax more legible, we depart from the previous presentation and instead consider System F as a pure type system. We also ignore the de Bruijn encoding. A scope is now of the shape $\vec{y} : \vec{u} \to \tau$, where

- $\vec{y} : \vec{u}$ is a list of variable declarations $y_1 : u_1, \ldots, y_n : u_n$ where u_i is either $*$, meaning that y_i is a type variable, or a type which is well-formed in context involving all the type variables occuring before y_i in the scope;
- τ is a type well-formed in $\vec{y} : \vec{u}$.

We use the notation $\prod(\alpha : u).\tau$, where τ may depend on α, to mean either $\forall\alpha.\tau_2$ in case $u = *$, or $u \Rightarrow \tau$ otherwise (in the latter case, τ does not depend on α).

Note that any type can be written as $\prod(y_1 : u_1) \prod \cdots \prod(y_n : u_n).\iota$, abbreviated as $\prod(\vec{y} : \vec{u}).\iota$, where ι is a type variable. Any scope $(\vec{y} : \vec{u}) \to \prod(\vec{z} : \vec{v}).\iota$, induces a type $\prod(\vec{y} : \vec{u})(\vec{z} : \vec{v}).\iota$. A morphism between two scopes inducing the types $\prod(\vec{y} : \vec{u}).\iota$ and $\prod(\vec{z} : \vec{v}).\iota'$ is an injective renaming ρ between $\vec{y} : \vec{u}$ and $\vec{z} : \vec{v}$ such that $\iota[\rho] = \iota'$.

Let us now describe the base syntax. We write $\Gamma \vdash t : *$ to mean that t is a type well-formed in Γ. We do not make any syntactic distinction between type and term abstractions.

The base syntax is generated by the following rule, where ι denotes a type variable, and u_i or v_i are either types or $*$.

https://doi.org/10.1017/S0956796825100130 Published online by Cambridge University Press

$$\frac{\Gamma, \vec{y} : \vec{u} \vdash x : \iota}{\Gamma \vdash \lambda \vec{y}.x : \prod(\vec{y} : \vec{u}).\iota} \qquad \frac{\begin{array}{rr} \Gamma, \vec{y} : \vec{u} \vdash x : \prod(\alpha_1 : v_1).\tau_1 & \\ \Gamma, \vec{y} : \vec{u} \vdash t_1 : v_1 & \tau_1[\alpha_1 \mapsto t_1] = \prod(\alpha_2 : v_2).\tau_2 \\ \Gamma, \vec{y} : \vec{u} \vdash t_2 : v_2 & \tau_2[\alpha_2 \mapsto t_2] = \prod(\alpha_3 : v_3).\tau_3 \\ \ldots & \tau_n[\alpha_n \mapsto t_n] = \iota \end{array}}{\Gamma \vdash \lambda \vec{y}.x\vec{t} : \prod(\vec{y} : \vec{u}).\iota}$$

Let us now describe the enriched syntax. We write $M :: \prod(\vec{y} : \vec{u}).\iota$ to mean that the type induced by the arity of M is $\prod(\vec{y} : \vec{u}).\iota$. The introduction rule for metavariables is the following:

$$\frac{M :: \prod(\vec{x} : \vec{t}).\iota \in \Gamma \qquad (\alpha_1, \ldots, \alpha_n) \text{ are distinct variables in } \vec{y}, \vec{z} \text{ of sort } \vec{t}}{\Gamma; \vec{y} : \vec{u} \vdash \lambda \vec{z}.M(\vec{\alpha}) : \prod(\vec{z} : \vec{v}).\iota'}$$

As in Section 7.3, thanks to our modified notion of scope morphism, this rule indeed complies with our introduction rule for metavariables, in the sense that it requires the same data.

7.5.3 Proof that $\mathcal{A}$ has finite connected limits (Section 7.5.1 on System F)

In this section, we show that the category $\mathcal{A}$ of arities for System F (Section 7.5.1) has finite connected limits. First, note that $\mathcal{A}$ is obtained by the *Grothendieck construction* (Jacobs, 1999, Definition 1.10.1) 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 Grothendieck construction of the functor $n \mapsto \mathbb{F}_m[S_n]$.

