Formal Metatheory of Second-Order Abstract Syntax

MARCELO FIORE[*] and DMITRIJ SZAMOZVANCEV[†], University of Cambridge, UK

Despite extensive research both on the theoretical and practical fronts, formalising, reasoning about, and implementing languages with variable binding is still a daunting endeavour – repetitive boilerplate and the overly complicated metatheory of capture-avoiding substitution often get in the way of progressing on to the actually interesting properties of a language. Existing developments offer some relief, however at the expense of inconvenient and error-prone term encodings and lack of formal foundations.

We present a mathematically-inspired language-formalisation framework implemented in Agda. The system translates the description of a syntax signature with variable-binding operators into an intrinsically-encoded, inductive data type equipped with syntactic operations such as weakening and substitution, along with their correctness properties. The generated metatheory further incorporates metavariables and their associated operation of metasubstitution, which enables second-order equational/rewriting reasoning. The underlying mathematical foundation of the framework – initial algebra semantics – derives compositional interpretations of languages into their models satisfying the semantic substitution lemma by construction.

CCS Concepts: • **Software and its engineering → Formal software verification**; **Functional languages**; • **Theory of computation → Type theory**; *Equational logic and rewriting.*

Additional Key Words and Phrases: Agda, abstract syntax, language formalisation, category theory

1 INTRODUCTION

Programming and logic research papers that introduce and study new languages and calculi with variable binding typically gloss over the associated notion of capture-avoiding substitution – it is often taken as standard along with its correctness properties. Nevertheless, the representation of variables and substitution becomes a major roadblock when one attempts to formalise such languages in proof assistants, as substituting into terms with variable binders involves dealing with variable shadowing and capture. Some ways of encoding variables (e.g. using strings or numeric de Bruijn indices) lead to fragile and error-prone implementations of syntactic operations: for instance, binding links are easily broken if one increments the wrong index or forgets to modify a string. Other approaches – such as type- and scope-safe de Bruijn indices – place static guarantees on the correctness of the implementation but require significant boilerplate and seemingly superfluous generality that puts the interesting metatheoretic proofs on hold. Either way, the end result is a formalisation littered with ad hoc definitions and lemmas about weakening, substitution, etc., all of which need to be tweaked if the syntax is changed even slightly.

We tackle this unsatisfactory state of affairs by stepping back and considering the mathematical foundations of *second-order abstract syntax*; that is, abstract syntax with variable binding and parametrised metavariables. Using an approach derived from the presheaf model developed by Fiore, Plotkin, and Turi [1999] and Fiore [2008], we build a powerful and entirely generic framework for language formalisation that extracts the maximum amount of syntactic metatheory from a second-order syntax description with minimal boilerplate and user effort. From a concise textual specification of a typed syntax with binding operators, such as the following one for the simply-typed lambda calculus (STLC)

[*]Research partially supported by EPSRC grant EP/V002309/1.
[†]Research supported by EPSRC grant EP/R513180/1.

Authors' address: Marcelo Fiore, marcelo.fiore@cst.cam.ac.uk; Dmitrij Szamozvancev, ds709@cst.cam.ac.uk, Department of Computer Science and Technology, University of Cambridge, UK.

```
type                          term
  N    : 0-ary                  app : α ≻ β    α  ⟶  β
  _≻_ : 2-ary                   lam : α.β         ⟶  α ≻ β
```

with base type N and function type $\succ$, our system generates Agda code for: (*i*) a grammar of types and an intrinsically-typed data type of terms; (*ii*) operations of weakening and substitution together with their correctness properties; (*iii*) a record that, when instantiated with a mathematical model, induces a semantic interpretation of the syntax in the model that preserves substitution; (*iv*) a term constructor for parametrised metavariables and their associated operation of metasubstitution; and (*v*) an equational/rewriting theory that can be instantiated with the axioms of the syntax to obtain a library for second-order equational/rewriting reasoning.

1.1 Background

The framework presented in this paper builds on a long series of practical and theoretical developments in the study of abstract syntax, incorporating elements from both lines of research.

Intrinsic typing. Among the numerous language-formalisation strategies available for a variety of proof assistants, the type- and scope-safe, intrinsically-typed encoding of typed syntax with variable binding popularised by Altenkirch and Reus [1999], Benton et al. [2012], and, more recently, by Allais, Atkey, Chapman, McBride, and McKinna [2021] (henceforth cited as AACMM [2021]) stands out as an excellent fit for dependently-typed metalanguages like Agda [Norell 2009] and Coq [The Coq Development Team 2004]. It represents terms of a syntax as a type- and context-indexed family $\mathcal{X}$ of sets: for a type α and typing context (list of types) Γ, the set $\mathcal{X}\,\alpha\,\Gamma$ consists of the terms t that satisfy the typing judgment $\Gamma \vdash t : \alpha$. In an intrinsically-typed formalisation this indexed family $\mathcal{X}$ is generated inductively: for example, application in the syntax of the STLC corresponds to a constructor app that combines a function term $f : \mathcal{X}\,(\alpha \succ \beta)\,\Gamma$ and an argument term $a : \mathcal{X}\,\alpha\,\Gamma$ into $\mathsf{app}(f, a) : \mathcal{X}\,\beta\,\Gamma$. As such, constructors of the syntax are directly encoded as their own typing rules, making ill-typed and ill-scoped terms unrepresentable.

In contrast, an extrinsic encoding resembles the way type systems are presented in the research literature: one has a grammar of raw terms t and an inductively-defined well-typedness relation $\Gamma \vdash t : \alpha$. While both approaches work and are widely used, dependent types and intrinsic encoding complement each other very well. For instance, in the introductory textbook *Programming Language Foundations in Agda*, Kokke, Siek, and Wadler [2020] present the formalisation of the STLC in both styles, clearly highlighting and advocating the superiority of intrinsic typing.

Another advantage of intrinsic typing is that the need for separate type-preservation proofs of syntactic operations is eliminated: for example, defining the single-variable substitution operation $\mathsf{sub}_1 : \mathcal{X}\;\alpha\;\Gamma \to \mathcal{X}\;\beta\;(\alpha \cdot \Gamma) \to \mathcal{X}\;\beta\;\Gamma$ immediately establishes the fact that substitution preserves typing. Just as the typing rules of terms are baked into their syntax, the well-typedness proofs of operations are baked into their definition. The downside is that implementing such operations often requires significant effort, with sub_1 being a prime example: it is not definable from first principles, because the induction hypothesis (viz. the recursive call) is not general enough to handle binding terms, whose fresh bound variables get added to the end of the typing context and thereby “cover” the free variable one wants to substitute for. Instead, someone unfamiliar with the approach has to walk through a long and frustrating path of trial and error. For instance: trying to define weakening and exchange (which are usually employed in the pen-and-paper well-typedness proof) is similarly futile; generalising to substitution for a variable in the middle of the context works for binding terms but not for variables; further generalising to simultaneous substitution still requires weakening; single-variable weakening cannot be implemented directly either, and must be derived from variable renaming. Even worse is that reasoning about single-variable substitution (proving the syntactic or semantic substitution lemmas, for example) forces one to prove similar properties about renaming, weakening, and simultaneous substitution, leading to a tedious and bloated formalisation that is specific to a particular syntax.

Generic traversals. If we accept the hoops needed to jump through to implement single-variable substitution, we can look for common patterns to abstract out – after all, renaming and substitution both proceed by traversing a term and replacing variables with *data*, either a variable or a term. McBride [2005] builds on this observation to axiomatise the properties that such *data* need possess in order to define a generic term-traversal function that can be instantiated to either renaming or substitution. Allais et al. [2017] generalise traversals to semantic interpretations in models of the syntax, recovering renaming and substitution as interpretations into the syntactic model. They also give a pattern for proving simulation and fusion properties of traversals, which capture – among other things – the correctness properties of renaming and substitution. In AACMM [2021], the authors further generalise their work on the STLC to an arbitrary universe of second-order syntaxes, producing a robust and flexible language-formalisation framework.

Presheaf model. The presheaf model of second-order abstract syntax is a category-theoretic approach to enriching typed languages with variable binding by means of substitution and metasubstitution structures, developed by Fiore and collaborators; see e.g. [Fiore et al. 1999; Fiore 2008; Fiore and Hamana 2013; Fiore and Hur 2010; Fiore and Mahmoud 2010]. It too represents syntax as a type- and context-indexed family of sets, but further requires it to have a functorial renaming structure upon which substitution, metasubstitution, etc. structures are built in a systematic categorical way. While it is an elegant and powerful mathematical theory, its adoption in language-formalisation frameworks has only been incidental: several concepts central to the abstract categorical approach (coends, colimits, etc.) are hard to represent in a dependently-typed setting and a faithful reproduction of the theory would need to heavily rely on a formalised category-theory library such as `agda-categories` [Hu and Carette 2021] or UniMath [Voevodsky et al. 2014].

Recent work by Borthelle et al. [2020] proposed that the presheaf model be adapted to a setting of sorted families by axiomatising the renaming structure and its interaction with initial algebra semantics – ideas which have been explored informally by Allais et al. [2017] and Kaiser et al. [2018] as well. To note, however, is that their skew-monoidal [Szlachányi 2012] approach still depends on categorical theory and results that lead to an impractical formalisation: for example, the syntax of terms is presented by means of the colimit of an ω-chain (and therefore represented by equivalence classes), rather than by an inductive data type.

1.2 Contributions

Our work aims to place the constructions and empirical observations of intrinsically-typed term encodings on a formal grounding, thereby bridging the gap between theory and practice. We are closely guided by the presheaf model of second-order abstract syntax, adapting and re-working its categorical theory (definitions and results) to a dependently-typed setting without sacrificing practicality and generality over arbitrary second-order signatures.

- In Section 2 we give the basic mathematical framework of sorted families, renaming coalgebras, and substitution monoids employed in the library. These notions are adapted from the presheaf model [Fiore et al. 1999] which, on its own, is too abstract and high-level to be used as a basis for practical language formalisation. However, with this new (and mathematically justifiable) change of perspective, we can translate all the theory of the presheaf approach to type- and

context-indexed families and, in doing so, shine a light upon the constructions and properties commonly encountered in type- and scope-safe treatments of syntax.
- Section 3 outlines the main conceptual tool we rely on: initial algebra semantics. Crucially, not only do we invoke initiality to derive generic traversals such as renaming and substitution, we also use it to prove their correctness, including the associativity of substitution and its interaction with renaming. The notion of a $\sum$-monoid is introduced; it axiomatises the structure of a model of the syntax where terms can be interpreted in a compositional and substitution-preserving manner. We also discuss metavariables and metasubstitution [Fiore 2008], together with their application to second-order equational/rewriting reasoning [Fiore and Hur 2010].
- After developing the entirely signature-generic metatheory, in Section 4 we present the translation of second-order syntax descriptions to signature endofunctors and give two approaches of encoding the initial algebra of terms for a signature.
- Finally, in Section 5, we showcase the features of our framework in two extended examples: the syntax of the STLC together with its sound denotational and operational semantics, and the equational theory of partial differentiation.

The paper is intended for an audience familiar with dependently-typed proof assistants and the struggles of language formalisation. Even though they play a vital role in the development, we chose to minimise references to advanced categorical concepts. In the listings we use "retouched Agda" which omits inessential implementation details to make the code cleaner, without impacting clarity (for example, we hide some implicit arguments and record declarations, and use shading instead of cong in equational proofs). The full formalisation with extended proofs and examples can be found on the project website.

2 MATHEMATICAL FOUNDATIONS

We start by setting up the abstract mathematical foundations of our work, with the aim of decoupling the renaming and substitution structures from a particular syntax of terms. Section 2.1 lists the standard definitions for contexts, families, and variables, highlighting their important categorical properties. In Section 2.2 we introduce the coalgebraic and monoidal views of renaming and substitution, and in Section 2.3 we consider properties of maps parametrised by a substitution mapping which will be important players in the development of initial algebra semantics.

2.1 Contexts and families

The definitions of contexts and variables are well-known from intrinsically-typed treatments of syntax. Rather than using named variable-type pairs, contexts are lists of types that come from a fixed set T, and variables are typed and scoped de Bruijn indices into the context: new points to the first element of the context, while old(v) points to the variable v in an extended context.

```
data Ctx : Set where
  ∅   : Ctx
  _·_ : (α : T) → (Γ : Ctx) → Ctx
```

```
data 𝓘 : T → Ctx → Set where
  new :           𝓘 α (α · Γ)
  old  : 𝓘 β Γ → 𝓘 β (α · Γ)
```

We call context-indexed sets of type Ctx → Set *families*, and sort-indexed families (such as $\mathcal{I}$ above) a *sorted family*: for an arbitrary sorted family $\mathcal{X}$: T → Ctx → Set, an element in the set $\mathcal{X}\ \alpha\ \Gamma$ can be seen as an $\mathcal{X}$-term of type α with free variables in context Γ. Maps between sorted families are sort- and context-indexed families of functions, and together they form the category $\mathbf{Fam}_s$.

```
Family : Set₁             Familyₛ : Set₁           _⇾_ : Familyₛ → Familyₛ → Set
Family = Ctx → Set        Familyₛ = T → Family     𝓧 ⇾ 𝓨 = {α : T}{Γ : Ctx} → 𝓧 α Γ → 𝓨 α Γ
```

The concatenation of two contexts is denoted $\Gamma \dotplus \Delta$. An associated endofunctor on families is that of *context extension*: a $(\delta\ \Theta\ \mathcal{X})$-term in context Γ is an $\mathcal{X}$-term in the extended context $\Theta \dotplus \Gamma$.

$$\begin{aligned} &\_\dotplus\_ : \mathsf{Ctx} \longrightarrow \mathsf{Ctx} \longrightarrow \mathsf{Ctx} \\ &\emptyset \qquad\quad \dotplus \Delta = \Delta \\ &(\alpha \cdot \Gamma) \dotplus \Delta = \alpha \cdot (\Gamma \dotplus \Delta) \end{aligned} \qquad \begin{aligned} &\delta : \mathsf{Ctx} \longrightarrow \mathsf{Family}_{\mathsf{S}} \longrightarrow \mathsf{Family}_{\mathsf{S}} \\ &\delta\ \Theta\ \mathcal{X}\ \alpha\ \Gamma = \mathcal{X}\ \alpha\ (\Theta \dotplus \Gamma) \end{aligned}$$

2.1.1 Cocartesian structure of contexts. Fundamental to the algebraic study of abstract syntax is the *category of contexts* $\mathbb{F}$, with contexts Γ, Δ as objects and type-preserving mappings between variables of two contexts as morphisms. We will call these morphisms *renamings*; intuitively, a renaming $\rho : \Gamma \rightsquigarrow \Delta$ assigns a variable $\rho(v) : \alpha$ in context Δ to each variable $v : \alpha$ in Γ. Renamings can be generalised to arbitrary *context maps* $\sigma : \Gamma \ {-[\ \mathcal{X}\ ]\!\rightarrow}\ \Delta$ that assign an $\mathcal{X}$-term $\sigma(v) : \mathcal{X}\ \alpha\ \Delta$ to each variable $v : \alpha$ in Γ – a renaming is then expressed as an $\mathcal{I}$-valued context map.

$$\begin{aligned} &\_{-[\_]\!\rightarrow}\_ : \mathsf{Ctx} \longrightarrow \mathsf{Family}_{\mathsf{S}} \longrightarrow \mathsf{Ctx} \longrightarrow \mathsf{Set} \\ &\Gamma\ {-[\ \mathcal{X}\ ]\!\rightarrow}\ \Delta = \{\alpha : T\} \longrightarrow \mathcal{I}\ \alpha\ \Gamma \longrightarrow \mathcal{X}\ \alpha\ \Delta \end{aligned} \qquad \begin{aligned} &\_\rightsquigarrow\_ : \mathsf{Ctx} \longrightarrow \mathsf{Ctx} \longrightarrow \mathsf{Set} \\ &\Gamma \rightsquigarrow \Delta = \Gamma\ {-[\ \mathcal{I}\ ]\!\rightarrow}\ \Delta \end{aligned}$$

The category of contexts $\mathbb{F}$ is *cocartesian*, with context concatenation $\dotplus$ serving as coproduct. The injection maps $\Gamma \rightsquigarrow (\Gamma \dotplus \Delta)$ and $\Delta \rightsquigarrow (\Gamma \dotplus \Delta)$ and the universal copairing (definable for arbitrary $\mathcal{X}$-valued context maps) $\mathsf{copair} : (\Gamma\ {-[\ \mathcal{X}\ ]\!\rightarrow}\ \Theta) \longrightarrow (\Delta\ {-[\ \mathcal{X}\ ]\!\rightarrow}\ \Theta) \longrightarrow (\Gamma \dotplus \Delta)\ {-[\ \mathcal{X}\ ]\!\rightarrow}\ \Theta$ can be constructed by pattern-matching on the contexts and variables, and repeated applications of old. A useful special case of copairing is adding a single term to a substitution, which corresponds to the standard *cons* operation in the theory of explicit substitutions [Abadi et al. 1989].

$$\begin{aligned} &\mathsf{add} : (\mathcal{X} : \mathsf{Family}_{\mathsf{S}}) \longrightarrow \mathcal{X}\ \alpha\ \Delta \longrightarrow \Gamma\ {-[\ \mathcal{X}\ ]\!\rightarrow}\ \Delta \longrightarrow (\alpha \cdot \Gamma)\ {-[\ \mathcal{X}\ ]\!\rightarrow}\ \Delta \\ &\mathsf{add}\ \mathcal{X}\ t\ \sigma = \mathsf{copair}\ \mathcal{X}\ (\lambda\{\ \mathsf{new} \rightarrow t\ \})\ \sigma \end{aligned}$$

2.1.2 Bicartesian closed and linear structures of families. Since families are indexed sets, they inherit the bicartesian closed structure of **Set**: we take sums and products of families via pointwise disjoint unions and Cartesian products, and also define family exponentials pointwise as $(\mathcal{X} \Rightarrow \mathcal{Y})\ \alpha\ \Gamma \triangleq \mathcal{X}\ \alpha\ \Gamma \longrightarrow \mathcal{Y}\ \alpha\ \Gamma$ (since there are no functoriality requirements, relevant in the case of presheaf exponentials, to be satisfied). We also have the following *linear exponential* (derived from the Day [1970] internal hom) which will play an important role in metasubstitution:

$$\begin{aligned} &\_\multimap\_ : \mathsf{Family}_{\mathsf{S}} \longrightarrow \mathsf{Family}_{\mathsf{S}} \longrightarrow \mathsf{Family}_{\mathsf{S}} \\ &(\mathcal{X} \multimap \mathcal{Y})\ \alpha\ \Gamma = \{\Delta : \mathsf{Ctx}\} \longrightarrow \mathcal{X}\ \alpha\ \Delta \longrightarrow \mathcal{Y}\ \alpha\ (\Delta \dotplus \Gamma) \end{aligned}$$

2.2 Renaming and substitution

Our next step is to precisely identify the structure required on a sorted family to support substitution; this way, defining substitution for a particular syntax will amount to equipping the family of terms of the syntax with such structure. Since, in practice, capture-avoiding substitution into syntactic terms involves weakening (a form of variable renaming), we are also looking for a way to axiomatise renaming as a basic fundamental notion. Our guiding principle throughout the development will be to characterise renaming, substitution, and other metatheoretic operations – along with their correctness laws – as natural, recognisable categorical constructions in sorted families.

