Normalization by evaluation for a statically typed language

Owen Lynch

May 7, 2026

1 Introduction

2 Theory

The meta-theory of hippogriff is a second-order generalized algebraic theory equipped with a phase distinction. In this section, we recall the definition of second-order generalized algebraic theory, and say what it means for a second-order generalized algebraic theory to have a phase distinction.

2.1 Presheaves and natural models

In this section, we give an overview of the modern approach to the syntax of dependent type theory via presheaf categories. The reader unfamiliar with second order generalized algebraic theory will not find a comprehensive account here (give references), however the reader unfamiliar with SOGATs may find an account here that is sufficient to read the rest of this paper, depending on their overall familiarity with dependent type theory and category theory.

Categorical semantics for a type theory are typically given by characterization of the category of contexts and substitutions in that type theory.[1] In such a setting, a type in context Γ is identified with the weakening substitution $\Gamma.A \to \Gamma$. More generally, the category of contexts and substitutions is equipped with a class of maps called "display maps" and the weakening substitution $\Gamma.A \to \Gamma$ is always in this class; then various type theoretic constructions can be phrased in terms of constructions involving the class of display maps [9].

For example, Σ-types are characterized by saying that display maps are closed under composition. Unpacking this, this says that $\Gamma.A.B \to \Gamma$ is equivalent to a weakening substitution. Namely, it is equivalent to $\Gamma.(\Sigma\, A\, B) \to \Gamma$.

Unfortunately, there is a problem in only looking at the category of contexts and substitutions. In category theory, the notion of equality of objects is badly behaved with respect to many categorical constructions. For instance, category theorists often say "the" pullback, but the universal property of pullback only determines an object up to to natural isomorphism. Thus, it is generally best to avoid definitions which involve equality of objects.

[1] See (Jon's paper on LCC as LF) for alternative.

This seems to be incompatible with the fact that the notion of *definitional equality of types* is an essential concept; the important theorems that one would wish to prove about a type theory depend crucially on the difference between definitional equality and mere isomorphism, whereas if types in a context Γ are merely the display maps into Γ then it seems like category theory has nothing to say about their equality, only their isomorphisms.

We can remedy this in the following naive way: define a presheaf $\mathsf{Ty}\colon C^{\mathrm{op}} \to \mathsf{Set}$ such that each element of $\mathsf{Ty}(\Gamma)$ corresponds to an object of C/Γ (see [3] for an early account of this approach). Then we interpret types in context Γ as elements of $\mathsf{Ty}(\Gamma)$, and it is no problem at all to inquire as to the equality of two elements of a presheaf, so we can use this equality to interpret the definitional equality of the type theory of interest.

But what does it mean to identify the elements of $\mathsf{Ty}(\Gamma)$ with the objects of C/Γ? Clearly we cannot give a bijection between them, as that would defeat the entire point of the exercise, which was to get away from asking after the equality of objects. What we can do, however, is for each $A\colon \mathsf{Ty}(\Gamma)$, give the set of sections $\mathsf{Tm}(\Gamma, A)$ for the display map $\Gamma.A \to \Gamma$. By Yoneda, this actually determines the display map $\Gamma.A \to \Gamma$ *up to unique isomorphism*, without assuming that we have actually picked out a specific $\Gamma.A \to \Gamma$.

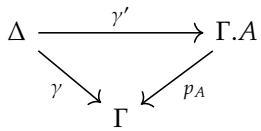
Definition 2.1 (Presheaf family)**.** An **presheaf family** over a presheaf $\mathsf{Ty}\colon C^{\mathrm{op}} \to \mathsf{Set}$ is a presheaf $\mathsf{Tm}\colon \left(\int \mathsf{Ty}\right)^{\mathrm{op}} \to \mathsf{Set}$.

Definition 2.2 (Representable presheaf family)**.** A presheaf family $\mathsf{Tm}\colon \left(\int \mathsf{Ty}\right)^{\mathrm{op}} \to \mathsf{Set}$ is **representable** if for each $(\Gamma, A)\colon \int \mathsf{Ty}$, the presheaf $\mathsf{Tm}_A\colon (C/\Gamma)^{\mathrm{op}} \to \mathsf{Set}$ defined by

$$\mathsf{Tm}_A(\gamma\colon \Delta \to \Gamma) = \mathsf{Tm}(\Delta, \gamma^*(A))$$

is representable.