Lemma 72. *$\mathcal{A}'$ 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 the dual of Jacobs (1999, Proposition 1.10.2.(i)), 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 (Gray, 1966, Corollary 4.3), since each $\mathbb{F}_m[S_n]$ has finite connected limits. ■

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

Lemma 73. *$\mathcal{A}$ 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 44.

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

https://doi.org/10.1017/S0956796825100130 Published online by Cambridge University Press

8 Related work

First-order unification has been explained from a lattice-theoretic point of view by Plotkin (1970) and later categorically analysed by Rydeheard & Burstall (1988, Section 9.7), Goguen (1989), Barr & Wells (1990) as coequalisers. However, there is little work on understanding pattern unification algebraically, with the notable exception of Vezzosi & Abel (2014), working with normalised terms of simply typed λ-calculus. The present paper can be thought of as a generalisation of their work as sketched in their conclusion, although our treatment of their case study differs (Section 7.3).

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

Nominal unification (Urban *et al*., 2003) is an alternative to pattern unification where metavariables are not supplied with the list of allowed variables. Instead, substitution can capture variables. Nominal unification explicitly deals with α-equivalence as an external relation on the syntax, and as a consequence deals with freshness problems in addition to unification problems.

Nominal unification and pattern unification problems are inter-translatable (Cheney, 2005; Levy & Villaret, 2012). 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 (Pientka, 2003; Nanevski et al., 2003; Nanevski *et al*., 2008) 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 7.4).

In the dependently typed setting, Pfenning (1991) provides unification and antiunification algorithms in the pattern fragment for the calculus of constructions. Gundry (2013, Chapter 4) presents a pattern unification algorithm for a dependent type theory as a component of his type inference algorithm, based on the *dynamic* pattern unification algorithm of Abel & Pientka (2011).

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, for example, Hu *et al*. (2022) or Hamana (2004). 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.

https://doi.org/10.1017/S0956796825100130 Published online by Cambridge University Press

Acknowledgements

We are grateful to the anonymous reviewers from JFP for their constructive feedback that greatly helped improving the paper.

Supplementary material

The supplementary material for this article can be found at https://doi.org/10.1017/S0956796825100130

Funding statement

This work was supported in part by a European Research Council (ERC) Consolidator Grant for the project "TypeFoundry," funded under the European Union's Horizon 2020 Framework Programme (grant agreement no. 101002277). This research was supported in part by government funding managed by the French National Research Agency under the France 2030 programme, reference "ANR-22-EXES-0013."

Conflicts of interest

Ambroise Lafont is part of the INRIA team PARTOUT.

References

Abel, A. & Pientka, B. (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. Springer, pp. 10–26.

Adámek, J., Borceux, F., Lack, S. & Rosicky, J. (2002) A classification of accessible categories. *J. Pure Appl. Algebra* **175**(1), 7–30. Special Volume celebrating the 70th birthday of Professor Max Kelly.

Adámek, J. & Rosicky, J. (1994) *Locally Presentable and Accessible Categories*. Cambridge University Press.

Altenkirch, T. & Morris, P. (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, pp. 277–285.

Barr, M. & Wells, C. (1990) *Category Theory for Computing Science*. USA: Prentice-Hall.

Blackwell, R., Kelly, G. & Power, A. (1989) Two-dimensional monad theory. *J. Pure Appl. Algebra* **59**(1), 1–41.

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

De Bruijn, N. G. (1972) Lambda-calculus notation with nameless dummies, a tool for automatic formula manipulation, with application to the church-rosser theorem. *Indagationes Math.* **34**, 381–392.

Dunfield, J. & Krishnaswami, N. R. (2019) Sound and complete bidirectional typechecking for higher-rank polymorphism with existentials and indexed types. *Proc. ACM Program. Lang.* **3**(POPL), 9:1–9:28.

Dybjer, P. (1996) Internal type theory. In *Types for Proofs and Programs*. Berlin, Heidelberg: Springer Berlin Heidelberg, pp. 120–134.

https://doi.org/10.1017/S0956796825100130 Published online by Cambridge University Press

Fiore, M., Plotkin, G. & Turi, D. (1999) Abstract syntax and variable binding. In Proceedings of the 14th Symposium on Logic in Computer Science. IEEE.