2.2.1 Renaming structure. A sorted family $\mathcal{X}$ admits renaming when $\mathcal{X}$-terms in one variable context can be compatibly transformed to $\mathcal{X}$-terms in another. The corresponding renaming operation lifts a renaming map $\Gamma \rightsquigarrow \Delta$ to a function $\mathcal{X}\ \alpha\ \Gamma \longrightarrow \mathcal{X}\ \alpha\ \Delta$. This amounts to the requirement that the family $\mathcal{X}\ \alpha$ be a (covariant) *presheaf* on $\mathbb{F}$ (i.e. a functor from $\mathbb{F}$ to **Set**), and imposing such structure on every family would give rise to the presheaf model of Fiore et al. [1999]. Here, instead of going down this route, we refactor the type of the renaming operation on families as follows:

```
r : {α : T}{Γ : Ctx} → 𝒳 α Γ → ({Δ : Ctx} → (Γ ⇝ Δ) → 𝒳 α Δ)
```

Note that the codomain of r can be expressed as a function of α and Γ. By abstracting the renaming-dependence as a *modal operator* $\Box$ [Allais et al. 2021], we may then express the renaming operation as a map of families $\mathcal{X} \rightarrowtriangle \Box\,\mathcal{X}$, internalising the functorial action as a $\mathbf{Fam}_{\mathsf{S}}$-morphism.

```
□ : Familyₛ → Familyₛ
□ 𝒳 α Γ = {Δ : Ctx} → (Γ ⇝ Δ) → 𝒳 α Δ
```

The $\Box$ modality is a comonad and a coalgebra for it has to respect the identity and composition of renamings. This leads to our first observation: a sorted family has renaming structure when equipped with a $\Box$-*coalgebra* structure. The Coalg record collects these requirements.

```
record Coalg (𝒳 : Familyₛ) : Set where
  field r       : 𝒳 ⇾ □ 𝒳
        counit  : {t : 𝒳 α Γ} → r t id ≡ t
        comult  : {ρ : Γ ⇝ Δ}{ϱ : Δ ⇝ Θ}{t : 𝒳 α Γ} → r t (ϱ ∘ ρ) ≡ r (r t ρ) ϱ
```

Since context transformations such as weakening, contraction, etc. correspond to renaming maps, the associated structural rules for a $\Box$-coalgebra $\mathcal{X}$ can be derived via renaming:

```
wkl : 𝒳 α Γ → 𝒳 α (Γ ∔ Δ)      wkr : 𝒳 α Δ → 𝒳 α (Γ ∔ Δ)      contr : 𝒳 α (Γ ∔ Γ) → 𝒳 α Γ
wkl t = r t (inl Δ)             wkr t = r t (inr Γ)             contr t = r t (copair ℐ id id)
```

The natural notion of a transformation between $\Box$-coalgebras is a *homomorphism*: a map $\mathcal{X} \rightarrowtriangle \mathcal{Y}$ that preserves the coalgebra structures of $\mathcal{X}$ and $\mathcal{Y}$.

```
record Coalg⇒ (𝒳ᵒ : Coalg 𝒳)(𝒴ᵒ : Coalg 𝒴) (f : 𝒳 ⇾ 𝒴) : Set where
  field ⟨r⟩ : {ρ : Γ ⇝ Δ}{t : 𝒳 α Γ} → f (𝒳.r t ρ) ≡ 𝒴.r (f t) ρ
```

2.2.2 *Pointed structure.* If $\mathcal{X}$ is a sorted family of terms for a second-order abstract syntax, there must be a way to coerce variables into $\mathcal{X}$-terms. Sorted families with such a coercion $\eta : \mathcal{I} \rightarrowtriangle \mathcal{X}$ will be called *pointed*. If the underlying family of a $\Box$-coalgebra is pointed, we may also impose the requirement that the point is compatible with renaming. We characterise pointed coalgebras and their point-preserving homomorphisms as records:

```
record Coalg* (𝒳 : Familyₛ) : Set where      record Coalg*⇒ (𝒳*ᵒ : Coalg* 𝒳) (𝒴*ᵒ : Coalg* 𝒴)
  field ᵒ : Coalg 𝒳   ;   η : ℐ ⇾ 𝒳                          (f : 𝒳 ⇾ 𝒴) : Set where
        r∘η : {v : ℐ α Γ}{ρ : Γ ⇝ Δ} →           field ᵒ⇒ : Coalg⇒ 𝒳.ᵒ 𝒴.ᵒ f
            r (η v) ρ ≡ η (ρ v)                         ⟨η⟩ : {v : ℐ α Γ} → f (𝒳.η v) ≡ 𝒴.η v
```

An example of a pointed $\Box$-coalgebra is the family of variables, with renaming implemented as application. The point η of a pointed $\Box$-coalgebra is itself a pointed $\Box$-coalgebra homomorphism.

2.2.3 *Substitution structure.* The $\Box$ modality parametrises a family by a renaming. We now generalise this by parametrising a family $\mathcal{Y}$ by an arbitrary $\mathcal{X}$-valued context map. This is captured as a binary operation on families which we call the *internal substitution hom* of $\mathcal{X}$ and $\mathcal{Y}$:

```
⦅_,_⦆ : Familyₛ → Familyₛ → Familyₛ
⦅ 𝒳 , 𝒴 ⦆ α Γ = {Δ : Ctx} → (Γ -[ 𝒳 ]→ Δ) → 𝒴 α Δ
```

A renaming operation has, equivalently, the type $\mathcal{X} \rightarrowtriangle ⦅\, \mathcal{I}, \mathcal{X}\, ⦆$, and a substitution operation on $\mathcal{X}$ has the type $\mathcal{X} \rightarrowtriangle ⦅\, \mathcal{X}, \mathcal{X}\, ⦆$. The internal hom has a left adjoint $\odot$, called the *substitution tensor product*; that is, maps of the form $\mathcal{X} \rightarrowtriangle ⦅\, \mathcal{Y}, \mathcal{Z}\, ⦆$ are naturally isomorphic to maps of the form $\mathcal{X} \odot \mathcal{Y} \rightarrowtriangle \mathcal{Z}$ via “uncurrying”. These operators equip $\mathbf{Fam}_{\mathsf{S}}$ with a skew-monoidal closed structure

[Street 2013] (full monoidality would require that e.g. the unitor $\mathcal{I} \odot \mathcal{X} \dot\rightarrow \mathcal{X}$ is invertible, which is not the case since $\odot$ is defined as a dependent sum, rather than a coend [Fiore et al. 1999]).

```
_⊙_ : Familyₛ → Familyₛ → Familyₛ
(𝒳 ⊙ 𝒴) α Δ = Σ[ Γ ∈ Ctx ] (𝒳 α Γ × (Γ ~[ 𝒴 ]↝ Δ))
```

Expressed using the tensor product, the substitution operation has the type $\mathcal{X} \odot \mathcal{X} \dot\rightarrow \mathcal{X}$ – it combines a term $\mathcal{X}\ \alpha\ \Gamma$ and substitution map Γ ~[$\mathcal{X}$]↝ Δ into a term $\mathcal{X}\ \alpha\ \Delta$. To be proper, it must also be associative, and be compatible with the point of $\mathcal{X}$ when it has one. If we package this structure on a family $\mathcal{X}$ into a record – a point and substitution operation $\mathcal{I} \xrightarrow{\eta} \mathcal{X} \xleftarrow{\mu} \mathcal{X} \odot \mathcal{X}$ satisfying unit and associativity laws – we end up with precisely a *monoid* in $(\mathbf{Fam}_{\mathsf{S}}, \mathcal{I}, \odot)$. Monoids can be equivalently expressed using the internal hom as $\mathcal{I} \xrightarrow{\eta} \mathcal{X} \xrightarrow{\mu} ⟦\mathcal{X}, \mathcal{X}⟧$, which is the presentation we prefer for technical reasons: the metatheory proofs would not go through if we had used $\odot$.

```
record Mon (ℳ : Familyₛ) : Set where
  field η : ℐ  ⇾ ℳ
        μ : ℳ ⇾ 〖 ℳ , ℳ 〗

        lunit  : {σ : Γ ~[ ℳ ]↝ Δ}{v : ℐ α Γ} → μ (η v) σ ≡ σ v
        runit  : {t : ℳ α Γ} → μ t η ≡ t
        assoc : {σ : Γ ~[ ℳ ]↝ Δ} {ς : Δ ~[ ℳ ]↝ Θ} {t : ℳ α Γ} →
                   μ (μ t σ) ς ≡ μ t (λ v → μ (σ v) ς)
```

The multiplication μ represents simultaneous substitution, replacing every variable in context Γ with an $\mathcal{M}$-term in Δ. In practice (e.g. in β-reduction) one often uses one- or two-variable substitution for the last variable or variables in the context, which is derived using add:

```
[_/] : ℳ α Γ → ℳ β (α ∙ Γ) → ℳ β Γ     [_,_/]₂ : ℳ α Γ → ℳ β Γ → ℳ τ (α ∙ β ∙ Γ) → ℳ τ Γ
[ s /] t = μ t (add ℳ s η)             [ s₁ , s₂ /]₂ t = μ t (add ℳ s₁ (add ℳ s₂ η))
```

Monoid homomorphisms preserve the unit and multiplication. When $\mathcal{M}$ is a family associated with an inductively defined syntax, $\mathcal{N}$ is a model of the syntax, and f is a map $\mathcal{M} \dot\rightarrow \mathcal{N}$, the preservation of multiplication ⟨μ⟩ : {σ : Γ ~[ℳ]↝ Δ}{t : ℳ α Γ} → f (ℳ.μ t σ) ≡ 𝒩.μ (f t) (f ∘ σ) expresses the *semantic substitution lemma*: the interpretation of substitution in the syntax is the substitution of interpretations in the model. We give its familiar form for single-variable substitution as an example below. The fact that f commutes with add is established by function extensionality, case analysis on the variable, and preservation of the unit ⟨η⟩ : {v : ℐ α Γ} → f (ℳ.η v) ≡ 𝒩.η v.

```
sub-lemma : (s : ℳ α Γ)(t : ℳ β (α ∙ Γ)) → f (ℳ.[ s /] t) ≡ 𝒩.[ f s /] (f t)
sub-lemma s t = trans ⟨μ⟩ (cong (𝒩.μ (f t)) (ext λ{ new → refl ; (old y) → ⟨η⟩}))
```

Every monoid has an induced pointed $\Box$-coalgebra instance $\mathcal{M}.^{\Box}_{*}$: Coalg$_*$, where renaming is implemented by substituting variables for variables, and pointed $\Box$-coalgebra laws follow from monoid axioms. But, what if $\mathcal{M}$ already comes with a coalgebra structure? We will revisit this question after examining some general properties of maps into internal homs.

2.3 Parametrised maps

Working with families with coalgebra structure lets us be precise about when renaming is required in constructions: for example, we need the coalgebra structure for weakening on $\mathcal{X}$, but not for copairing of $\mathcal{X}$-valued context maps. However, as Fiore et al. [1999] show, the interaction between the substitution tensor and the presheaf structure forms a core part of the model theory that will need to be recast in our coalgebraic setting. To achieve this, we introduce an important collection of properties for maps into internal homs (equivalently, out of substitution tensors).

2.3.1 Coalgebraic maps. Maps of the form $f : \mathcal{X} \multimap \llparenthesis\, \mathcal{P} , \mathcal{Y} \,\rrparenthesis$ play a vital role in the mathematical theory, since they transform $\mathcal{X}$ to $\mathcal{Y}$ while altering the variable context according to a parameter family $\mathcal{P}$. Many syntactic operations are of this form: for example, as we have already mentioned, renaming $\mathcal{X} \multimap \llparenthesis\, \mathcal{I} , \mathcal{X} \,\rrparenthesis$ and substitution $\mathcal{X} \multimap \llparenthesis\, \mathcal{X} , \mathcal{X} \,\rrparenthesis$ are maps parametrised by $\mathcal{I}$ and $\mathcal{X}$, respectively. If all three families $\mathcal{X}, \mathcal{P}, \mathcal{Y}$ have pointed coalgebra structure, there are two ways in which a map f can be compatible with the various renaming operations: it can preserve renaming, or identify terms that get renamed via the coalgebra structure and via the context map. Furthermore, one can state a compatibility law between the points of all three families. Maps which satisfy these three properties are called *coalgebraic*.

```
record Coalgebraic (f : 𝒳 ⇾ ⦇ 𝒫 , 𝒴 ⦈) : Set where
  field  r∘f : {σ : Γ -[ 𝒫 ]→ Δ}{ϱ : Δ ⇝ Θ} {t : 𝒳 α Γ} → 𝒴.r (f t σ) ϱ  ≡ f t (λ v → 𝒫.r (σ v) ϱ)
         f∘r : {ρ : Γ ⇝ Δ}{ς : Δ -[ 𝒫 ]→ Θ} {t : 𝒳 α Γ} → f (𝒳.r t ρ) ς  ≡ f t (ς ∘ ρ)
         f∘η :                             {v : ℐ α Γ} → f (𝒳.η v) 𝒫.η ≡ 𝒴.η v
```

Though the hom of pointed families is not in general pointed, the codomain of coalgebraic maps has a pointed □-coalgebra structure, and the map itself is a pointed □-coalgebra homomorphism.

```
Cod□* : Coalg* ⦇ 𝒫 , 𝒴 ⦈
Cod□* = record { η = λ v σ → f (𝒳.η v) σ
               ; □ = record { r = λ h ρ σ → h (σ ∘ ρ) ; counit = refl ; comult = refl }
               ; r∘η = λ {v}{ρ} → ext λ σ → begin f (𝒳.η v) (σ ∘ ρ)     ≡˘⟨ f∘r ⟩
                                                  f (𝒳.r (𝒳.η v) ρ) σ ≡⟨ 𝒳.r∘η ⟩
f□*⇒ : Coalg*⇒ 𝒳□* Cod□* f                       f (𝒳.η (ρ v)) σ       ∎ }
f□*⇒ = record { □⇒ = record { ⟨r⟩ = ext (λ σ → f∘r) } ; ⟨η⟩ = refl }
```

Examples of coalgebraic maps are the renaming map for a pointed □-coalgebra:

```
rᶜ : (𝒳□* : Coalg* 𝒳) → Coalgebraic 𝒳□* ℐ□* 𝒳□* 𝒳.r
rᶜ 𝒳□* = record { r∘f = sym comult ; f∘r = sym comult ; f∘η = r∘η }
```

as well as the multiplication μ for a substitution monoid and the structural map $\mathsf{j} : \mathcal{I} \multimap \llparenthesis\, \mathcal{X} , \mathcal{X} \,\rrparenthesis$ of skew-closed categories that corresponds to application (mapping v to $\sigma \mapsto \sigma(v)$). Later on, we will be able to show that universal parametrised maps from an initial syntactic algebra are coalgebraic, which will be a prerequisite for proving the substitution axioms.

2.3.2 Lifting and strength. The definition of simultaneous substitution by hand involves traversing the host term, applying the substitution map to variables, and recursing into subterms. The difficulties arise when the subterm has a newly bound variable, such as in the body of a λ-abstraction. To avoid variable shadowing, substitution must map the newly bound variable to itself; to avoid variable capture, the terms to be substituted must not contain free occurrences of the new variable. Fortunately, intrinsically-typed encoding guides the complex de Bruijn "arithmetic" required and guarantees to maintain type- and scope-safety.

The critical step is applying a substitution $\sigma : \Gamma$ -[$\mathcal{P}$]→ Δ to a term in an extended context $\mathcal{P}\ \beta\ (\alpha \cdot \Gamma)$, without disturbing the newly bound variable α. The usual name for the transformation needed on σ is *lifting* [Altenkirch and Reus 1999; Benton et al. 2012], which can be generalised to arbitrary extensions as lift $\Theta\ \sigma : (\Theta \dotplus \Gamma)$ -[$\mathcal{P}$]→ $(\Theta \dotplus \Delta)$. The inductive definition of lifting requires both a point (to map newly bound variables to themselves) and renaming (to weaken the context of the recursive call), so we demand the structure of a pointed $\Box$-coalgebra on $\mathcal{P}$:

$$
\begin{aligned}
&\mathsf{lift} : (\Theta : \mathsf{Ctx}) \longrightarrow (\Gamma \mathrel{-[\ \mathcal{P}\ ]\!\!\to} \Delta) \longrightarrow (\Theta \mathbin{\dot{+}} \Gamma) \mathrel{-[\ \mathcal{P}\ ]\!\!\to} (\Theta \mathbin{\dot{+}} \Delta)\\
&\mathsf{lift}\ \emptyset \qquad\quad\ \sigma\ v \qquad\ \ = \sigma\ v\\
&\mathsf{lift}\ (\tau \cdot \Theta)\ \sigma\ \mathsf{new} \quad\ = \mathcal{P}.\eta\ \mathsf{new}\\
&\mathsf{lift}\ (\tau \cdot \Theta)\ \sigma\ (\mathsf{old}\ v) = \mathcal{P}.\mathsf{r}\ (\mathsf{lift}\ \Theta\ \sigma\ v)\ \mathsf{old}
\end{aligned}
$$

While this definition works, it expresses lift as a Set-level transformation of context maps, rather than as a morphism of sorted families – as such, it counterposes our goal of representing metasyntactic operations purely algebraically. Consequently, it is not a priori obvious what laws lift should satisfy. Benton et al. [2012, Section 4] demonstrate the intricate inter-dependence of lift with the other structures by listing eight laws that concern the interaction amongst lifting, renaming, and substitution, all of which must be proved in order, with each property building on top of the previous ones. It is these sorts of auxiliary definitions and ad-hoc lemmas that we wish to avoid in our systematic, categorical development of abstract syntax.