If Tm is a representable presheaf family, we identify "the" arrow $p_A\colon \Gamma.A \to \Gamma$ with any of the representing objects for Tm_A. Then the idea behind Definition 2.2 is that the substitutions $\gamma'\colon \Delta \to \Gamma.A$ such that



commutes are in bijection with terms of type $\gamma^*(A)$ in context Δ.

Definition 2.3. A category equipped with a representable presheaf family is often called a **natural model** (a slight variation is due to [1]), a category equipped with a *represented* presheaf family is called a **category with families** [3]. The distinction between the two can be important when doing proofs, but for the conceptual overview that we are giving now we will not worry about it.

We can then say interesting things about the category C by "working internally" to presheaves on C and keeping track of which presheaf families are representable.

More specifically, we can work in a type theory where a context $\mathbf{\Gamma}$ is interpreted as a presheaf on C. We use a bold font to distinguish contexts in the metatheory (presheaves on C) from contexts in the theory (objects of C). Then we have two type judgments. The first one, $\mathbf{\Gamma} \vdash \mathbf{A}\ \mathsf{jdg}$ means that $\mathbf{A}$ is an presheaf family over $\mathbf{\Gamma}$. The second one, $\mathbf{\Gamma} \vdash \mathbf{A}\ \mathsf{jdg}^+$ means that $\mathbf{A}$ is a representable presheaf family over $\mathbf{\Gamma}$. In either case, a term judgment $\mathbf{\Gamma} \vdash \mathbf{a}\colon \mathbf{A}$ means that $\mathbf{a}$ is a *section* of $\mathbf{A}$.

Definition 2.4 (Section of presheaf family)**.** If Q is an presheaf family over P, a **section** of Q is a natural map $(x : P(C)) \to Q(C, x)$.

Then, so that we can work fully in the metatheory, we let Jdg and Jdg^+ be universes that decode into jdg and jdg^+ judgments, respectively; this requires some size subtleties but we will elide them for now. These are both "top-level" types, in other words we have a judgment $\mathbf{\Gamma} \vdash \mathbf{A}\ \mathsf{top}$, and $\vdash \mathsf{Jdg}\ \mathsf{top}$ and $\vdash \mathsf{Jdg}^+\ \mathsf{top}$. We also internalize the judgment form via the following Π-type constructor:

$$\frac{\mathbf{\Gamma} \vdash \mathbf{A}\ \mathsf{jdg} \qquad \mathbf{\Gamma}, x : \mathbf{A} \vdash \mathbf{B}\ \mathsf{top}}{\mathbf{\Gamma} \vdash (x : \mathbf{A}) \to \mathbf{B}\ \mathsf{top}}$$

The basic setup for a typical dependent type theory would then be given by the signature

$$\begin{aligned}&\mathsf{Ty} : \mathsf{Jdg}\\ &\mathsf{Tm} : \mathsf{Ty} \to \mathsf{Jdg}^+\end{aligned}$$

In other words, the "set of inner types" becomes a *universe* in the metatheory with *representable decoding*.

Finally, we have a rule in our type theory which allows taking Π-types with representable domains (we omit the introduction, elimination, computation and uniqueness rules as they are entirely standard).

$$\frac{\mathbf{\Gamma} \vdash \mathbf{A}\ \mathsf{jdg}^+ \qquad \mathbf{\Gamma}, x : \mathbf{A} \vdash \mathbf{B}\ \mathsf{jdg}}{\mathbf{\Gamma} \vdash (x : \mathbf{A}) \to \mathbf{B}\ \mathsf{jdg}}$$

Intuitively, this captures the idea that representable types are judgments that may appear in contexts.