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

Gabbay, M. J. & Pitts, A. M. (1999) A new approach to abstract syntax involving binders. In Proceedings of the 14th Symposium on Logic in Computer Science. IEEE.

Goguen, J. A. (1989) What is unification? - A categorical view of substitution, equation and solution. In Resolution of Equations in Algebraic Structures, Volume 1: Algebraic Techniques. Academic, pp. 217–261.

Gray, J. W. (1966) Fibred and cofibred categories. In Proceedings of the Conference on Categorical Algebra. Berlin, Heidelberg: Springer Berlin Heidelberg, pp. 21–83.

Gundry, A. M. (2013) *Type Inference, Haskell and Dependent Types*. PhD Thesis. University of Strathclyde, Glasgow, UK.

Hamana, M. (2004) Free Σ-monoids: A higher-order syntax with metavariables. In Proceedings of the 2nd Asian Symposium on Programming Languages and Systems. Springer, pp. 348–363.

Hamana, M. (2011) Polymorphic abstract syntax via grothendieck construction.

Hu, J. Z. S., Pientka, B. & Schöpp, U. (2022) A category theoretic view of contextual types: From simple types to dependent types. *ACM Trans. Comput. Log.* **23**(4), 25:1–25:36.

Jacobs, B. (1999) *Categorical Logic and Type Theory*, Studies in Logic and the Foundations of Mathematics, vol. 141. Amsterdam: North Holland.

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

Levy, J. & Villaret, M. (2012) Nominal unification from a higher-order perspective. *ACM Trans. Comput. Log.* **13**(2), 10:1–10:31.

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

Miller, D. (1991) A logic programming language with lambda-abstraction, function variables, and simple unification. *J. Log. Comput.* **1**(4), 497–536.

Nanevski, A., Pfenning, F. & Pientka, B. (2008) Contextual modal type theory. *ACM Trans. Comput. Log.* **9**(3), 23:1–23:49.

Nanevski, A., Pientka, B. & Pfenning, F. (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.

Pfenning, F. (1991) Unification and anti-unification in the calculus of constructions. In Proceedings of the Sixth Annual Symposium on Logic in Computer Science (LICS'91), Amsterdam, The Netherlands, July 15–18, 1991. IEEE Computer Society, pp. 74–85.

Pientka, B. (2003) *Tabled Higher-Order Logic Programming*. Carnegie Mellon University.

Plotkin, G. D. (1970) A note on inductive generalization. *Mach. Intell.* **5**, 153–163.

Polakow, J. & Pfenning, F. (2000) Properties of terms in continuation-passing style in an ordered logical framework. In 2nd Workshop on Logical Frameworks and Meta-languages (LFM'00), Santa Barbara, California. Proceedings available as INRIA Technical Report.

Reed, J. (2009) Higher-order constraint simplification in dependent type theory. In Proceedings of the Fourth International Workshop on Logical Frameworks and Meta-Languages: Theory and Practice. New York, NY, USA: Association for Computing Machinery, pp. 49–56.

Reiterman, J. (1977) A left adjoint construction related to free triples. *J. Pure Appl. Algebra* **10**(1), 57–71.

Rydeheard, D. E. & Burstall, R. M. (1988) *Computational Category Theory*, Prentice Hall International Series in Computer Science. Prentice Hall.

Schack-Nielsen, A. & Schürmann, C. (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, pp. 101–116.

Uemura, T. (2021) *Abstract and Concrete Type Theories*. PhD Thesis. University of Amsterdam.

https://doi.org/10.1017/S0956796825100130 Published online by Cambridge University Press

Urban, C., Pitts, A. & Gabbay, M. (2003) Nominal unification. In Computer Science Logic. Berlin, Heidelberg: Springer Berlin Heidelberg, pp. 513–527.
Vezzosi, A. & Abel, A. (2014) A categorical perspective on pattern unification. In RISC-Linz, p. 69.
Watkins, K., Cervesato, I., Pfenning, F. & Walker, D. (2003) *A Concurrent Logical Framework I: Judgments and Properties*. Technical report. Technical Report CMU-CS-02-101, Department of Computer Science, Carnegie . . ..

https://doi.org/10.1017/S0956796825100130 Published online by Cambridge University Press