The key to demystifying lift lies in the categorical concept of *cotensorial strength* [Kock 1971] for a sorted-family endofunctor F: namely, an operation of type $F⦇\mathcal{X}, \mathcal{Y}⦈ \dashrightarrow ⦇\mathcal{X}, F\mathcal{Y}⦈$ satisfying certain coherence laws. The analogous tensorial strength $F(\mathcal{X}) \odot \mathcal{Y} \dashrightarrow F(\mathcal{X} \odot \mathcal{Y})$ plays a central role in the presheaf model of abstract syntax [Fiore 2008], since it captures the intuition of pushing a substitution into a syntactic structure represented by the endofunctor F. Most notably, lift attains a natural place within the categorical model as it can be used to implement strength for the instance $F = \delta\Theta$: the resulting operation $\delta\Theta⦇\mathcal{P}, \mathcal{X}⦈ \dashrightarrow ⦇\mathcal{P}, \delta\Theta\mathcal{X}⦈$ is then responsible for pushing a substitution $\sigma\colon\ \Gamma \mathrel{-[\ \mathcal{P}\ ]\!\!\to} \Delta$ under a binder of variables in context Θ.

In our setting of families and coalgebras, a more refined notion of strength is required (similar to the structural strength of Borthelle et al. [2020]). A *coalgebraic strength* for F is a transformation

$$\mathsf{str} : F⦇\mathcal{P}, \mathcal{X}⦈ \dashrightarrow ⦇\mathcal{P}, F\mathcal{X}⦈$$

where $\mathcal{P}$ is a pointed $\Box$-coalgebra and $\mathcal{X}$ is a sorted family. The operation must be *natural* in both components: that is, it must suitably commute with the functorial mapping of any pointed $\Box$-coalgebra homomorphism $f\colon \mathcal{Q} \dashrightarrow \mathcal{P}$ and family of maps $g\colon \mathcal{X} \dashrightarrow \mathcal{Y}$.

$$
\begin{aligned}
&\mathsf{str\text{-}nat}_1 : (f^{\Box}_{*}\Rightarrow : \mathsf{Coalg}_{*}\!\Rightarrow \mathcal{Q}^{\Box}_{*}\ \mathcal{P}^{\Box}_{*}\ f)\ (h : F⦇\mathcal{P}, \mathcal{X}⦈\ \alpha\ \Gamma)\ (\sigma : \Gamma \mathrel{-[\ \mathcal{Q}\ ]\!\!\to} \Delta) \longrightarrow\\
&\quad \mathsf{str}\ \mathcal{P}^{\Box}_{*}\ \mathcal{X}\ h\ (f \circ \sigma) \equiv \mathsf{str}\ \mathcal{Q}^{\Box}_{*}\ \mathcal{X}\ (F_1\ (\lambda\ h'\ \sigma' \to h'\ (f \circ \sigma')\ )\ h)\ \sigma\\
&\mathsf{str\text{-}nat}_2 : (g : \mathcal{X} \dashrightarrow \mathcal{Y})(h : F⦇\mathcal{P}, \mathcal{X}⦈\ \alpha\ \Gamma)(\sigma : \Gamma \mathrel{-[\ \mathcal{P}\ ]\!\!\to} \Delta) \longrightarrow\\
&\quad \mathsf{str}\ \mathcal{P}^{\Box}_{*}\ \mathcal{Y}\ (F_1\ (\lambda\ h'\ \sigma' \to g\ (h'\ \sigma'))\ h)\ \sigma \equiv F_1\ g\ (\mathsf{str}\ \mathcal{P}^{\Box}_{*}\ \mathcal{X}\ h\ \sigma)
\end{aligned}
$$

The strength also satisfies a unit law str-unit: $(h : F⦇\mathcal{I}, \mathcal{X}⦈\ \alpha\ \Gamma) \longrightarrow \mathsf{str}\ \mathcal{I}^{\Box}_{*}\ \mathcal{X}\ h\ \mathsf{id} \equiv F_1\ (\mathsf{i}\ \mathcal{X})\ h$ and an associativity law with respect to the two other structural transformations of skew-closed categories: the unit $\mathsf{i} = \lambda\ h \to h\ \mathsf{id} : ⦇\mathcal{I}, \mathcal{X}⦈ \dashrightarrow \mathcal{X}$ and the (curried) composition of internal homs $\mathsf{L} = \lambda\ h\ \sigma\ \varsigma \to h\ (\lambda\ v \to \sigma\ v\ \varsigma) : ⦇\mathcal{Y}, \mathcal{Z}⦈ \dashrightarrow ⦇⦇\mathcal{X}, \mathcal{Y}⦈, ⦇\mathcal{X}, \mathcal{Z}⦈⦈$. The associativity law has to be stated in terms of coalgebraic maps $f : \mathcal{P} \dashrightarrow ⦇\mathcal{Q}, \mathcal{R}⦈$, since the usual pentagon identity [Kock 1971, Lemma 1.3] is too strict in the skew-closed setting (see str-assoc in the diagram below). This generalised associativity law neatly combines with naturality to give the following powerful corollary, given both as Agda code and – for a clearer presentation – as a commutative diagram:

$$\begin{aligned}
&\mathsf{str\text{-}dist} : \{f : \mathcal{P} \twoheadrightarrow \llbracket\, Q \,,\, \mathcal{R} \,\rrbracket\}\,(f^{\mathrm{c}} : \mathsf{Coalgebraic}\ \mathcal{P}^{\square}_{*}\ Q^{\square}_{*}\ \mathcal{R}^{\square}_{*}\ f) \\
&\qquad (h : F\ \llbracket\, \mathcal{R} \,,\, \mathcal{X} \,\rrbracket\ \alpha\ \Gamma)(\sigma : \Gamma\ \text{-[}\ \mathcal{P}\ \text{]}\!\!\to \Delta)(\varsigma : \Delta\ \text{-[}\ Q\ \text{]}\!\!\to \Theta) \to \\
&\quad \mathsf{str}\ \mathcal{R}^{\square}_{*}\ \mathcal{X}\ h\ (\lambda\, v \to f\ (\sigma\ v)\ \varsigma) \\
&\equiv \mathsf{str}\ Q^{\square}_{*}\ \mathcal{X}\ (\mathsf{str}\ \mathcal{P}^{\square}_{*}\ \llbracket\, Q \,,\, \mathcal{X} \,\rrbracket\ (F_1\ (\lambda\, h\ \sigma\ \varsigma \to h\ (\lambda\, v \to f\ (\sigma\ v)\ \varsigma))\ h)\ \sigma)\ \varsigma
\end{aligned}$$

$$\begin{tikzcd}
F\llbracket \mathcal{R},\mathcal{X} \rrbracket \arrow[r, "F\mathsf{L}"] \arrow[dd, "\mathsf{str}_{\mathcal{R},\mathcal{X}}"'] & F\llbracket \llbracket Q,\mathcal{R} \rrbracket, \llbracket Q,\mathcal{X} \rrbracket \rrbracket \arrow[r, "F\llbracket f\text{,}\mathrm{id} \rrbracket"] \arrow[d, "\mathsf{str}_{\llbracket Q,\mathcal{R}\rrbracket,\llbracket Q,\mathcal{X}\rrbracket}"'] \arrow[rd, phantom, "\mathsf{str\text{-}nat}_1"] & F\llbracket \mathcal{P}, \llbracket Q,\mathcal{X} \rrbracket \rrbracket \arrow[d, "\mathsf{str}_{\mathcal{P},\llbracket Q,\mathcal{X}\rrbracket}"] \\
\arrow[r, phantom, "\mathsf{str\text{-}assoc}"] & \llbracket \llbracket Q,\mathcal{R} \rrbracket, F\llbracket Q,\mathcal{X} \rrbracket \rrbracket \arrow[r, "\llbracket f\text{,}\mathrm{id} \rrbracket"'] & \llbracket \mathcal{P}, F\llbracket Q,\mathcal{X} \rrbracket \rrbracket \arrow[d, "\llbracket \mathrm{id}\text{,}\mathsf{str}_{Q,\mathcal{X}} \rrbracket"] \\
\llbracket \mathcal{R}, F\mathcal{X} \rrbracket \arrow[r, "\mathsf{L}"'] & \llbracket \llbracket Q,\mathcal{R} \rrbracket, \llbracket Q,F\mathcal{X} \rrbracket \rrbracket \arrow[r, "\llbracket f\text{,}\mathrm{id} \rrbracket"'] & \llbracket \mathcal{P}, \llbracket Q,F\mathcal{X} \rrbracket \rrbracket
\end{tikzcd}$$

As f is coalgebraic, its codomain $\llbracket\, Q \,,\, \mathcal{R} \,\rrbracket$ is a pointed $\square$-coalgebra (Section 2.3.1) and thus is a valid first component to $\mathsf{str}_{\llbracket Q,\mathcal{R}\rrbracket,\llbracket Q,\mathcal{X}\rrbracket}$. Remarkably, for different choices of f, the str-dist corollary above generalises all four lifting lemmas given by Benton et al., fulfilling our goal of placing lift and its laws on a formal foundation.

The axiomatisation of strength as a categorical concept is required for the initiality and freeness theorems of Section 3. The strength for a signature endofunctor (Section 4.2) will invariably derive from the Strength instance of the context extension endofunctor δ, whose implementation makes use of lift: to feed a context map $\Gamma\ \text{-[}\ \mathcal{P}\ \text{]}\!\!\to \Delta$ into a hom $\llbracket\, \mathcal{P} \,,\, \mathcal{X} \,\rrbracket\ \alpha\ (\Theta \dotplus \Gamma)$, one has to extend both its domain and codomain with Θ. The strength proofs feature many of the intricate properties we axiomatised earlier, such as homomorphism and coalgebraic laws. We refer the reader to the formalisation for the details.

2.3.3 Coalgebraic monoids. We revisit the question posed at the end of Section 2.2.3. As shown there, every substitution monoid is a $\square$-coalgebra, since renaming with $\rho : \Gamma \rightsquigarrow \Delta$ can be implemented as substitution with $\eta \circ \rho : \Gamma\ \text{-[}\ \mathcal{X}\ \text{]}\!\!\to \Delta$. This may suggest that renaming for a syntax can be derived from substitution, and so that one only needs to define the latter operation. Attempting this will be futile, however: substitution into terms with variable binding will require weakening (as part of lift), a special case of renaming. Consequently, to equip a family of terms with substitution structure, one needs to have already shown that it is a pointed $\square$-coalgebra.

This a priori renaming structure will not necessarily be equivalent to the one induced by substitution (unlike in the presheaf model, where the two are identified by the quotiented dependent sum used in the definition of the substitution tensor product), but their equivalence is a prerequisite of the free monoid proof in Theorem 3.1 below. We overcome this conflict by axiomatising families with compatible pointed coalgebra and monoid structures, called *coalgebraic monoids*:

$$\begin{aligned}
&\mathsf{record}\ \mathsf{CoalgMon}\ (\mathcal{X} : \mathsf{Family}_{\mathrm{S}}) : \mathsf{Set}\ \mathsf{where} \\
&\quad \mathsf{field}\quad \mathcal{X}^{\square}_{*} : \mathsf{Coalg}_{*}\ \mathcal{X} \\
&\qquad\qquad\ \ \mathcal{X}^{\mathrm{M}} : \mathsf{Mon}\quad \mathcal{X} \\
&\qquad\qquad\ \ \eta\text{-}\mathsf{compat} : \qquad\qquad\qquad \{v : \mathcal{I}\ \alpha\ \Gamma\} \to \eta^{\square}_{*}\ v \equiv \eta^{\mathrm{M}}\ v \\
&\qquad\qquad\ \ \mu\text{-}\mathsf{compat} : \{\rho : \Gamma \rightsquigarrow \Delta\}\ \{t : \mathcal{X}\ \alpha\ \Gamma\} \to \mathsf{r}\ t\ \rho \equiv \mu\ t\ (\eta^{\mathrm{M}} \circ \rho)
\end{aligned}$$

The compatibility laws ensure that the existing $\mathcal{X}^{\square}_{*}$ and derived $\mathcal{X}^{\mathrm{M}}.^{\square}_{*}$ pointed $\square$-coalgebra structures on $\mathcal{X}$ are equivalent, and, in particular, can be exchanged in the first component of str.

We have now set up all the mathematical foundations needed for the development. Next, we move on to the central conceptual tool of our framework: initial algebra semantics.

3 INITIAL ALGEBRA SEMANTICS

Initial algebra semantics [Goguen et al. 1976] is one of the most useful concepts in the study of data types and functional programming. It stems from the observation that every inductive data type T corresponds to an endofunctor F_T of which the data type is an initial algebra. For example, the data type $\mathbb{N}$ of natural numbers has associated endofunctor $F_{\mathbb{N}}(A) = 1 + A$ and comes equipped with an isomorphism $1 + \mathbb{N} \xrightarrow{\cong} \mathbb{N}$. Furthermore, for any $F_{\mathbb{N}}$-algebra A – equivalently, a type A with an element $z \colon A$ and an operation $s \colon A \longrightarrow A$ – one has a unique function $\mathrm{rec}_{\mathbb{N}}(z, s) \colon \mathbb{N} \longrightarrow A$ that satisfies $\mathrm{rec}_{\mathbb{N}}(z, s)(0) = z$ and $\mathrm{rec}_{\mathbb{N}}(z, s)(n+1) = s(\mathrm{rec}_{\mathbb{N}}(z, s)(n))$. This is usually called the *recursor*, *fold*, or *catamorphism* for $\mathbb{N}$ and captures the process of consuming an inhabitant of the inductive type by recursion on its structure.

This idea extends to inductively defined families of types and endofunctors thereon [Altenkirch et al. 2015], such as a data type of intrinsically-typed terms. In particular, the recursor out of an initial family of terms can then be seen as a compositional semantic interpretation map: for example, an interpretation of the STLC in any of its models. As we explore next, initial algebra semantics is not only useful for semantic purposes: it can also be used to implement syntactic operations and prove their laws for free.

3.1 Signature algebras and monoids

Just as the categorical theory, our framework is entirely signature-generic. The metatheory developed here can be freely instantiated for any second-order signature, encompassing algebraic theories, computational calculi, logics, etc. Thus, for the remainder of the section, we fix a sorted-family endofunctor $\Sigma \colon \mathbf{Fam}_{\mathsf{S}} \longrightarrow \mathbf{Fam}_{\mathsf{S}}$ with an instance of coalgebraic strength Σ:Str. Algebras $\Sigma\mathcal{A} \dot{\rightarrow} \mathcal{A}$ for this endofunctor represent families that support the operations of the signature. For example, for the signature Σ_Λ of the STLC, a Σ_Λ-algebra is a family $\mathcal{A}$ equipped with operations $\mathrm{app} \colon \mathcal{A}\,(\alpha \succ \beta)\,\Gamma \times \mathcal{A}\,\alpha\,\Gamma \longrightarrow \mathcal{A}\,\beta\,\Gamma$ and $\mathrm{abs} \colon \mathcal{A}\,\beta\,(\tau \cdot \Gamma) \longrightarrow \mathcal{A}\,(\alpha \succ \beta)\,\Gamma$.

We also consider Σ-algebras with substitution structure, known as Σ-*monoids* [Fiore et al. 1999; Fiore and Saville 2017]. These are families with algebra structure $\Sigma\mathcal{X} \dot{\rightarrow} \mathcal{X}$ and monoid structure $\mathcal{I} \dot{\rightarrow} \mathcal{X} \dot{\rightarrow} \llparenthesis \mathcal{X}, \mathcal{X} \rrparenthesis$ which are compatible with each other; the compatibility is expressed using the coalgebraic strength str on Σ, used to swap the constructor application with the parametrisation by a context map. Note that str uses the pointed $\Box$-coalgebra structure derived from substitution.

```
record ΣMon (𝒳 : Familyₛ) : Set where
  field 𝒳ᴹ     : Mon 𝒳
        alg    : ∑𝒳 ⇾ 𝒳
        μ⟨alg⟩ : {σ : Γ -[ 𝒳 ]→ Δ}{t : ∑𝒳 α Γ} →
                 μ (alg t) σ ≡ alg (str 𝒳ᴮ* 𝒳 (∑₁ μ t) σ)
```

$$\begin{tikzcd}
\Sigma\mathcal{X} \arrow[r, "\Sigma\mu"] \arrow[d, "\mathrm{alg}"'] & \Sigma\llparenthesis \mathcal{X}, \mathcal{X} \rrparenthesis \arrow[r, "\mathrm{str}_{\mathcal{X},\mathcal{X}}"] & \llparenthesis \mathcal{X}, \Sigma\mathcal{X} \rrparenthesis \arrow[d, "\llparenthesis \mathrm{id}, \mathrm{alg} \rrparenthesis"] \\
\mathcal{X} \arrow[rr, "\mu"'] & & \llparenthesis \mathcal{X}, \mathcal{X} \rrparenthesis
\end{tikzcd}$$

3.2 Algebras with metavariables

Our formalisation is novel in that it also incorporates *parametrised metavariables* [Aczel 1978; Hamana 2004; Fiore 2008]: terms that take the syntactic form $\mathfrak{m}\langle -_1, \cdots, -_\ell \rangle$ and are to be imagined as “named holes with slots”, with the “hole” named $\mathfrak{m}$ standing for an unspecified term with ℓ open variables and the “slots” containing terms that occupy the open variables. They too come from a sorted family: for instance, the family $\mathfrak{B}$ containing the metavariables $\mathfrak{m} \colon \mathfrak{B}\,\beta\,[\alpha]$ and $\mathfrak{n} \colon \mathfrak{B}\,\alpha\,[]$ can be used to construct terms such as $r = \big(\lambda x : \alpha.\ \mathfrak{m}\langle x \rangle\big)\big(\mathfrak{n}\langle\,\rangle\big)$ and $t = \mathfrak{m}\langle \mathfrak{n}\langle\,\rangle \rangle$. Metavariables let us abstractly reason about generic terms of some specific syntactic structure: for example, the notion of β-equivalence can be axiomatised as the identification of all common instances of r and t above. For the rest of this section until Section 3.5, we also fix a sorted family $\mathfrak{X}$ of metavariables.