This allows us to, for instance, postulate Σ-types in $(\mathsf{Ty}, \mathsf{Tm})$ via postulating the following constants

$$\begin{aligned}&\Sigma : (A : \mathsf{Ty}) \to ((x : \mathsf{Tm}\, A) \to \mathsf{Ty}) \to \mathsf{Ty}\\ &\textbf{variables} \quad A : \mathsf{Ty}, B : (x : \mathsf{Tm}\, A) \to \mathsf{Ty}\\ &\mathsf{pair} : (a : \mathsf{Tm}\, A) \to \mathsf{Tm}\,(B\, a) \to \mathsf{Tm}\,(\Sigma\, A\, B)\\ &\mathsf{proj}_0 : \mathsf{Tm}\,(\Sigma\, A\, B) \to \mathsf{Tm}\, A\\ &\mathsf{proj}_1 : (p : \mathsf{Tm}\,(\Sigma\, A\, B)) \to \mathsf{Tm}\,(B\,(\mathsf{proj}_0 p))\\ &\Sigma/\beta_0 : (a : \mathsf{Tm}\, A) \to (b : \mathsf{Tm}\,(B\, a)) \to \mathsf{proj}_0(\mathsf{pair}\, a\, b) = a\\ &\Sigma/\beta_1 : (a : \mathsf{Tm}\, A) \to (b : \mathsf{Tm}\,(B\, a)) \to \mathsf{proj}_1(\mathsf{pair}\, a\, b) = b\\ &\Sigma/\eta : \mathsf{Tm}\,(\Sigma\, A\, B) \to \mathsf{pair}\,(\mathsf{proj}_0\, p)\,(\mathsf{proj}_1\, p) = p\end{aligned}$$

Crucial to this is the ability to say that "B is a type in the context of a postulated term of type A"; the Π type allows us to do this.

Note that $(\mathsf{pair}, \mathsf{proj}_0, \mathsf{proj}_1 \Sigma/\beta_0, \Sigma/\beta_1, \Sigma/\eta)$ form the components of an isomorphism

$$\mathsf{Tm}\,(\Sigma\, A\, B) \cong (x : \mathsf{Tm}\, A) \times \mathsf{Tm}\,(B\, a)$$

In later developments, we will describe the universal properties of various type constructors in this succinct manner.

Ultimately, these postulated judgments are translated into statements about representable presheaves on C, and consequently into statements about the existence and uniqueness-up-to-isomorphism of certain objects in the various slice categories associated to C.

The advantage of working internally to presheaves is that naturality with respect to substitutions is automatically obtained. Classically, type theorists set up their theory carefully so that substitution is *admissible* rather than a primitive notion, and then rules like $(A \times B)[\gamma] \equiv A[\gamma] \times B[\gamma]$ are implicitly a consequence of how substitution is defined. However, admissibility of substitution is a property of the *inductive definition of terms*. When considering a general model of a type theory, substitution must be assumed as a primitive, and thus its interaction with type formers must be explicitly postulated as well. Working internally to presheaves permits omission of tedious boilerplate around substitution by assuming well-behaved function types in the metatheory.

Definition 2.5 (Second-order generalized algebraic theories, informally)**.** A second-order generalized algebraic theory is a collection of postulated constants in the minimal dependent type theory involving the stuff that we needed to define Σ-types earlier. A model of a second-order generalized algebraic theory consists of a category C with an interpretation of each of the constants in $\mathsf{Psh}(C)$.

A great deal of work has recently gone into second-order generalized algebraic theories [5, 2], allowing many theorems in type theory to be proved in a cleaner, more general way than was previously possible.

For the purposes of this paper, however, the intuition that second order generalized algebraic theories are "formulating a type theory by working internally to presheaves on contexts, paying careful attention to which presheaves are representable" will mostly suffice.

2.2 Phase distinctions and modalities

A key property of a type theory that is meant to be implemented is for definitional equality of types to be decidable (ideally, efficiently decidable). In a dependent type theory, this is achieved via a normalization result; all terms have dedicable definitional equality so in particular types have decidable definitional equality.

In simple type theory, however, the absence of strong normalization for general terms is not an obstruction to decidable definitional equality of types because types do not depend on terms. This permits use of, for instance, unrestricted recursion via the fixpoint operation $\mathsf{fix} : (A \to A) \to A$.