A Σ-algebra structure map on $\mathcal{A}$ captures all the constructors of a second-order syntax, but for full generality, one also needs to account for both variables and metavariables. These are respectively represented by a point var : $\mathcal{I} \Rightarrow \mathcal{A}$ and a metavariable operator mvar : $\mathfrak{X} \Rightarrow ⦇ \mathcal{A} , \mathcal{A} ⦈$. In elementary terms, the latter is a function that associates an $\mathcal{A}$-term mvar $\mathfrak{m}\ \varepsilon\colon \mathcal{A}\,\tau\,\Gamma$ to every parametrised metavariable $\mathfrak{m}\colon \mathfrak{X}\,\tau\,\Pi$ and metavariable environment $\varepsilon\colon \Pi\ \text{-[}\ \mathcal{A}\ \text{]}\!\!\to \Gamma$. For example, the term $\mathfrak{m}\langle t, s\rangle : \mathcal{A}\,\tau\,\Gamma$ for a metavariable $\mathfrak{m} : \mathfrak{X}\,\tau\,[\alpha, \beta]$, and terms $t : \mathcal{A}\,\alpha\,\Gamma$ and $s : \mathcal{A}\,\beta\,\Gamma$ is represented by mvar $\mathfrak{m}\ \lambda\{$ new $\to t$; old new $\to s\ \}$. Families that support this structure will be called $(\Sigma, \mathfrak{X})$-*meta-algebras*, with the expected notion of homomorphism between them.

```
record MetaAlg (𝒜 : Familyₛ) : Set where
  field alg  : ⅀ 𝒜 ⇾ 𝒜
        var  :   𝓘 ⇾ 𝒜
        mvar :   𝔛 ⇾ 〖 𝒜 , 𝒜 〗
```

```
record MetaAlg⇒ (𝒜ᐩ : MetaAlg 𝒜) (ℬᐩ : MetaAlg ℬ) (f : 𝒜 ⇾ ℬ) : Set where
  field ⟨alg⟩  : {t : ⅀ 𝒜 α Γ}                 → f (𝒜.alg t)       ≡ ℬ.alg (⅀ f t)
        ⟨var⟩  : {v : 𝓘 α Γ}                   → f (𝒜.var v)       ≡ ℬ.var v
        ⟨mvar⟩ : {𝔪 : 𝔛 α Π}{ε : Π ~[ 𝒜 ]→ Γ} → f (𝒜.mvar 𝔪 ε) ≡ ℬ.mvar 𝔪 (f ∘ ε)
```

Such meta-algebras and their homomorphisms form a category **MetaAlg**, whose initial object – whenever it exists – will be denoted $\mathbb{T}\,\mathfrak{X}$ (or just $\mathbb{T}$, if the metavariable family is clear from the context) with structural maps alg, var and mvar. The universal property of initial objects states that there is a unique meta-algebra homomorphism sem : $\mathbb{T} \Rightarrow \mathcal{A}$ into any meta-algebra $\mathcal{A}$. Note that, varying $\mathfrak{X}$, $\mathbb{T}$ acts as a mapping from families $\mathfrak{X}$ to $(\Sigma, \mathfrak{X})$-meta-algebras $\mathbb{T}\,\mathfrak{X}$. Adapted from Fiore [2008, Theorem 2], we have the following main result in the study of abstract syntax:

Theorem 3.1. *The initial meta-algebra $\mathbb{T}\,\mathfrak{X}$ is the free Σ-monoid on $\mathfrak{X}$.*

The most interesting implication of this concise statement is that the substitution structure of the free Σ-monoid is induced purely by initiality. Since the initial meta-algebra $\mathbb{T}$ corresponds to the inductively defined grammar of terms (see Section 4.3), it follows that the syntactic structure fully determines the action of substitution. The theorem formally captures the observation that much of syntactic metatheory is uninteresting boilerplate, and the methodology of initial algebra semantics will allow us to extract substitution, compositional interpretations, and related correctness laws from the syntax for free. We outline the proof of the theorem in the rest of the section.

3.3 Parametrised interpretations

The initial algebra approach explains and validates our adherence to formality and efforts to internalise syntactic operations as categorical constructions at the level of sorted families. Since they are maps out of an initial meta-algebra, the renaming $\mathbb{T} \Rightarrow \Box\mathbb{T}$ and the substitution $\mathbb{T} \Rightarrow ⦇ \mathbb{T} , \mathbb{T} ⦈$ operations may be induced by initiality as soon as we show that their codomains $\Box\mathbb{T}$ and $⦇ \mathbb{T} , \mathbb{T} ⦈$ are meta-algebras. This will be derived from the following lemma.

Lemma 3.2. *Given a pointed $\Box$-coalgebra $\mathcal{P}$, a Σ-algebra $\mathcal{A}$, and family maps $\varphi\colon \mathcal{P} \Rightarrow \mathcal{A}$ and $\chi\colon \mathfrak{X} \Rightarrow ⦇ \mathcal{A} , \mathcal{A} ⦈$, the internal hom $⦇ \mathcal{P} , \mathcal{A} ⦈$ acquires a $(\Sigma, \mathfrak{X})$-meta-algebra structure.*

The proof is encapsulated in the Traversal module, instantiations of which will give rise to homomorphic initial algebra interpretations $\mathbb{T} \Rightarrow ⦇ \mathcal{P} , \mathcal{A} ⦈$. It crucially relies on the coalgebraic strength when defining the structure map $\Sigma ⦇ \mathcal{P} , \mathcal{A} ⦈ \Rightarrow ⦇ \mathcal{P} , \Sigma\mathcal{A} ⦈ \Rightarrow ⦇ \mathcal{P} , \mathcal{A} ⦈$. A simple corollary (derived by instantiating $\mathcal{P}$ with $\mathcal{I}$) is that if $\mathcal{A}$ is a meta-algebra, then so is $\Box\,\mathcal{A}$.

```
module Traversal (𝒫□* : Coalg* 𝒫) (alg𝒜 : ⅀𝒜 ⇾ 𝒜) (φ : 𝒫 ⇾ 𝒜) (χ : 𝔛 ⇾ ⦇ 𝒜 , 𝒜 ⦈) where
  Trav⅀ : MetaAlg ⦇ 𝒫 , 𝒜 ⦈
  Trav⅀ = record { alg  = λ h     σ → alg𝒜 (str 𝒫□* 𝒜 h σ)
                 ; var  = λ v     σ → φ (σ v)
                 ; mvar = λ 𝔪 ε σ → χ 𝔪 (λ v → ε v σ) }
```

One may be tempted to immediately induce the substitution map $\mathbb{T} \rightarrowtriangle ⦇ \mathbb{T} , \mathbb{T} ⦈$ as a $\mathbb{T}$-parametrised traversal into $\mathbb{T}$. However, that would require a pointed $\Box$-coalgebra instance on $\mathbb{T}$, which we do not yet have. Our formalism very concretely exhibits the dependence of substitution on renaming.

3.4 ⅀-monoid structure by initiality

The construction of the renaming and substitution maps on the initial meta-algebra $\mathbb{T}$ make extensive use of *initiality*: maps of the form $\mathbb{T} \rightarrowtriangle \mathcal{A}$ are uniquely induced by equipping $\mathcal{A}$ with a meta-algebra structure. The proofs of the renaming and substitution laws are then established by proving that the (composite) maps that correspond to the two sides of an equation are meta-algebra homomorphisms and must therefore be equal.

As an example, the renaming map ren : $\mathbb{T} \rightarrowtriangle \Box\mathbb{T}$ is induced as the unique homomorphism from $\mathbb{T} \in$ **MetaAlg** to the sorted family $\Box\mathbb{T}$ regarded as a meta-algebra by means of Lemma 3.2. The counit law ren t id $= t$ amounts to showing that the mapping $t \mapsto$ ren t id : $\mathbb{T} \rightarrowtriangle \mathbb{T}$ is a meta-algebra homomorphism so that – by initiality of $\mathbb{T}$ – must be equal to the identity $t \mapsto t$. Similarly, proving that the $\mathbf{Fam}_{\mathrm{s}}$-morphisms defined as $t, \rho, \varrho \mapsto$ r t $(\varrho \circ \rho)$ and $t, \rho, \varrho \mapsto$ r (r t ρ) ϱ of type $\mathbb{T} \rightarrowtriangle ⦇ \mathcal{I}, ⦇ \mathcal{I}, \mathbb{T} ⦈ ⦈$ are meta-algebra homomorphisms implies that they must be equal, giving us the comultiplication law. The proofs (available in the formalisation) depend on the structure-preservation properties of ren, the strength laws str-unit and str-nat$_2$, and the corollary str-dist applied to the coalgebraic map j : $\mathcal{I} \rightarrowtriangle ⦇ \mathcal{I}, \mathcal{I} ⦈$.

We therefore have a pointed $\Box$-coalgebra instance $\mathbb{T}^{\Box}_{*}$: Coalg$_*$ $\mathbb{T}$ that is then used as a traversal parameter to induce the substitution map sub : $\mathbb{T} \rightarrowtriangle ⦇ \mathbb{T} , \mathbb{T} ⦈$. Initiality once again helps us prove the substitution monoid laws abstractly; in fact, the reasoning steps very closely resemble those needed in the proof of the renaming structure, and they can all be presented in the form of clear, categorical proofs. Once the instances $\mathbb{T}^{\mathrm{M}}$: Mon $\mathbb{T}$ and $\mathbb{T}^{\Box\mathrm{M}}$: CoalgMon $\mathbb{T}$ are derived, we further establish that $\mathbb{T}$ is a ⅀-monoid, the proof of which relies on the fact that coalgebraic monoids identify the given coalgebra instance $\mathbb{T}^{\Box}_{*}$ with the one induced from substitution.

The effort put into identifying the notions of coalgebraic map and strength (Section 2.3.2) pays off repeatedly: the comultiplication law, coalgebraic axioms, and substitution associativity (the *fusion lemmas* of Allais et al. [2017, Section 9.2]) are all established using str-dist, instantiated at different coalgebraic maps (the application map j, renaming ren and substitution sub) to derive the generalised forms of the four lifting laws listed by Benton et al. [2012]. The renaming and substitution operations and their correctness laws are derived in an elegant, mathematically-motivated manner, with no auxiliary definitions and ad-hoc lemmas, or additional reasoning machinery needed (such as the bisimulation/fusion framework developed by AACMM [2021]).

3.5 Free ⅀-monoid structure

In the previous section we generically derived a ⅀-monoid structure for $\mathbb{T}\,\mathfrak{X}$, for any family of metavariables $\mathfrak{X}$. This gives us a lawful substitution operation on $\mathbb{T}\,\mathfrak{X}$ that can be used in further developments, such as an operational semantics or equational theory. We can go further, however, and characterise $\mathbb{T}\,\mathfrak{X}$ as the *free ⅀-monoid on* $\mathfrak{X}$: given any ⅀-monoid $\mathcal{M}$ and metavariable interpretation $\omega \colon \mathfrak{X} \rightarrowtriangle \mathcal{M}$, there is a unique ⅀-monoid homomorphism $\omega^{\#} \colon \mathbb{T}\,\mathfrak{X} \rightarrowtriangle \mathcal{M}$ that satisfies $\omega^{\#}($mvar $\mathfrak{m}$ var$) = \omega(\mathfrak{m})$ for every metavariable $\mathfrak{m} \colon \mathfrak{X}\,\tau\,\Pi$. As expected, $\omega^{\#}$ is constructed by initiality, using the fact that the $\sum$-monoid $\mathcal{M}$ is a meta-algebra with metavariable operator $\mathfrak{X} \xrightarrow{\omega} \mathcal{M} \xrightarrow{\mu} \llparenthesis \mathcal{M}, \mathcal{M} \rrparenthesis$. Initiality is also used to establish that the unique map $\omega^{\#}$ preserves substitution, i.e. that it is a $\sum$-monoid homomorphism.

Freeness induces a free-forgetful adjunction between the categories of sorted families and of $\sum$-monoids, and makes $\mathbb{T}$ into the *free* $\sum$*-monoid monad* on sorted families, whose Kleisli extension $(\mathfrak{X} \rightarrowtriangle \mathbb{T}\,\mathfrak{Y}) \longrightarrow (\mathbb{T}\,\mathfrak{X} \rightarrowtriangle \mathbb{T}\,\mathfrak{Y})$ acts as a form, albeit limited, of *metasubstitution* [Hamana 2004]: occurrences of metavariables from a family $\mathfrak{X}$ in a term of $\mathbb{T}\,\mathfrak{X}$ get replaced with terms of $\mathbb{T}\,\mathfrak{Y}$ according to a mapping $\mathfrak{X} \rightarrowtriangle \mathbb{T}\,\mathfrak{Y}$. This important notion and its generalisation are discussed next.

3.6 Metasubstitution

A main difference between metasubstitution and object-level substitution is that the former is *capture-permitting*: a metavariable $\mathfrak{m}$ in a term $\lambda x : \mathbb{N}.\ \mathfrak{m}\langle x\rangle$ can be replaced with terms that contain free occurrences of x; so that valid instances could be $\lambda x.\ x$, $\lambda x.\ x \cdot (x+1)$, or $\lambda x.\ \mathfrak{p}\langle x, x\rangle$ for a metavariable $\mathfrak{p} \colon \mathfrak{B}\,\mathbb{N}\,[\mathbb{N},\mathbb{N}]$. The aforementioned metasubstitution map $(\mathfrak{X} \rightarrowtriangle \mathbb{T}\,\mathfrak{Y}) \longrightarrow (\mathbb{T}\,\mathfrak{X} \rightarrowtriangle \mathbb{T}\,\mathfrak{Y})$ is limited in that the only free variables that the mapping $\zeta \colon \mathfrak{X} \rightarrowtriangle \mathbb{T}\,\mathfrak{Y}$ may refer to are the parameters of the metavariable: for example, in the open term $x \colon \mathbb{N}, y \colon \mathbb{N} \vdash \mathfrak{m}\langle y\rangle$ one cannot instantiate $\mathfrak{m}$ with $x + y$, since x is not a parameter of $\mathfrak{m}$ and is therefore not in scope in the output of ζ. As we wish metasubstitution to model "textual replacement", this is an unnatural restriction.

Our goal then is to capture the following intuition [Fiore 2008]: the term that replaces a metavariable $\mathfrak{m}$ can feature both the parameters of $\mathfrak{m}$, and all the object-level variables in scope at the occurrence of $\mathfrak{m}$ in the term. In elementary terms, the type of the operation may be given as

$$\mathbb{msub} \colon \forall\{\alpha\,\Gamma\} \longrightarrow (t \colon \mathbb{T}\,\mathfrak{X}\,\alpha\,\Gamma) \longrightarrow \big(\zeta \colon \forall\{\tau\,\Pi\} \longrightarrow \mathfrak{X}\,\tau\,\Pi \longrightarrow \mathbb{T}\,\mathfrak{Y}\,\tau\,(\Pi \dotplus \Gamma)\big) \longrightarrow \mathbb{T}\,\mathfrak{Y}\,\alpha\,\Gamma$$

As usual, we aim to represent this operation as a morphism of families, and ideally derive it by initiality. Key to this is recognising the type $\forall\{\Pi\} \longrightarrow \mathfrak{X}\,\tau\,\Pi \longrightarrow \mathbb{T}\,\mathfrak{Y}\,\tau\,(\Pi \dotplus \Gamma)$ as the linear exponential of families $(\mathfrak{X} \multimap \mathbb{T}\,\mathfrak{Y})\,\tau\,\Gamma$ (Section 2.1.2). The derived definition $[\,\mathcal{X} \multimap \mathcal{Y}\,]\,\Gamma \triangleq \forall\{\tau\} \longrightarrow (\mathcal{X} \multimap \mathcal{Y})\,\tau\,\Gamma$ combines two sorted families into an unsorted one. We also modify the family exponential to take an unsorted family as first argument; that is, overloading notation: $(X \Rrightarrow \mathcal{Y})\,\alpha \triangleq X \Rrightarrow (\mathcal{Y}\,\alpha)$. The type of metasubstitution may be now succinctly expressed as

$$\mathbb{msub} \colon \mathbb{T}\,\mathfrak{X} \rightarrowtriangle \big([\mathfrak{X} \multimap \mathbb{T}\,\mathfrak{Y}] \Rrightarrow \mathbb{T}\,\mathfrak{Y}\big)$$

We can then use the following general result to induce $\mathbb{msub}$ by initiality: given a $\sum$-monoid $\mathcal{M}$, the family $[\,\mathfrak{X} \multimap \mathcal{M}\,] \Rrightarrow \mathcal{M}$ acquires a $(\sum, \mathfrak{X})$-meta-algebra structure. However, this only applies if we assume an additional property of the signature endofunctor $\sum$: it comes with an *exponential strength* estr: $\sum(X \Rrightarrow \mathcal{Y}) \rightarrowtriangle (X \Rrightarrow \sum\mathcal{Y})$ (equivalent to the usual Cartesian strength) for every unsorted $\square$-coalgebra X (i.e. unsorted family X with an operation $X\,\Gamma \longrightarrow (\Gamma \rightsquigarrow \Delta) \longrightarrow X\,\Delta$). With this in place, the meta-algebra structure on $[\,\mathfrak{X} \multimap \mathcal{M}\,] \Rrightarrow \mathcal{M}$ is given as follows:

$$\begin{aligned}
&\mathsf{MSub}^{\Sigma} : (\mathfrak{X} : \mathsf{Family}_{\mathsf{S}}) \longrightarrow \Sigma\mathsf{Mon}\ \mathcal{M} \longrightarrow \mathsf{MetaAlg}\ \mathfrak{X}\ ([\,\mathfrak{X} \multimap \mathcal{M}\,] \Rrightarrow \mathcal{M})\\
&\mathsf{MSub}^{\Sigma}\ \mathfrak{X}\ \mathcal{M}^{\Sigma\mathrm{M}} = \mathsf{record}\ \{\ \mathsf{alg}\ \ \ = \lambda\, t\qquad \zeta \to \mathcal{M}.\mathsf{alg}\ (\mathsf{estr}\ [\,\mathfrak{X} \multimap \mathcal{M}.^{\square}\,]^{\square}\ \mathcal{M}\ t\ \zeta)\\
&\qquad\qquad\qquad\qquad\qquad\ ;\ \mathsf{var}\ \ \ = \lambda\, v\qquad \zeta \to \mathcal{M}.\eta\ v\\
&\qquad\qquad\qquad\qquad\qquad\ ;\ \mathsf{mvar} = \lambda\, \mathfrak{m}\ \varepsilon\ \zeta \to \mathcal{M}.\mu\ (\zeta\ \mathfrak{m})\ (\mathsf{copair}\ \mathcal{M}\ (\lambda\, v \to \varepsilon\ v\ \zeta)\ \mathcal{M}.\eta)\ \}
\end{aligned}$$

- For alg, we use estr to swap the $\sum$-algebra structure and dependence on the metasubstitution map. We make use of the fact that $[\,\mathfrak{X} \multimap \mathcal{P}\,]$ is an unsorted coalgebra if $\mathcal{P}$ is a sorted coalgebra.
- For var, we ignore ζ entirely and use the point of $\mathcal{M}$.
- For mvar, we start by looking up the metavariable $\mathfrak{m} \colon \mathfrak{X}\,\tau\,\Pi$ in the linear metasubstitution map $\zeta : [\,\mathfrak{X} \multimap \mathcal{M}\,]\,\Gamma$, obtaining the term $\zeta\ \mathfrak{m} : \mathcal{M}\ \tau\ (\Pi \dotplus \Gamma)$ in an extended context. We need to tweak this further to get the required output in $\mathcal{M}\,\tau\,\Gamma$, which we do by applying a substitution

$(\Pi \dotplus \Gamma) \mathrel{-[\ \mathcal{M}\ ]\!\!\to} \Gamma$ to $\zeta\ \mathfrak{m}$. It is constructed as the copairing of the unit $\mathcal{M}.\eta : \Gamma \mathrel{-[\ \mathcal{M}\ ]\!\!\to} \Gamma$ and the substitution map $\Pi \mathrel{-[\ \mathcal{M}\ ]\!\!\to} \Gamma$ which looks up a variable in context Π in $\mathfrak{m}$'s metavariable environment $\varepsilon : \Pi \mathrel{-[\ ([\ \mathfrak{X} \multimap \mathcal{P}\ ] \Rightarrow \mathcal{M})\ ]\!\!\to} \Gamma$ and applies the resulting parametrised term to ζ.

This latter specification is quite a mouthful: metasubstitution (which is derived by initiality into the meta-algebra $\mathsf{MSub}^{\Sigma}\ \mathfrak{X}\ \mathbb{T}^{\Sigma\mathrm{M}}$) is less a form of "textual replacement" and more an intricate surgical procedure with several interconnected parts. The metavariable-preservation property of $\mathbb{msub}$ illuminates the role of recursively applying metasubstitution to the elements of the metavariable environment ε, then substituting the terms into the parameters of $\zeta\ \mathfrak{m}$:

$$\mathbb{msub}\ (\mathbb{mvar}\ \mathfrak{m}\ \varepsilon)\ \zeta \equiv \mathbb{sub}\ (\zeta\ \mathfrak{m})\ (\mathsf{copair}\ (\mathbb{T}\mathfrak{Y})\ (\lambda\ v \to \mathbb{msub}\ (\varepsilon\ v)\ \zeta)\ \mathbb{var})$$

As an illustration of metasubstitution, consider the open term $x : \mathbb{N} \vdash \lambda y : \mathbb{N}.\ \mathfrak{a}\langle x+1, \mathfrak{b}\langle y\rangle\rangle : \mathbb{N}$ and the metasubstitution mapping $\zeta = (\mathfrak{a}\langle m,n\rangle \mapsto \mathfrak{c}\langle m\rangle \times n; \mathfrak{b}\langle m\rangle \mapsto \mathfrak{c}\langle m+x\rangle)$ in the global context $x\colon \mathbb{N}$ (note that the latter term includes both the parameter m and the free variable x). The evaluation of the metasubstitution proceeds as follows (where $\mathbb{sub}\ t\ [\cdots]$ and $\mathbb{msub}\ t\ \langle\!\langle\cdots\rangle\!\rangle$ denote substitution and metasubstitution into t, respectively):

$$\begin{aligned}
&\mathbb{msub}\ (\lambda y : \mathbb{N}.\ \mathfrak{a}\langle x+1, \mathfrak{b}\langle y\rangle\rangle)\ \langle\!\langle \zeta \rangle\!\rangle \\
\equiv\ &\lambda y : \mathbb{N}.\ \mathbb{msub}\ (\mathfrak{a}\langle x+1, \mathfrak{b}\langle y\rangle\rangle)\ \langle\!\langle \mathsf{wk}\ \zeta \rangle\!\rangle && ① \\
\equiv\ &\lambda y : \mathbb{N}.\ \mathbb{sub}\ (\mathfrak{c}\langle m\rangle \times n)\ \big[m \mapsto x+1, n \mapsto \mathbb{msub}\ (\mathfrak{b}\langle y\rangle)\ \langle\!\langle \mathsf{wk}\ \zeta\rangle\!\rangle\big] && ② \\
\equiv\ &\lambda y : \mathbb{N}.\ \mathbb{sub}\ (\mathfrak{c}\langle m\rangle \times n)\ \big[m \mapsto x+1, n \mapsto \mathbb{sub}\ (\mathfrak{c}\langle m+x\rangle)\ [m \mapsto y]\big] && ③ \\
\equiv\ &\lambda y : \mathbb{N}.\ \mathbb{sub}\ (\mathfrak{c}\langle m\rangle \times n)\ \big[m \mapsto x+1, n \mapsto \mathfrak{c}\langle y+x\rangle\big] && ④ \\
\equiv\ &\lambda y : \mathbb{N}.\ \mathfrak{c}\langle x+1\rangle \times \mathfrak{c}\langle y+x\rangle && ⑤
\end{aligned}$$

In step ① we push the metasubstitution under the binder. A crucial component of this step is the weakening of the terms in the metasubstitution mapping ζ represented here by $\mathsf{wk}\ \zeta$. Indeed, since the local context changes from $x\colon \mathbb{N}$ to $y\colon \mathbb{N}, x\colon \mathbb{N}$, the de Bruijn index of the variable x has to be shifted without altering the parameters m, n (for example, $m\colon \mathbb{N}, x\colon \mathbb{N} \vdash \mathfrak{c}\langle m+x\rangle$ is renamed to $m\colon \mathbb{N}, y\colon \mathbb{N}, x\colon \mathbb{N} \vdash \mathfrak{c}\langle m+x\rangle$ by shifting the index of x). Although this modification makes no difference with the named-variable representation displayed here, in practice it becomes an explicit application of $\mathbb{ren}$ – implemented as part of the exponential strength for δ. It is worth noting the delicate interplay between $\mathbb{ren}$, $\mathbb{sub}$, and $\mathbb{msub}$. In step ② we apply the metasubstitution to $\mathfrak{a}$ by looking up the term $\mathfrak{c}\langle m\rangle \times n$, substituting the contents of $\mathfrak{a}$'s metavariable environment for m and n, and recursively metasubstituting $\mathsf{wk}\ \zeta$ into $\mathfrak{b}\langle y\rangle$ (and into $x+1$, where it is a no-op). The mappings of variables x, y to themselves are omitted. Steps ③ and ④ evaluate the recursive calls, applying an object-level substitution to $\mathfrak{c}\langle m+x\rangle$ that replaces the parameter m with the variable y (and the global variable x with itself). The final substitution is evaluated at step ⑤.

3.7 Equational systems

An immediate application of metasubstitution is building generic *equational systems* for second-order languages [Fiore and Hur 2010; Fiore and Mahmoud 2010]. By specifying the axioms of an equational theory with the use of metavariables, one can use metasubstitution to extract axiom instances between terms and apply rewrite rules within compound expressions. For example, β-equivalence arises from instances of the axiom

$$\mathfrak{b}\colon [\alpha]\beta,\ \ \mathfrak{a}\colon []\alpha\ \ \triangleright\ \ \emptyset\ \ \vdash\ \ (\lambda x : \alpha.\ \mathfrak{b}\langle x\rangle)\ \mathfrak{a}\langle\rangle\ \ \approx\ \ \mathfrak{b}\langle \mathfrak{a}\langle\rangle\rangle\ \ : \beta$$

with terms (of the appropriate types and contexts) metasubstituted for the metavariables $\mathfrak{a}$ and $\mathfrak{b}$. Generic equality can be then directly encoded in Agda as the smallest equivalence relation closed under a given relation $\mathsf{Axiom} : \forall(\mathfrak{X}\ \Gamma\ \{\alpha\}) \longrightarrow \mathbb{T}\ \mathfrak{X}\ \alpha\ \Gamma \longrightarrow \mathbb{T}\ \mathfrak{X}\ \alpha\ \Gamma \longrightarrow \mathsf{Set}$ and metasubstitution:

```
data _▷_⊢_≈_ : (𝔛 : Familyₛ)(Γ : Ctx){α : T} → 𝕋 𝔛 α Γ → 𝕋 𝔛 α Γ → Set₁ where
  eq  : t ≡ s → 𝔛 ▷ Γ ⊢ t ≈ s
  sy  : 𝔛 ▷ Γ ⊢ t ≈ s → 𝔛 ▷ Γ ⊢ s ≈ t
  tr  : 𝔛 ▷ Γ ⊢ t ≈ s → 𝔛 ▷ Γ ⊢ s ≈ u → 𝔛 ▷ Γ ⊢ t ≈ u
  ax  : Axiom 𝔛 Γ t s → 𝔛 ▷ Γ ⊢ t ≈ s
  ms : 𝔛 ▷ Γ ⊢ t ≈ s → (ζ ξ : [ 𝔛 ⊸ 𝕋 𝔜 ] Γ) →
       (∀{τ Π}(𝔪 : 𝔛 τ Π) → 𝔜 ▷ Π ∔ Γ ⊢ ζ 𝔪 ≈ ξ 𝔪) → 𝔜 ▷ Γ ⊢ msub t ζ ≈ msub s ξ
```

The ms constructor expresses that whenever two terms t and s are equivalent, and two instantiations for their metavariable context ζ and ξ are equivalent (for every metavariable), then performing the metasubstitution also gives equivalent terms. Using the equivalence constructors one can derive useful proof combinators and a library for equational reasoning; for example, ax≈ equates two terms via the instantiation of an axiom:

```
ax≈ : Axiom 𝔛 Γ t s → (ζ : [ 𝔛 ⊸ 𝕋 𝔜 ] Γ) → 𝔜 ▷ Γ ⊢ msub t ζ ≈ msub s ζ
ax≈ a ζ = ms (ax a) ζ ζ (λ _→ eq refl)
```

The biggest gains, however, come from not having to tediously encode the congruence rules for every subterm of every term of the syntax. To rewrite a deeply nested subexpression, we simply mark its location in the term with a "typed hole" implemented as a distinguished metavariable ◌, and use ms to instantiate it with the two sides of an equality rule: for example, $f \approx g$ and $(◌\, a) \approx (◌\, a)$ imply that $f\,a = \mathsf{msub}\ (◌\, a)\ (\lambda\{◌ \to f\}) \overset{\mathsf{ms}}{\approx} \mathsf{msub}\ (◌\, a)\ (\lambda\{◌ \to g\}) = g\,a$. Further examples of this and other combinators are given in Section 5. An important future development is generically proving the soundness and completeness of second-order equational logic.

We thus conclude our abstract development of initial algebra semantics and move on to the discussion of second-order signatures.

4 GENERIC SIGNATURES

The abstract development discussed so far was entirely generic over the second-order signature and term syntax. In this section we discuss how endofunctors Σ are constructed from descriptions of syntax signatures, the benefits and drawbacks of various term representations, and how we leverage code generation to turn our library into a practical framework for language formalisation. Thanks to our modular implementation, we have several choices in each of the following matters:

- how to encode the signature of a second-order syntax (e.g. binding algebras [Fiore et al. 1999], indexed containers [Altenkirch et al. 2015], AACMM [2021]-style Descriptions);
- how to convert the signature into a $\mathbf{Fam}_s$ endofunctor Σ (e.g. polynomial functors [Fiore 2012; Arkor and Fiore 2020], higher- or first-order argument collections, Desc interpretations);
- how to define the data type for the initial $(\Sigma, \mathfrak{X})$-meta-algebra (implicit or explicit encodings).

Each of these have their benefits and drawbacks, and we identify three choices that work particularly well together and combine convenience, flexibility, and appropriate computational behaviour.

4.1 Binding signatures

Binding signatures were introduced by Aczel [1978] as a generalisation of standard algebraic signatures to languages with variable binding. Our formalisation will use the typed variant of the notion as given in [Fiore and Hur 2010]:

Definition 4.1. A *second-order signature* $\Sigma = (T, O, |-|)$ is specified by a set of sorts T, a set of operators O, and an arity function $|-| : O \longrightarrow \mathsf{List}\ ((\mathsf{List}\ T) \times T) \times T$.

For an operator o, the tuple $|o| = ([(\vec{\alpha}_1, \beta_1), (\vec{\alpha}_2, \beta_2), \ldots, (\vec{\alpha}_n, \beta_n)], \tau)$ consists of the output sort $\tau \in T$, and a list of n arguments of type β_i, with each argument binding variables as listed in $\vec{\alpha}_i$. We write this more concisely as $o\colon [\vec{\alpha}_1]\beta_1, \ldots, [\vec{\alpha}_n]\beta_n \longrightarrow \tau$, omitting empty bound variable lists.

Example 4.2. The second-order signature of the simply-typed λ-calculus over a base type has the set of types T generated inductively from a base type N and a binary function type $\succ$, and the two type-indexed families of operators

$$\mathsf{app}_{\alpha,\beta} \;:\; (\alpha \succ \beta),\ \alpha \;\longrightarrow\; \beta \qquad\qquad \mathsf{lam}_{\alpha,\beta} \;:\; [\alpha]\beta \;\longrightarrow\; (\alpha \succ \beta)$$

This definition of signatures can be adapted to Agda almost verbatim, using Ctx in place of List T:

```
record Signature (O : Set) : Set₁ where
  constructor sig
  field |_| : O ⟶ List (Ctx × T) × T

Arity : O ⟶ List (Ctx × T)
Arity o = proj₁ | o |

Sort : O ⟶ T
Sort o = proj₂ | o |
```

The set T of types (or sorts) is normally given as an inductive data type, and O as an enumeration of operators. For example, the STLC has the following sorts and operator symbols:

```
data ΛT : Set where
  N   : ΛT
  _≻_ : ΛT ⟶ ΛT ⟶ ΛT

data Λₒ : Set where
  appₒ lamₒ : {α β : ΛT} ⟶ Λₒ
```

The Signature instances are direct translations of Example 4.2 above. One can use some simple shorthands for specifying arguments and bound variables to make the declaration concise.

```
Λ:Sig : Signature Λₒ
Λ:Sig = sig λ where (appₒ {α}{β}) → (⊢₀ α ≻ β) , (⊢₀ α)  ⟼₂  β
                    (lamₒ {α}{β}) →         (α ⊢₁ β)      ⟼₁  α ≻ β
```

4.2 Signature endofunctor

The signature contains all the information needed to determine the syntactic structure of a language. To make use of the abstract development in Section 3, we need to convert a Signature into a sorted-family endofunctor $\sum$, which captures the way in which constructors of the syntax are associated with arguments. For example, given the signature of the STLC, an element of $\sum \mathcal{X}\, \beta\, \Gamma$ may be the operator app associated with two $\mathcal{X}$-terms $f\colon \mathcal{X}\ (\alpha \succ \beta)\, \Gamma$ and $a\colon \mathcal{X}\, \alpha\, \Gamma$, while an element of $\sum \mathcal{X}\ (\alpha \succ \beta)\, \Gamma$ may be the operator lam with a term $b\colon \mathcal{X}\, \beta\, (\alpha \cdot \Gamma)$.

For technical reasons, that we will expand upon later, we choose to represent the "collection" of arguments of an operator as a tuple of terms. An alternative would be a higher-order encoding as a mapping from an argument index to a term (similar to the implementation of substitutions as context maps); however, even though constructing the Strength for such a representation would be easier, it complicates the initiality proof which we wish to keep as simple as possible.

```
Arg : List (Ctx × T) ⟶ Familyₛ ⟶ Family
Arg []            𝒳 Γ = ⊤
Arg ((Θ , τ) ∷ as) 𝒳 Γ = δ Θ 𝒳 τ Γ × Arg as 𝒳 Γ
```

Note the use of the context extension endofunctor δ: it is used to add the new variables bound by an argument to the the global context, making all variables available in the body of the binder.

We are ready to give the definition of the signature endofunctor for a signature $(T, O, \mathsf{Arity}, \mathsf{Sort})$:

```
⅀ : Familyₛ ⟶ Familyₛ
⅀ 𝒳 α Γ = Σ[ o ∈ O ] (α ≡ Sort o × Arg (Arity o) 𝒳 Γ)
```

An element of the set $⅀\ \mathcal{X}\ \alpha\ \Gamma$ is a dependent tuple consisting of an operator symbol $o \in O$, a proof that the output sort of the operator is α, and a tuple of $\mathcal{X}$-terms for each operand of the operator of the type and extension context given by the operator arity. For example, an element of $⅀\ \mathcal{X}\ \beta\ \Gamma$ is (app, refl, $(f, t,$ tt)), for terms $f : \mathcal{X}\ (\alpha \succ \beta)\ \Gamma$ and $t : \mathcal{X}\ \alpha\ \Gamma$. One can suppress the tt for operators of positive arity by adding a case for a singleton argument list in the definition of Arg, and use a pattern synonym [Pickering et al. 2016] to hide the refl element, writing app ⋮ (f, t) for the above.