Parametric type theory, such as System F, falls in an intermediate point between dependent and simple. A type theoretic presentation of System F uses a split context $\Gamma_0 | \Gamma_1$, and types may depend on variables in Γ_0 but not those in Γ_1 [4]. Then problematic term constructors such as fixpoint are restricted to terms of types that appear in Γ_1, and thus do not interfere with decidability of equality for types. This provides facilities for abstractions ("generics"), while avoiding the complications of full dependent types; it is no accident that most modern statically typed programming languages fall under the description of "parametric type theories" in one way or another (Haskell, OCaml, Rust, etc.).

However, the natural unit of abstraction for programming often binds together types and terms. For instance, consider the following module type.

```
module type Eq = sig
```

```
  type t
  val eq : t -> t -> t
end
```

The ability in an ML-style module system to work abstractly on an element `M : Eq` is a key feature enabling code reuse and modularity. However, if we are to take seriously the notion that `Eq` should be some sort of type, then a split context presentation should not permit us to form `Eq`. Intuitively, given `M N : Eq`, type equality can certainly depend on the difference between `M.t` and `N.t`, so `Eq` should be a type in Γ_0, but type equality should not depend on deciding whether `M.eq` and `N.eq` are equal, so `Eq` cannot be a type in Γ_0.

The difficulties in maintaining the delicate balance between the parts of a module signature that are permissible to consider in the course of deciding type equality and those parts which must be excluded from type equality is arguably one of the reasons why ML-style module systems have not seen much uptake outside of ML-family languages.

Classically, the name for this balance is the "static/dynamic phase distinction." Intuitively, terms have a "static part" which is relevant for type equality, and a "dynamic part" which is irrelevant for type equality; any language features which are problematic for normalization (such as unrestricted recursion) are then restricted to the "dynamic fragment" of the language.

Sterling and Harper have given a particularly elegant synthetic account of the phase distinction in terms of the open and closed modalities associated with a proposition. In this section, we give an explicit description of the connection between this synthetic account and the classical theory of System F; this description is found in a slightly different style in Logical Relations as Types [8].

We start with a re-establishment of categorical semantics for System F. As a split-context type theory, the basic object of study is an indexed category $\mathbb{C}_1 : C_0^{\mathrm{op}} \to \mathsf{Cat}$; C_0 is the category of outer contexts and substitutions, and $\mathbb{C}_1(\Gamma_0)$ is the category of inner contexts and substitutions in an outer context Γ_0. We will refer to the corresponding fibration as $c : C_1 \to C_0$.

The base category C_0 is a simple type theory. Specifically, we have a representable presheaf family $\mathsf{Tm}_0 : \left(\int \mathsf{Ty}_0\right)^{\mathrm{op}} \to \mathsf{Set}$ such that Ty_0 is a constant presheaf on C_0. This means that the set of types in each context is always the same. In System F we have $\mathsf{Ty}_0(\Gamma_0) = \{\star\}$ for each $\Gamma_0 : C_0$, and $C_0 \cong \mathsf{FinSet}^{\mathrm{op}}$, the free cartesian category on a single object. System F_ω instead makes C_0 the free cartesian closed category on a single object.

Then, indexed over $\Gamma_0 : C_0$, $\mathbb{C}_1(\Gamma_0)$ is also a simple type theory. That is, we have a presheaf $\mathsf{Ty}_1 : C_1 \to \mathsf{Set}$ with a representable presheaf family $\mathsf{Tm}_1 : \left(\int \mathsf{Ty}_1\right)^{\mathrm{op}} \to \mathsf{Set}$ such that Ty_1 factors through $c : C_1 \to C_0$, and such that the weakening substitution for a type $A : \mathsf{Ty}_1(\Gamma_0, \Gamma)$ is given by a vertical map in the fibration $p_A : (\Gamma_0, \Gamma_1.A) \to (\Gamma_0, \Gamma_1)$.