The only other construction we need is the Strength instance for ⅀. The family $\mathcal{X}$ in a $(⅀\ \mathcal{X})$-term is only used in the argument list, so the ⅀-strength can be easily derived from the Arg-strength Arg as ⟅ 𝒫 , 𝒳 ⟆ ⇾ ⟅ 𝒫 , Arg as 𝒳 ⟆. Applying strength to the tuple of arguments simply applies it to every component of type δ ⊖ ⟅ 𝒫 , 𝒳 ⟆ $\tau\ \Gamma$ – and this is nothing but the strength instance for δ which we constructed back in Section 2.3.2.

```
strᴬ : (𝒫ᵒ₊ : Coalg₊ 𝒫)(𝒳 : Familyₛ)(as : List (Ctx × T)) ⟶ Arg as ⟅ 𝒫 , 𝒳 ⟆ ⇾ ⟅ 𝒫 , Arg as 𝒳 ⟆
strᴬ 𝒫ᵒ₊ 𝒳 []                tt        σ = tt
strᴬ 𝒫ᵒ₊ 𝒳 ((Θ , τ) ∷ as) (h , hs) σ = (δ:Str.str Θ 𝒫ᵒ₊ 𝒳 h σ) , (strᴬ 𝒫ᵒ₊ 𝒳 as hs σ)
```

The strength laws are similarly established by pointwise application of the appropriate δ:Str fields to the elements of the argument tuple. Extending Arg-strength to ⅀ is easy, since the operation does not modify the operator or sort equality proof. In addition to ⅀:Str below, we also have an instance of exponential strength for ⅀ derived via weakening.

```
⅀:Str : Strength ⅀F
⅀:Str = record { str      = λ 𝒫ᵒ₊ 𝒳 (o , e , a) σ → o , e , (strᴬ 𝒫ᵒ₊ 𝒳 (Arity o) a σ)
               ; str-nat₁ = λ fᵒ₊⇒  (o , e , a) σ → cong (o , e ,_) (strᴬ-nat₁ fᵒ₊⇒ (Arity o) a σ)
               ; str-nat₂ = λ g     (o , e , a) σ → cong (o , e ,_) (strᴬ-nat₂ g     (Arity o) a σ) ; … }
```

4.3 Term syntax

The final piece of the puzzle is constructing the initial $(⅀, \mathfrak{X})$-meta-algebra $\mathbb{T}$ from a second-order signature endofuctor. Such initial algebras correspond to inductive data types whose constructors combine $\mathbb{T}$-terms into other $\mathbb{T}$-terms, allowing for arbitrarily nested syntactic structure.

We have several choices in the approach we take. One is to treat the tuples (op ⋮ $(a_1, \ldots, a_n)$) as the terms of the syntax, directly encoding the ⅀-algebra structure as a unified term constructor con : ⅀ 𝕋 ⇾ 𝕋; the other is the more common implementation with one constructor for each operator, applicable for signatures with a finite set of operators.

Implicit encoding. Alongside the ⅀-algebraic structure, terms of a second-order syntax also have to include constructors for variables and metavariables. This suggests the following generic data type of terms for an arbitrary signature:

```
data Tm : Familyₛ where                           data Sub (𝒳 : Familyₛ) : Ctx ⟶ Ctx ⟶ Set where
  con  : ⅀ Tm τ Γ               ⟶ Tm τ Γ            •   : Sub 𝒳 ∅ Γ
  var  : ℐ τ Γ                  ⟶ Tm τ Γ            _◂_ : 𝒳 α Γ ⟶ Sub 𝒳 Π Γ ⟶ Sub 𝒳 (α · Π) Γ
  mvar : 𝔛 τ Π ⟶ Sub Tm Π Γ ⟶ Tm τ Γ
```

Note the use of Sub (defined on the right above) in place of a context map to represent the metavariable environment. It is a first-order, inductive encoding of a simultaneous substitution of terms in context Γ for every variable in context Π, which, while isomorphic to context maps (with conversion functions lookup and tabulate), is a more appropriate choice for syntax. Recalling that metasubstitution recurses into the metavariable environment, in a higher-order representation the recursive call would get suspended in the body of a λ-abstraction and lead to terms with unevaluated expressions. With context maps, the application sem (mvar $\mathfrak{m}$ (λ{ new → con (op ⁝ t) })) would only normalise to mvar $\mathfrak{m}$ (λ{ new → sem (con (op ⁝ t)) }), and sem would not get pushed further under the constructor con unless the environment function is applied to new. In contrast, using the first-order encoding Sub, each element of the sequence gets fully normalised, so, as desired, sem (mvar $\mathfrak{m}$ (con (op ⁝ t) ◂ •)) reduces to mvar $\mathfrak{m}$ (con (op ⁝ sem t) ◂ •). The same reasoning was behind our choice to represent operator arguments as tuples, rather than higher-order assignments – though the strength instance would have been simpler, specifying arguments via case-analysis is cumbersome, and syntactic operations only recurse one layer deep into subterms.

The proof of initiality of Tm asks us to define a recursive function sem : Tm ⇾ $\mathcal{A}$ for any meta-algebra $\mathcal{A}$, translating constructors of Tm to the algebra and (meta)variable structure of $\mathcal{A}$. There is an issue with the obvious definition: recursively interpreting the subterms of a constructor involves mapping sem over a tuple of terms, which Agda does not recognise as a terminating function call. Allais et al. [2021, Section 4] also encountered this problem and proposed the use of Agda's *sized types* [Abel 2010] (which, unfortunately, are logically unsound in the current Agda version 2.6.2) to mark a term as strictly "larger" than its subterms. The workaround suits their needs of defining sem – at the expense of having to carry around the size index everywhere – but it does not extend to proving the uniqueness of sem which our framework closely relies on (see Pitts [2019] for the technical details of the same issue in the context of general F-algebras).

The solution to this issue is surprisingly simple: instead of recursively interpreting the subterms using a functorial mapping ($\mathcal{X}$ ⇾ $\mathcal{Y}$) → (Arg *as* $\mathcal{X}$ Γ → Arg *as* $\mathcal{Y}$ Γ) (and similarly for Sub), we inline the transformations as the mutually recursive functions 𝔸 and 𝕊 that apply sem : Tm ⇾ $\mathcal{A}$ to subterms directly. The termination checker is satisfied without the need for sized types!

```
𝔸 : ∀ as → Arg as Tm Γ → Arg as 𝒜 Γ          𝕊 : Sub Tm Π Γ → Π –[ 𝒜 ]→ Γ
𝔸 []        tt      = tt                       𝕊 (t ◂ σ) new     = sem t
𝔸 (a ∷ as) (t , ts) = (sem t , 𝔸 as ts)        𝕊 (t ◂ σ) (old v) = 𝕊 σ v

              sem (con (o , e , a)) = alg (o , e , 𝔸 (Arity o) a)
              sem (var v)           = var v
              sem (mvar 𝔪 ε)        = mvar 𝔪 (𝕊 ε)
```

The proof that sem is a unique $(\sum, \mathfrak{X})$-meta-algebra homomorphism is also quite straightforward, only requiring a few mutually inductive lemmas about 𝔸 and 𝕊. We then have the following result:

Theorem 4.3. *Tm is an initial* $(\sum, \mathfrak{X})$*-meta-algebra.*

We can thus instantiate our previous metatheory with this concrete data type to get access to substitution and its correctness properties, sound compositional interpretations in models, etc.

Explicit encoding. The implicit encoding achieves our conceptual goals: it is a first-order initial meta-algebra for an arbitrary second-order signature. Its main practical disadvantage is that the term syntax is closely tied to the algebraic framework and forces users to adopt a cumbersome encoding of terms, rather than the natural and elegant "one constructor for each typing rule" principle of intrinsic typing. Pattern synonyms may be used to simplify the surface syntax, though we found them quite fragile when working with parametrised modules – not to mention that pattern synonyms are untyped and feel inherently "hacky" for something as important as the terms of a formalised language. Ideally, we would like users to be able to adopt our framework as seamlessly as possible, perhaps even plugging it into an existing formalisation without changing the fundamental data type of syntactic terms.

This is very much possible, thanks to our rigid separation between signatures, endofunctors and initial meta-algebras. Defining the initial meta-algebra instance for an existing data type is not an especially laborious task: one needs a recursive initial interpretation function, a homomorphism proof, and an inductive uniqueness proof. Metavariables still require the mutually recursive transformation 𝕊 (defined as before and so omitted here), but now that one is manually recursing into subterms, the transformation 𝔸 is not needed.

```
data Λ : Familyₛ where                         sem : Λ ⇾ 𝒜
  var  : 𝐼 τ Γ → Λ τ Γ                         sem (var v)      = var v
  mvar : 𝔛 τ Π → Sub Λ Π Γ → Λ τ Γ             sem (mvar 𝔪 ε) = mvar 𝔪 (𝕊 ε)
  app  : Λ (α ↣ β) Γ → Λ α Γ → Λ β Γ           sem (app g a)    = alg (appₒ ⋮ sem g , sem a)
  lam  : Λ β (α ∙ Γ) → Λ (α ↣ β) Γ             sem (lam b)      = alg (lamₒ ⋮ sem b)
```

The homomorphism instance uses a simple lemma 𝕊-tab about the interaction of 𝕊 and tabulate (for all context maps ε, 𝕊 (tabulate ε) = sem ∘ ε), and the ⅀-algebra homomorphism proof, which is satisfied merely by pattern-matching on the operand and sort equality proof.

```
sem᷾⇒ : MetaAlg⇒ Λ᷾ 𝒜᷾ sem
sem᷾⇒ = record { ⟨alg⟩ = λ {t = t} → ⟨alg⟩ t ; ⟨var⟩ = refl
                ; ⟨mvar⟩ = λ {𝔪 = 𝔪}{ε} → cong (mvar 𝔪) (𝕊-tab ε) }
  where ⟨alg⟩ : (t : ⅀ Λ α Γ) → sem (Λ᷾.alg t) ≡ 𝒜᷾.alg (⅀₁ sem t)
        ⟨alg⟩ (appₒ ⋮ _) = refl
        ⟨alg⟩ (lamₒ ⋮ _) = refl
```

The uniqueness proof – that sem is equal to any meta-algebra homomorphism $g : \Lambda \rightharpoonup \mathcal{A}$ – involves the mutually inductive lemma 𝕊-lu and the inverse property of tabulate and lookup in the metavariable case; everything else follows from the homomorphism properties of g.

```
𝕊-lu : (ε : Sub Λ Π Γ)(v : 𝐼 α Π) → 𝕊 ε v ≡ g (lookup ε v)
𝕊-lu (t ◂ ε) new     = sem! t
𝕊-lu (t ◂ ε) (old v) = 𝕊-lu ε v

sem! : (t : Λ α Γ) → sem t ≡ g t
sem! (var v) = sym ⟨var⟩
sem! (mvar 𝔪 ε) rewrite 𝕊-lu ε = trans (sym ⟨mvar⟩) (cong (g ∘ mvar 𝔪) (tab∘lu≈id ε))
sem! (app f a)    rewrite sem! f | sem! a = sym ⟨alg⟩
sem! (lam b)      rewrite sem! b          = sym ⟨alg⟩
```

At the expense of minimal extra boilerplate, one is able to prove that the inductively defined family Λ is an initial meta-algebra, and instantiate our metatheory with this result. The Theory module associated with an initial meta-algebra exports every definition and lemma given in the framework for easy access; for example, the coveted single-variable substitution operation is directly accessible as Theory.[_/] : Λ α Γ → Λ β (α · Γ) → Λ β Γ, with no extra work required.

4.4 Code generation

The next logical step is eliminating the need to write boilerplate altogether and allow users to go from a direct specification of the second-order signature to the formalised metatheory of an explicitly encoded data type of terms automatically. Since the initiality proof for the explicit encoding is rather formulaic and much of it is signature-independent, one may as well generate the associated Agda code from a syntax description using a simple Python script. This signature-to-Agda compiler takes a simple textual specification of a type and term signature, and produces the Agda code for the Signature declaration and initiality proof outlined in Sections 4.1 and 4.3. For instance, one can give the signature of our running STLC example as follows (recall the introduction), also optionally specifying infix symbols and fixity information:

```
type                        term
  N   : 0-ary                 app : α ≻ β   α  →  β      | _$_  l20
  _≻_ : 2-ary | r30           lam : α.β        →  α ≻ β  | ƛ_   r10
```

Saving this in a file `stlc` and running `python soas.py stlc` produces the files `Signature.agda` and `Syntax.agda` with all the required imports, declarations of the signature Λ:Sig and explicit inductive data type of terms Λ, and the initiality proof. One is thereby free to proceed with the interesting parts of language formalisation like operational and denotational semantics right away.

The compiler is a fairly straightforward Python program that parses the specification language and produces formatted Agda files using the built-in string templating system. Its simplicity is one of its advantages: the boilerplate code it generates is systematic and minimal (1-2 lines of code for every type constructor and 6-7 lines for every term constructor), so manual testing on a wide range of examples gives us sufficient confidence in the robustness and correctness of the script. Unlike most other code generation solutions (some listed in Section 6.1) that produce the entire syntax-specific formalisation, including fragile and impenetrable de Bruijn arithmetic proofs, we leverage the fully syntax-generic metatheory implemented in the library and generate just enough code (namely the initiality proof, which is less boilerplate and more an elegant categorical argument) to instantiate it. Consequently, the compiler output is concise, readable, and easy to maintain.

Having examined the generic metatheory, construction of signatures, and the term syntax, we conclude with some demonstrations of the framework used in practice.

5 EXAMPLES

Our framework is flexible and unopinionated: it can be plugged into any intrinsically-typed formalisation of a second-order calculus and equip the syntax with the often-needed operations of weakening and substitution, and their corresponding laws. It also helps users in defining evaluation functions, interpreters, and syntactic translations in a concise and provably-correct manner. In this section, we give two extended examples of how the library may be used.

5.1 Computational calculi

The STLC has been our running example throughout the paper, and our framework is a great playground for experimenting with various extensions of it, be it with new types, terms, or equations. Below is a list of constructs that can be easily represented and compiled into Agda. The semantics of such a language would be rather complicated, but its syntax is still just a second-order signature.

```
type          term
  N   : 0-ary   app    : α ≻ β   α          →  β        pair : α     β                →  α ⊗ β
  _≻_ : 2-ary   lam    : α.β                →  α ≻ β    fst  : α ⊗ β                  →  α
  _⊗_ : 2-ary                                           snd  : α ⊗ β                  →  β
  _⊕_ : 2-ary   let    : α       α.β        →  β
  ¬_  : 1-ary   fix    : α ≻ α              →  α        inl  : α                      →  α ⊕ β
  T   : 1-ary                                           inr  : β                      →  α ⊕ β
                throw  : α       ¬ α        →  β        case : α ⊕ β   α.γ    β.γ     →  γ
                callcc : (¬ α).α            →  α
                                                        ze   :                        →  N
                return : α                  →  T α      su   : N                      →  N
                bind   : T α     α.(T β)    →  T β      nrec : N       α    (α,N).α   →  α
```

We will use the minimal fragment of STLC (with app and lam) to showcase the construction of models and interpretations. The CCC model of the STLC [Lambek 1980] in the category **Set** interprets types as sets, and terms $\Gamma \vdash t : \alpha$ as functions from the interpretation of Γ to the interpretation of α. Interpretation of contexts can be higher-order or first-order (as a Cartesian product of type interpretations) – with the higher-order encoding the model definition is remarkably concise.

```
⟦_⟧ : ΛT ⟶ Set
⟦ N ⟧      = ℕ
⟦ α ↣ β ⟧ = ⟦ α ⟧ → ⟦ β ⟧

⟦_⟧ᶜ : Ctx ⟶ Set
⟦ Γ ⟧ᶜ = ∀{α} → 𝓘 α Γ → ⟦ α ⟧

_⁺_ : ⟦ α ⟧ → ⟦ Γ ⟧ᶜ → ⟦ α · Γ ⟧ᶜ
(a ⁺ γ) new     = a
(a ⁺ γ) (old v) = γ v

Env : Familyₛ
Env α Γ = ⟦ Γ ⟧ᶜ → ⟦ α ⟧
```

```
Env^ΣM : ΣMon Env
Env^ΣM = record
  { ᴹ = record { η = λ v γ → γ v ; μ = λ t σ δ → t (λ v → σ v δ)
               ; lunit = refl ; runit = refl ; assoc = refl }
  ; alg     = λ { (appₒ ⋮ f , a) γ → f γ (a γ)
               ; (lamₒ ⋮ b)     γ → λ a → b (a ⁺ γ) }
  ; μ⟨alg⟩ = λ { (appₒ ⋮ _) → refl
               ; (lamₒ ⋮ b) → ext² λ δ a → cong b (dext
                    λ { new → refl ; (old v) → refl }) } }

module Env = FreeMonoid Ø Env^ΣM (λ ())

eval : Λ₀ ⇾ Env
eval = Env.𝕤𝕖𝕞
```

Here we restrict to the sorted family $\Lambda_0 = \Lambda\, \varnothing$ of λ-terms without metavariables for simplicity. The eval function interprets λ-terms as Agda programs; for example, eval (ƛ ƛ x_1) (λ ()) (where the 1st de Bruijn index var (old new) is denoted x_1) normalises to the Agda function $\lambda\, x\, y \to x$. Since it is derived by initiality, the interpretation is compositional and satisfies the semantic substitution lemma *by construction.* This is an enormous time-saver, since proving the soundness of substitution is often one of the most tedious steps required for the development of denotational semantics – the binding terms are the usual suspects, forcing one to reason about semantics of lifting, weakening, renaming, etc. Our framework does all the heavy lifting, allowing users to move on to less bureaucratic proofs. For example, after defining the predicate Val satisfied by value terms of the form ƛ b for $b : \Lambda_0\, \beta\, (\alpha \cdot \Gamma)$, it takes minimal effort to equip the language with an intrinsically-typed call-by-value reduction relation and a proof that it preserves the interpretation of terms:

```
data _⇝_ : Λ₀ α Γ ⟶ Λ₀ α Γ ⟶ Set where
  ζ-$₁ : {f g : Λ₀ (α ↣ β) Γ} {a   : Λ₀ α Γ}                    → f ⇝ g → f $ a   ⇝ g $ a
  ζ-$₂ : {f   : Λ₀ (α ↣ β) Γ} {a b : Λ₀ α Γ}          → Val f → a ⇝ b → f $ a   ⇝ f $ b
  β-ƛ  : {t   : Λ₀ α Γ}         {b   : Λ₀ β (α · Γ)} → Val t           → (ƛ b) $ t ⇝ [ t /] b

          sound : {t s : Λ₀ α Γ} → t ⇝ s → (γ : ⟦ Γ ⟧ᶜ) → eval t γ ≡ eval s γ
          sound (ζ-$₁ r)       γ rewrite sound r γ = refl
          sound (ζ-$₂ _ r)     γ rewrite sound r γ = refl
          sound (β-ƛ {t}{b} _) γ rewrite Env.sub-lemma t b
            = cong (eval b) (dext λ { new → refl ; (old v) → refl })
```

Note the use of the freely-obtained single-variable substitution [t /] b in the β-ƛ axiom, and the invocation of sub-lemma which translates eval ([t /] b) γ to Env.[eval t /] (eval s) γ. The STLC is of course a basic calculus, but implementing its syntax, operational and denotational semantics from scratch still involves a considerable amount of effort, mostly spent defining and reasoning about substitution. Leveraging our framework, this standard but relatively robust formalisation is possible in fewer than 100 lines of Agda code. We can also continue in the standard way with proofs like progress, determinacy, normalisation, etc. – if a syntactic property like associativity of substitution is required, it is likely to be present in the Theory module of the library already.

5.2 Partial differentiation

Another example of a second-order calculus is the axiomatisation of partial differentiation laid out by Plotkin [2020]. The syntax consists of the first-order theory of commutative rings with a second-order partial-differentiation operator $\mathrm{PDiff}(x.e\langle x\rangle, d)$, interpreted as the partial derivative of the expression $e\langle x\rangle$ with respect to x, evaluated at d (which has no free occurrences of x). This is usually denoted $\frac{\partial e\langle x\rangle}{\partial x}\big|_{x=d}$. To differentiate $e\langle x\rangle$ without evaluation, one renames x to a dummy variable w, differentiates $e\langle w\rangle$ with respect to w, then evaluates the result at $w = x$ – thus, the notation $\frac{\partial}{\partial x}e\langle x\rangle$ is taken as abbreviating $\frac{\partial e\langle w\rangle}{\partial w}\big|_{w=x}$.

The signature of rings augmented with the partial-differentiation operator can be readily expressed as an unsorted syntax description. We manually implement some derived operators in Agda, such as ∂_0 and ∂_1 (with symbols d0 and d1), respectively the partial derivatives w.r.t. the first and second variable. Note the use of weakening below, available from the Theory module.

```
term
  zero  :              * | 𝟘
  one   :              * | 𝟙
  inv   :  *      →    * | ⊖_   r50
  add   :  *   *  →    * | _⊕_  l20
  mult  :  *   *  →    * | _⊗_  l40
  pdiff : *.*  *  →    * | ∂_|_
```

```
∂₀_ : PD 𝔛 * (* · Γ) → PD 𝔛 * (* · Γ)
∂₀ e = ∂ (Theory.𝕨𝕜 𝔛 e) | x₀
∂₁_ : PD 𝔛 * (* · * · Γ) → PD 𝔛 * (* · * · Γ)
∂₁ e = ∂ (Theory.𝕨𝕜 𝔛 e) | x₁
```

The equational theory may also be included in the syntax description, listed both as explicit equations of the form ‘(name) metavars ▷ vars ⊢ expr$_1$ = expr$_2$’, and algebraic properties of operators. Plotkin’s use of function variables matches up with parametrised metavariables, so the axioms of the paper can be directly translated to our formalism. A few examples are given below.

```
theory
  ‘zero’ unit of ‘add’
  ‘mult’ distributes over ‘add’
  (∂⊕) a : *         ▷  x : *          ⊢  d0 (add (x,a))    = one
  (∂C) f : (*,*).*   ▷  x : *   y : *  ⊢  d1 (d0 (f[x,y])) = d0 (d1 (f[x,y]))
```

The generated Agda modules include the intrinsically-typed syntax of semirings with partial differentiation (as an inductive sorted family PD), and the generic equational reasoning framework of Section 3.7 instantiated with the data type $\_\triangleright\_\vdash\_\approx_A\_$ generated from the axiom descriptions. It employs some syntactic sugar for the purposes of readability and ease of use. Instead of defining a named inductive family of named metavariables for every axiom, we build up a context of metavariables in-place with the notation $\lceil \Pi_1 \Vdash \tau_1 \rceil\ \lceil \Pi_2 \Vdash \tau_2 \rceil \cdots \lceil \Pi_n \Vdash \tau_n \rceil$, and refer to the metavariables using (alphabetic) de Bruijn indices $\mathfrak{a}, \mathfrak{b}, \mathfrak{c}$. The environment for an n-ary metavariable $\mathfrak{m}$ is specified as $\mathfrak{m}\langle\ t_0 \blacktriangleleft \ldots \blacktriangleleft t_{n-1}\ \rangle$, or just $\mathfrak{m}$ if $n = 0$. Some examples are shown below:

```
data _▷_⊢_≈ₐ_ : (𝔛 : MCtx)(Γ : Ctx){α : *T} → PD 𝔛 α Γ → PD 𝔛 α Γ → Set where
  𝟘U⊕ᴸ : ⌈ * ⌉               ▷ ∅       ⊢ 𝟘 ⊕ 𝔞                ≈ₐ 𝔞
  ⊗D⊕ᴸ : ⌈ * ⌉ ⌈ * ⌉ ⌈ * ⌉   ▷ ∅       ⊢ 𝔞 ⊗ (𝔟 ⊕ 𝔠)          ≈ₐ (𝔞 ⊗ 𝔟) ⊕ (𝔞 ⊗ 𝔠)
  ∂⊕   : ⌈ * ⌉               ▷ ⌊ * ⌋   ⊢ ∂₀ (x₀ ⊕ 𝔞)          ≈ₐ 𝟙
  ∂⊗   : ⌈ * ⌉               ▷ ⌊ * ⌋   ⊢ ∂₀ (𝔞 ⊗ x₀)          ≈ₐ 𝔞
  ∂C   : ⌈ * · * ⊩ * ⌉       ▷ ⌊ * · * ⌋ ⊢ ∂₁ (∂₀ 𝔞⟨ x₀ ◂ x₁ ⟩) ≈ₐ ∂₀ (∂₁ 𝔞⟨ x₀ ◂ x₁ ⟩)
  ∂Ch₂ : ⌈ * · * ⊩ * ⌉ ⌈ * ⊩ * ⌉ ⌈ * ⊩ * ⌉ ▷ ⌊ * ⌋ ⊢
             ∂₀ 𝔞⟨ 𝔟⟨ x₀ ⟩ ◂ 𝔠⟨ x₀ ⟩ ⟩ ≈ₐ (∂ 𝔞⟨ x₀ ◂ 𝔠⟨ x₁ ⟩ ⟩ | 𝔟⟨ x₀ ⟩) ⊗ (∂₀ 𝔟⟨ x₀ ⟩) ⊕
                                        (∂ 𝔞⟨ 𝔟⟨ x₁ ⟩ ◂ x₀ ⟩ | 𝔠⟨ x₀ ⟩) ⊗ (∂₀ 𝔠⟨ x₀ ⟩)
```

The first two constructors correspond to the left unit and distributivity laws, with the right versions automatically derived via the commutativity of ⊕ and ⊗ (omitted). The four constructors involving differentiation correspond to the axioms of sums and products ($\frac{\partial x \oplus a}{\partial x} = 1$, $\frac{\partial ax}{\partial x} = a$), exchange of derivative operators ($\frac{\partial}{\partial y}\frac{\partial}{\partial x}f(x, y) = \frac{\partial}{\partial x}\frac{\partial}{\partial y}f(x, y)$), and the binary version of the chain rule. Note that these axioms relate open terms with one or two object variables, which must be listed in the context in addition to the metavariables.

We may now feed the axiom relation to the signature-generic equational logic and establish theorems via equational reasoning – be they properties of rings, derivable partial-differentiation laws, or explicit calculations of derivatives. The user can once again make use of some syntactic sugar to streamline the proofs. For a simple example, consider the corollary $\frac{\partial 0}{\partial x} = 0$, derivable from the left annihilation law of 𝟘 and ⊗, and the product differentiation axiom.

```
∂𝟘 : ⁅⁆ ▷ ⌊ * ⌋ ⊢ ∂₀ 𝟘 ≈ 𝟘
∂𝟘 = begin  ∂₀ 𝟘          ≈⟨ cong[ ax 𝟘X⊗ᴸ with 《 x₀ 》 ]in ∂₀ ◌ᵃ ⟩ₛ
            ∂₀ (𝟘 ⊗ x₀)   ≈⟨ ax ∂⊗ with 《 𝟘 》 ⟩
            𝟘             ■
```

The ax a with 《 t_1 ◂ . . . ◂ t_n 》 notation associates an axiom a with an instantiation of metavariables in the metavariable context of the axiom, given as a list of terms. Applications of an equation in a subexpression of t is done with the cong[e]in $t\langle\mathfrak{m}\rangle$ combinator, where $\mathfrak{m}$ is a new metavariable (denoted ◌ᵐ to make its role as a 'hole' clear) added to the context to indicate the location in which the equation e is applied. Thus, in the first step, we apply the left annihilation axiom $\mathbb{0} = \mathbb{0} \otimes x$ instantiated at $x = \mathsf{x}_0$ to the subexpression of the ∂_0 operator. As a more involved example, we derive the unary chain rule from the binary axiom (instantiated with $f(x, y) \triangleq f(x)$ and $h(x) = 0$), the ∂𝟘 corollary above, and the unit and annihilation laws for 𝟘:

```
∂Ch₁ : ⁅ * ⊩ * ⁆ ⁅ * ⊩ * ⁆ ⊢ ∂₀ 𝔞⟨ 𝔟⟨ x₀ ⟩ ⟩ ≈ (∂ 𝔞⟨ x₀ ⟩ ∣ 𝔟⟨ x₀ ⟩) ⊗ (∂₀ 𝔟⟨ x₀ ⟩)
∂Ch₁ = begin  ∂₀ 𝔞⟨ 𝔟⟨ x₀ ⟩ ⟩
          ≈⟨ ax ∂Ch₂ with 《 𝔞⟨ x₀ ⟩ ◂ 𝔟⟨ x₀ ⟩ ◂ 𝟘 》 ⟩
              (∂ 𝔞⟨ x₀ ⟩ ∣ 𝔟⟨ x₀ ⟩) ⊗ (∂₀ 𝔟⟨ x₀ ⟩) ⊕ (∂ 𝔞⟨ 𝔟⟨ x₁ ⟩ ⟩ ∣ 𝟘) ⊗ ∂₀ 𝟘
          ≈⟨ cong[ thm ∂𝟘 ]in [...] ⊕ (∂ 𝔞⟨ 𝔟⟨ x₁ ⟩ ⟩ ∣ 𝟘) ⊗ ◌ᶜ ⟩
              (∂ 𝔞⟨ x₀ ⟩ ∣ 𝔟⟨ x₀ ⟩) ⊗ (∂₀ 𝔟⟨ x₀ ⟩) ⊕ (∂ 𝔞⟨ 𝔟⟨ x₁ ⟩ ⟩ ∣ 𝟘) ⊗ 𝟘
          ≈⟨ cong[ thm 𝟘X⊗ᴿ with 《 ∂ 𝔞⟨ 𝔟⟨ x₁ ⟩ ⟩ ∣ 𝟘 》 ]in [...] ⊕ ◌ᶜ ⟩
              (∂ 𝔞⟨ x₀ ⟩ ∣ 𝔟⟨ x₀ ⟩) ⊗ (∂₀ 𝔟⟨ x₀ ⟩) ⊕ 𝟘
          ≈⟨ ax 𝟘U⊕ᴿ with 《 (∂ 𝔞⟨ x₀ ⟩ ∣ 𝔟⟨ x₀ ⟩) ⊗ ∂₀ 𝔟⟨ x₀ ⟩ 》 ⟩
              (∂ 𝔞⟨ x₀ ⟩ ∣ 𝔟⟨ x₀ ⟩) ⊗ (∂₀ 𝔟⟨ x₀ ⟩)      ■
```

We wrote [. . .] above for the sake of brevity – the context of the congruence needs to be written out explicitly for Agda to be able to evaluate the metasubstitution that instantiates the distinguished hole metavariable. Note also the use of thm, which uses an established (non-axiomatic) equality as a proof step. The precise and sufficiently general definition of metasubstitution ensures that we always have access to the right metavariables and object variables where we need them, making the construction of equational proofs quite intuitive.

6 CONCLUSION

We presented a language-formalisation framework that allows users to produce Agda implementations of second-order languages at the press of a button. The generated term language is explicitly represented as an inductive, intrinsically-encoded data type, and the formalised metatheory can be used as and where required: substitution for operational semantics, compositional interpretations for denotational semantics, metasubstitution for equational reasoning, etc. All the formalisms featured in the library naturally derive from the mathematical theory of abstract syntax, without the need for ad-hoc definitions or lemmas. The convenient code-generation script allows for rapid prototyping and experimentation, and it is easy to manually extend a formalised signature or incorporate an existing intrinsic term syntax into the framework.

6.1 Related work

The question of formalising and reasoning about abstract syntax was motivated by the development of proof assistants and the realisation that the Barendregt [1984] variable convention – *"rename variables as needed to avoid clashes"* – is difficult to translate into a formal setting. This has lead to a host of approaches that address the encoding of variable binding in proof assistants and functional languages, such as: higher-order abstract syntax [Pfenning and Elliot 1988; Chlipala 2008]; locally nameless representation [Bird and Paterson 1999; McBride and McKinna 2004; Weirich et al. 2011; Charguéraud 2012]; intrinsically-typed encoding [Benton et al. 2012; Allais et al. 2021; Érdi 2018]; and others [Shinwell et al. 2003; Urban and Kaliszyk 2011; Copello et al. 2017]. Similarly active is the mathematical study of abstract syntax and variable binding: developments include presheaf models [Fiore et al. 1999; Hofmann 1999]; nominal sets [Gabbay and Pitts 1999]; monadic approaches [Bellegarde and Hook 1994; Altenkirch and Reus 1999]; and others [Pigozzi and Salibra 1995; Sun 1999; Blanchette et al. 2019; Chen and Roşu 2020].

Benchmarks. The PoplMark challenge [Aydemir et al. 2005; Abel et al. 2019] sets out a collection of criteria according to which metatheory-formalisation efforts can be compared. Several submissions use Coq as the target language, in some cases involving code generation from a second-order signature [Aydemir et al. 2008; Vouillon 2011; Lee et al. 2012; Polonowski 2013; Keuchel et al. 2016; Stark et al. 2019]. Though impressive, the approaches rarely adopt intrinsically-typed encodings of variables and terms, usually opting for numeric de Bruijn indices with all their complicated and error-prone arithmetic. We believe that the nameless, intrinsic representation is hard to surpass in dependently-typed proof assistants thanks to its static guarantees on the typing and scoping of terms. Its drawbacks (boilerplate and types "getting in the way") are significant and form one of the motivations of our line of research, but they are ultimately not unreasonable: a rigorous pen-and-paper proof of the type-preservation of substitution would involve the same difficulties we encounter in defining it in Agda. Our approach also incorporates generic traversals, and – for the first time, as far as we are aware – equational logic with the aid of parametrised metavariables, all naturally derived from the syntax.

Type- and scope-safe syntax. Our work is closely aligned with that of AACMM [2021]: how to simplify the work of a language researcher by automating the boring metatheory. In their discussion of the presheaf model (loc. cit. Section 9.3), the authors suggest that freeing oneself from the formalities of the mathematics enables further progress in the development of the metatheory. We found very much the opposite: without the systematic view of the problem provided by the categorical model, one misses out on powerful principles that simplify proofs and untangle the conceptual labyrinth that is formal abstract syntax. A case in point is our reformulation of AACMM's core Semantics record as an instance of initial algebra semantics into the internal hom, which are very useful abstractions that the authors seemingly overlooked despite working from similar foundations. We also inherit the modularity of the categorical model by cleanly separating the mathematical groundwork, abstract metatheory, second-order signatures, and term representation; whereas AACMM's formalisation is very closely tied to their Desc data type which users are required to adopt to make use of the library. We do not have an analogue of their proof framework for simulations and fusions; but we suspect that equality-based properties (in addition to the fusion lemmas, which we already incorporate) can be captured through initiality, without reliance on another complex set of proof techniques.

Presheaf approach. The backdrop of our Agda formalisation is a comprehensive mathematical reformulation of the presheaf model of variable binding [Fiore et al. 1999] with parametrised metavariables [Fiore 2008]. Namely, we shift attention to an indexed model (in the category of families over the set of contexts, equivalently functors from the discrete category $|\mathbb{F}|$ to **Set**) equipped with a pair of canonical adjoint monadic and comonadic modalities ($\Diamond \dashv \Box$, induced by the inclusion of $|\mathbb{F}|$ in $\mathbb{F}$) and their respective algebras and coalgebras, which are equivalent to presheaves. Although in this paper we have chosen not to emphasise the abstract categorical development in favour of a discussion geared towards programming-language researchers, we stress that the alternative viewpoint is not a matter of taste but crucial to the practical formal development that we have put forward. For instance, a direct translation of the presheaf model leads to a formalisation that inevitably requires quotients (colimits and coends) and cannot be used for computation (such as pretty printing, because of the lack of canonical representatives).