In System F, it happens that $\mathsf{Ty}_1(\Gamma_0, \Gamma_1) = \mathsf{Tm}_0(\Gamma_0, \star)$, and so in a traditional presentation Tm_0 and Ty_1 are often confused. However, this should really be thought of as the universal property of $\star$ rather than as the definition of Ty_1.

The classical definition for a module type A in context (Γ_0, Γ_1) involves a static part $A_0 : \mathsf{Ty}_0(\Gamma_0)$ and a dynamic part $A_1 : \mathsf{Ty}_1(\Gamma_0.A_0, \Gamma_1)$, which by simplicity is equivalent to $A_0 : \mathsf{Ty}_0(\cdot)$ and $A_1 : \mathsf{Ty}_1(\Gamma_0.A_0, \cdot)$ where $\cdot$ is the empty context. We capture this via working in the category of *indexed presheaves* over $\mathbb{C}_1$.

Definition 2.6 (Indexed presheaf)**.** An indexed presheaf P over an indexed category $\mathbb{C}_1 : C_0^{\mathrm{op}} \to \mathsf{Cat}$ consists of a presheaf $P_0 : C_0^{\mathrm{op}} \to \mathsf{Set}$ along with a presheaf family $P_1 : \left(\int c \circ P_0\right)^{\mathrm{op}} \to \mathsf{Set}$

over $c \circ P_0 : C_1^{\mathrm{op}} \to \mathsf{Set}$.

More concretely, if P is an indexed presheaf then the type signature for P_1 is

$$P_1 : ((\Gamma_0 : C_0) \times \mathbb{C}_1(\Gamma_0) \times P_0(\Gamma_0))^{\mathrm{op}} \to \mathsf{Set}.$$

Definition 2.7. If $C = (C_0, \mathbb{C}_1)$ is an indexed category, then let $\mathsf{IPsh}(C)$ be the category of indexed presheaves over C.

For those that like topos theory, the category $\mathsf{IPsh}(C)$ is equivalent to the Artin gluing of the geometric morphism $c^* : \mathsf{Psh}(C_0) \to \mathsf{Psh}(C_1)$.

Example 2.8 (Indexed yoneda embedding)**.** Given $\Gamma_0 : C_0, \Gamma_1 : \mathbb{C}_1(\Gamma_0)$, define the indexed presheaf $\text{ɟ}(\Gamma_0, \Gamma_1)$ by

$$\text{ɟ}(\Gamma_0, \Gamma_1)_0 = \text{ɟ}\Gamma_0$$
$$\text{ɟ}(\Gamma_0, \Gamma_1)_1(\Delta_0, \Delta_1, \gamma_0 : \Delta_0 \to \Gamma_0) = \mathbb{C}_1(\Delta_0)(\Delta_1, \gamma_0^*(\Gamma_1))$$

Definition 2.9. An indexed presheaf is representable if it is isomorphism to an indexed presheaf of the form $\text{ɟ}(\Gamma_0, \Gamma_1)$ for some (Γ_0, Γ_1).

Definition 2.10 (Signatures and modules from a split-context type theory)**.** If $\mathsf{Ty}_0, \mathsf{Tm}_0, \mathsf{Ty}_1, \mathsf{Tm}_1$ are as before, then we can form an indexed presheaf Sig (module signatures) defined by

$$\mathsf{Sig}_0(\Gamma_0) = \mathsf{Ty}_0(\Gamma_0)$$
$$\mathsf{Sig}_1(\Gamma_0, \Gamma_1, A_0) = \mathsf{Ty}_1(\Gamma_0.A_0, \Gamma)$$

and then an *indexed presheaf family* Mod (modules) over Sig defined by

$$\mathsf{Mod}_0(\Gamma_0, A_0) = \mathsf{Tm}_0(\Gamma_0, A_0)$$
$$\mathsf{Mod}_1(\Gamma_0, \Gamma_1, A_0, A_1 : \mathsf{Ty}_1(\Gamma_0.A_0, \Gamma_1), M_0 : \mathsf{Tm}_0(\Gamma_0, A)) = \mathsf{Tm}_1(\Gamma_0, \Gamma_1, A_1[M_0])$$

where $A_1[M_0] : \mathsf{Ty}_1(\Gamma_0, \Gamma_1)$ is A_1 pulled back by the substitution $\Gamma_0 \to \Gamma_0.A_0$ given by adjoining M_0 to the identity substitution.