Our work required a variety of new considerations. Definitions and proofs had to be recast in the setting of (i) skew-closed structure and (ii) initiality; along the way, new notions and techniques had to be developed to bypass quotienting. Concerning (i), moving onto families with skew-monoidal structure is not enough to avoid the need for quotienting (e.g. Borthelle et al. [2020] achieve the freeness proof in a skew-monoidal setting only with the aid of a general lemma of Fiore and Saville [2017, Theorem 4.8] that relies on the presentation of initial algebras as colimits of ω-chains, rather than as inductive data types); while, concerning (ii), the fact that initial algebras in presheaves can be lifted from initial algebras in families had to be given mathematical grounding. As an upshot, using families (indexed types), instead of presheaves, leads to a lightweight practical formalisation that suits the intrinsically-typed setting well.

6.2 Future work

We recognise that the formal systems studied in modern type theory go far beyond second-order ones with algebraic types; indeed, linear, dual-context, polymorphic, dependent, polarised, etc. calculi abound. The presheaf approach has been extended to several of these [Tanaka 2000; Fiore 2006; Hyland and Tasson 2020; Fiore and Hamana 2013; Fiore 2008] and we are of course interested in adapting and/or extending our framework to these and combinations thereof.

As hinted at above, our background work involved a categorical reformulation of the highly abstract presheaf model to the formalisation-friendly realm of sorted families and $\Box$-coalgebras; indeed, much of the Agda code has been directly read off categorical definitions and commutative diagrams. Work on the formalisation, particularly in relation to metasubstitution, is ongoing. We also anticipate interesting applications in the study of parametrised signatures and signature translations; for example, we can encode the second-order equational theory of first-order logic and modularly extend it with the relation and function symbols of any first-order signature. Further experiments with more complex languages and proofs (such as the ones given in the PoplMark challenge) will also inform and motivate the future development of our library.

ACKNOWLEDGMENTS

We thank the anonymous paper and artefact reviewers for their valuable suggestions and support. We are also grateful to Nathanael Arkor and Neel Krishnaswami for helpful discussions about technical and practical aspects of abstract syntax and language formalisation throughout the research.

REFERENCES

M. Abadi, L. Cardelli, P.-L. Curien, and J.-J. Levy. 1989. Explicit Substitutions. In *Proceedings of the 17th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL 1990.* ACM Press, 31–46. https://doi.org/10.1145/96709.96712

Andreas Abel. 2010. MiniAgda: Integrating Sized and Dependent Types. arXiv:1012.4896 [cs.PL] https://arxiv.org/abs/1012.4896

Andreas Abel, Guillaume Allais, Aliya Hameer, Brigitte Pientka, Alberto Momigliano, Steven Schäfer, and Kathrin Stark. 2019. PoplMark reloaded: Mechanizing proofs by logical relations. *Journal of Functional Programming* 29 (2019), e19. https://doi.org/10.1017/S0956796819000170

Peter Aczel. 1978. A General Church–Rosser Theorem. (1978). http://www.ens-lyon.fr/LIP/REWRITING/MISC/AGeneralChurch-RosserTheorem.pdf Unpublished note.

Guillaume Allais, Robert Atkey, James Chapman, Conor McBride, and James McKinna. 2021. A type- and scope-safe universe of syntaxes with binding: their semantics and proofs. *Journal of Functional Programming* 31 (2021), e22. https://doi.org/10.1017/S0956796820000076

Guillaume Allais, James Chapman, Conor McBride, and James McKinna. 2017. Type-and-Scope Safe Programs and Their Proofs. In *Proceedings of the 6th ACM SIGPLAN Conference on Certified Programs and Proofs (CPP 2017).* ACM Press, 195–207. https://doi.org/10.1145/3018610.3018613

Thorsten Altenkirch, Neil Ghani, Peter Hancock, Conor McBride, and Peter Morris. 2015. Indexed containers. *Journal of Functional Programming* 25 (2015), e5. https://doi.org/10.1017/S095679681500009X

Thorsten Altenkirch and Bernhard Reus. 1999. Monadic Presentations of Lambda Terms Using Generalized Inductive Types. In *Proceedings of the 13th International Workshop on Computer Science Logic (CSL 1999) (Lecture Notes in Computer Science (LNCS), Vol. 1683).* Springer, 453–468. https://doi.org/10.1007/3-540-48168-0_32

Nathanael Arkor and Marcelo Fiore. 2020. Algebraic Models of Simple Type Theories: A Polynomial Approach. In *Proceedings of the 35th Annual ACM/IEEE Symposium on Logic in Computer Science (LICS 2020) (LICS '20).* ACM Press, 88–101. https://doi.org/10.1145/3373718.3394771

Brian Aydemir, Aaron Bohannon, Matthew Fairbairn, J. Nathan Foster, Benjamin C. Pierce, Peter Sewell, Dimitrios Vytiniotis, Geoffrey Washburn, Stephanie Weirich, and Steve Zdancewic. 2005. Mechanized Metatheory for the Masses: The PoplMark Challenge. In *Proceedings of the 18th International Conference on Theorem Proving in Higher Order Logics (TPHOLs 2005).* Springer, 50–65. https://doi.org/10.1007/11541868_4

Brian Aydemir, Arthur Charguéraud, Benjamin C. Pierce, Randy Pollack, and Stephanie Weirich. 2008. Engineering Formal Metatheory. *ACM SIGPLAN Notices* 43, 1 (2008), 3–15. https://doi.org/10.1145/1328897.1328443

Henk P. Barendregt. 1984. *The Lambda Calculus: Its Syntax and Semantics.* Elsevier. https://doi.org/10.1016/c2009-0-14341-6

Françoise Bellegarde and James Hook. 1994. Substitution: A Formal Methods Case Study Using Monads and Transformations. *Science of Computer Programming* 23, 2-3 (1994), 287–311. https://doi.org/10.1016/0167-6423(94)00022-0

Nick Benton, Chung-Kil Hur, Andrew J Kennedy, and Conor McBride. 2012. Strongly typed term representations in Coq. *Journal of Automated Reasoning* 49, 2 (2012), 141–159. https://doi.org/10.1007/s10817-011-9219-0

Richard S. Bird and Ross Paterson. 1999. De Bruijn Notation as a Nested Datatype. *Journal of Functional Programming* 9, 1 (1999), 77–91. https://doi.org/10.1017/S0956796899003366

Jasmin Christian Blanchette, Lorenzo Gheri, Andrei Popescu, and Dmitriy Traytel. 2019. Bindings as Bounded Natural Functors. *Proceedings of the ACM on Programming Languages* 3, POPL (2019). https://doi.org/10.1145/3290335

Peio Borthelle, Tom Hirschowitz, and Ambroise Lafont. 2020. A Cellular Howe Theorem. In *Proceedings of the 35th Annual ACM/IEEE Symposium on Logic in Computer Science (LICS 2020).* ACM Press, 273–286. https://doi.org/10.1145/3373718.3394738

Arthur Charguéraud. 2012. The locally nameless representation. *Journal of Automated Reasoning* 49, 3 (2012), 363–408.

Xiaohong Chen and Grigore Roşu. 2020. A General Approach to Define Binders Using Matching Logic. *Proceedings of the ACM on Programming Languages* 4, ICFP (2020). https://doi.org/10.1145/3408970

Adam Chlipala. 2008. Parametric Higher-Order Abstract Syntax for Mechanized Semantics. In *Proceedings of the 13th ACM SIGPLAN International Conference on Functional Programming (ICFP '08).* Association for Computing Machinery, 143–156. https://doi.org/10.1145/1411204.1411226

Ernesto Copello, Nora Szasz, and Álvaro Tasistro. 2017. Formal metatheory of the Lambda calculus using Stoughton's substitution. *Theoretical Computer Science* 685 (2017), 65 – 82. https://doi.org/10.1016/j.tcs.2016.08.025

Brian Day. 1970. On closed categories of functors. In *Reports of the Midwest Category Seminar IV.* Springer, 1–38.

Gergő Érdi. 2018. Generic description of well-scoped, well-typed syntaxes. arXiv:1804.00119 [cs.PL] https://arxiv.org/abs/1804.00119

Marcelo Fiore. 2006. On the structure of substitution. https://www.cl.cam.ac.uk/~mpf23/talks/MFPS2006.pdf Talk at MFPS 2006.

Marcelo Fiore. 2008. Second-Order and Dependently-Sorted Abstract Syntax. In *Proceedings of the 23rd Annual ACM/IEEE Symposium on Logic in Computer Science (LICS 2008)*. IEEE Computer Society, 57–68. https://doi.org/10.1109/LICS.2008.38

Marcelo Fiore. 2012. Discrete Generalised Polynomial Functors. In *39th International Colloquium on Automata, Languages and Programming (ICALP 2012) (Lecture Notes in Computer Science, Vol. 7392)*. Springer, 214–226.

Marcelo Fiore and Makoto Hamana. 2013. Multiversal Polymorphic Algebraic Theories: Syntax, Semantics, Translations, and Equational Logic. In *Proceedings of the 28th Annual ACM/IEEE Symposium on Logic in Computer Science (LICS 2013)*. IEEE Computer Society, 520–529. https://doi.org/10.1109/LICS.2013.59

Marcelo Fiore and Chung-Kil Hur. 2010. Second-Order Equational Logic. In *Proceedings of the 24th International Workshop on Computer Science Logic (CSL 2010) (Lecture Notes in Computer Science (LNCS), Vol. 6247)*, Anuj Dawar and Helmut Veith (Eds.). Springer, 320–335. https://doi.org/10.1007/978-3-642-15205-4_26

Marcelo Fiore and Ola Mahmoud. 2010. Second-Order Algebraic Theories. In *Proceedings of the 35th International Symposium on Mathematical Foundations of Computer Science (MFCS 2010) (Lecture Notes in Computer Science (LNCS), Vol. 6281)*, Petr Hliněný and Antonín Kučera (Eds.). Springer, 368–380. https://doi.org/10.1007/978-3-642-15155-2_33

Marcelo Fiore, Gordon Plotkin, and Daniele Turi. 1999. Abstract Syntax and Variable Binding. In *Proceedings of the 14th Annual ACM/IEEE Symposium on Logic in Computer Science (LICS 1999)*. IEEE Computer Society, 193–202. https://doi.org/10.1109/LICS.1999.782615

Marcelo Fiore and Philip Saville. 2017. List Objects with Algebraic Structure. In *Proceedings of the 2nd International Conference on Formal Structures for Computation and Deduction (FSCD 2017) (Leibniz International Proceedings in Informatics (LIPIcs), Vol. 84)*, Dale Miller (Ed.). Schloss Dagstuhl–Leibniz-Zentrum für Informatik, 16:1–16:18. https://doi.org/10.4230/LIPIcs.FSCD.2017.16

Murdoch J. Gabbay and Andrew M. Pitts. 1999. A New Approach to Abstract Syntax Involving Binders. In *Proceedings of the 14th Annual ACM/IEEE Symposium on Logic in Computer Science (LICS 1999)*. IEEE Computer Society, 214–224. https://doi.org/10.1109/LICS.1999.782617

Joseph A. Goguen, James W. Thatcher, and E. G. Wagner. 1976. An Initial Algebra Approach to the Specification, Correctness and Implementation of Abstract Data Types. *IBM Research Report* 6487 (1976).

Makoto Hamana. 2004. Free Σ-Monoids: A Higher-Order Syntax with Metavariables. In *Proceedings of the Second Asian Symposium on Programming Languages and Systems (APLAS 2004)*, Wei-Ngan Chin (Ed.). Lecture Notes in Computer Science (LNCS), Vol. 3302. Springer, 348–363. https://doi.org/10.1007/978-3-540-30477-7_23

Martin Hofmann. 1999. Semantical Analysis of Higher-Order Abstract Syntax. *Proceedings of the 14th Annual ACM/IEEE Symposium on Logic in Computer Science (LICS 1999)*, 204–213. https://doi.org/10.1109/lics.1999.782616

Jason Hu and Jacques Carette. 2021. Formalizing category theory in Agda. In *Proceedings of the 10th ACM SIGPLAN International Conference on Certified Programs and Proofs*. 327–342.

Martin Hyland and Christine Tasson. 2020. The linear-non-linear substitution 2-monad. arXiv:2005.09559 [math.CT] https://arxiv.org/abs/2005.09559

Jonas Kaiser, Steven Schäfer, and Kathrin Stark. 2018. Binder Aware Recursion over Well-Scoped de Bruijn Syntax. In *Proceedings of the 7th ACM SIGPLAN Conference on Certified Programs and Proofs (CPP 2018)*. ACM Press, 293–306. https://doi.org/10.1145/3167098

Steven Keuchel, Stephanie Weirich, and Tom Schrijvers. 2016. Needle & Knot: Binder Boilerplate Tied Up. In *Programming Languages and Systems*, Peter Thiemann (Ed.). Springer Berlin Heidelberg, 419–445.

Anders Kock. 1971. Closed categories generated by commutative monads. *Journal of the Australian Mathematical Society* 12, 4 (1971), 405–424. https://doi.org/10.1017/S1446788700010272

Wen Kokke, Jeremy G. Siek, and Philip Wadler. 2020. Programming language foundations in Agda. *Science of Computer Programming* 194 (2020), 102440. https://doi.org/10.1016/j.scico.2020.102440

Joachim Lambek. 1980. From λ-calculus to cartesian closed categories. *To HB Curry: Essays on Combinatory Logic, Lambda Calculus and Formalism* (1980), 375–402.

Gyesik Lee, Bruno C. D. S. Oliveira, Sungkeun Cho, and Kwangkeun Yi. 2012. GMeta: A Generic Formal Metatheory Framework for First-Order Representations. In *Programming Languages and Systems*, Helmut Seidl (Ed.). Springer Berlin Heidelberg, 436–455.

The Coq Development Team. 2004. *The Coq proof assistant reference manual.* https://coq.inria.fr/distrib/current/files/CoqRefMan.pdf

Conor McBride. 2005. Type-preserving renaming and substitution. (2005). http://strictlypositive.org/ren-sub.pdf Unpublished note.

Conor McBride and James McKinna. 2004. Functional pearl: I am not a number – I am a free variable. In *Proceedings of the 2004 ACM SIGPLAN Workshop on Haskell*. 1–9.

Ulf Norell. 2009. Dependently Typed Programming in Agda. *Proceedings of the 4th International Workshop on Types in Language Design and Implementation (TLDI 2009)* (2009), 1–2. https://doi.org/10.1145/1481861.1481862

Frank Pfenning and Conal Elliot. 1988. Higher-order abstract syntax. *Proceedings of the 9th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI 1988)*, 199–208. https://doi.org/10.1145/960116.54010

Matthew Pickering, Gergő Érdi, Simon Peyton Jones, and Richard A. Eisenberg. 2016. Pattern Synonyms. In *Proceedings of the 9th International Symposium on Haskell (Haskell 2016)*. Association for Computing Machinery, 80–91. https://doi.org/10.1145/2976002.2976013

Don Pigozzi and Antonino Salibra. 1995. The abstract variable-binding calculus. *Studia Logica* 55, 1 (1995), 129–179. https://doi.org/10.1007/bf01053036

Andrew M. Pitts. 2019. *Initial algebra for a strictly positive endofunctor constructed using sized types and quotient types.* https://www.cl.cam.ac.uk/~amp12/agda/initial-T-algebras/

Gordon D. Plotkin. 2020. A Complete Equational Axiomatisation of Partial Differentiation. *Electronic Notes in Theoretical Computer Science* 352 (2020), 211 – 232. https://doi.org/j.entcs.2020.09.011

Emmanuel Polonowski. 2013. Automatically Generated Infrastructure for De Bruijn Syntaxes. In *Interactive Theorem Proving*, Sandrine Blazy, Christine Paulin-Mohring, and David Pichardie (Eds.). Springer Berlin Heidelberg, 402–417.

Mark R. Shinwell, Andrew M. Pitts, and Murdoch J. Gabbay. 2003. FreshML: Programming with Binders Made Simple. In *Proceedings of the 8th ACM SIGPLAN International Conference on Functional Programming (ICFP 2003)*. ACM Press, 263–274. https://doi.org/10.1145/944705.944729

Kathrin Stark, Steven Schäfer, and Jonas Kaiser. 2019. Autosubst 2: reasoning with multi-sorted de Bruijn terms and vector substitutions. In *Proceedings of the 8th ACM SIGPLAN International Conference on Certified Programs and Proofs*. 166–180.

Ross Street. 2013. Skew-closed categories. *Journal of Pure and Applied Algebra* 217, 6 (2013), 973 – 988. https://doi.org/10.1016/j.jpaa.2012.09.020

Yong Sun. 1999. An algebraic generalization of Frege structures – binding algebras. *Theoretical Computer Science* 211, 1 (1999), 189 – 232. https://doi.org/10.1016/S0304-3975(97)00170-9

Kornél Szlachányi. 2012. Skew-monoidal categories and bialgebroids. *Advances in Mathematics* 231, 3 (2012), 1694 – 1730. https://doi.org/10.1016/j.aim.2012.06.027

Miki Tanaka. 2000. Abstract Syntax and Variable Binding for Linear Binders. In *Proceedings of the 25th International Symposium on Mathematical Foundations of Computer Science (MFCS 2000) (Lecture Notes in Computer Science (LNCS), Vol. 1893)*, Mogens Nielsen and Branislav Rovan (Eds.). Springer, 670–679. https://doi.org/10.1007/3-540-44612-5_62

Christian Urban and Cezary Kaliszyk. 2011. General Bindings and Alpha-Equivalence in Nominal Isabelle. In *Programming Languages and Systems*, Gilles Barthe (Ed.). Springer Berlin Heidelberg, 480–500.

Vladimir Voevodsky, Benedikt Ahrens, Daniel Grayson, et al. 2014. *UniMath — a computer-checked library of univalent mathematics*. https://github.com/UniMath/UniMath

Jérôme Vouillon. 2011. A Solution to the PoplMark Challenge Based on de Bruijn Indices. *Journal of Automated Reasoning* 49 (2011), 327–362.

Stephanie Weirich, Brent A. Yorgey, and Tim Sheard. 2011. Binders Unbound. In *Proceedings of the 16th ACM SIGPLAN International Conference on Functional Programming (ICFP '11)*. Association for Computing Machinery, 333–345. https://doi.org/10.1145/2034773.2034818