The reader with do-it-yourself spirit should stop at this point and work out what the appropriate definition of indexed presheaf family should be; we give a definition later to check against. The frustration that the reader feels at this exercise should instill the appropriate dislike for the classical approach to modules.

Intuitively, indexed presheaves capture the classical notion of a module, which is that a module signature has a static part and a dynamic part, and the static part depends on the dynamic part. In general, an indexed presheaf P is "something that has a static part P_0 which only depends on the static context and then a dynamic part P_1 which depends on the whole context and also on the static part."

Fortunately, there is a better way of doing modules than always explicitly dealing with static and dynamic parts. Just as we can avoid irritating side-conditions about naturality of substitutions by working internally to the category of presheaves on a category of contexts and substitutions, we can also avoid drowning in a sea of functors with five arguments by working internally to the category of indexed presheaves on the indexed category of contexts and substitutions.

Just as the key ingredient to doing this for presheaves is an appropriate notion of a representable presheaf family, the key ingredient for indexed presheaves is an appropriate notion of representable indexed presheaf family.

Definition 2.11 (Indexed presheaf family)**.** Let P be an indexed presheaf on $\mathcal{C} = (\mathcal{C}_0, \mathbb{C}_1)$. Then an **indexed presheaf family** Q over P consists of a presheaf family $Q_0 \colon \left(\int P_0\right)^{\mathrm{op}} \to \mathsf{Set}$ along with a functor

$$Q_1 \colon ((X : \mathcal{C}_0) \times (Y : \mathbb{C}_1(X)) \times (i : P_0(X)) \times Q_0(X, i) \times P_1(X, Y, i))^{\mathrm{op}} \to \mathsf{Set}$$

where we have used dependent product notation in order to express a fairly complex Grothendieck construction.

In order to translate Definition 2.2 into the indexed setting, we must first have a notion of slice for an indexed category.

Definition 2.12 (Indexed slice)**.** Suppose that $X_0 : \mathcal{C}_0, X_1 : \mathbb{C}_1(X)$. Then the indexed slice category $\mathcal{C}/(X_0, X_1)$ is defined by

$$(\mathcal{C}/(X_0, X_1))_0 = \mathcal{C}_0/X$$
$$(\mathcal{C}/(X_0, X_1))_1(f : Y_0 \to X_0) = \mathbb{C}_1(Y_0)/f^*(X_1)$$

Definition 2.13 (Representable indexed presheaf family)**.** If Sig is an indexed presheaf and Mod is an indexed presheaf family over Sig, then Mod is **representable** if for any

$$\Gamma_0 : \mathcal{C}_0, \Gamma_1 : \mathbb{C}_1(\Gamma_0), A_0 : \mathsf{Sig}_0(\Gamma_0), A_1 : \mathsf{Sig}_1(\Gamma_0, \Gamma_1, A_0)$$

the indexed presheaf Mod_{A_0,A_1} on $\mathcal{C}/(\Gamma_0, \Gamma_1)$ defined by

$$\mathsf{Mod}_{A_0,A_1,0}(\gamma_0 : \Delta_0 \to \Gamma_0) = \mathsf{Mod}_0(\Delta_0, \gamma_0^*(A_0))$$
$$\mathsf{Mod}_{A_0,A_1,1}(\gamma_0 : \Delta_0 \to \Gamma_0, \gamma_1 : \Delta_1 \to \gamma_0^*(\Gamma_1), a_0 : \mathsf{Mod}_0(\Delta_0, \gamma_0^*(A_0))) =$$
$$\quad \mathsf{Mod}_1(\Delta_0, \Delta_1, \gamma_0^*(A_0), a_0, (\gamma_0, \gamma_1)^*(A_1))$$

is representable in the sense of Definition 2.9.

Unpacking Definitions 2.9 and 2.12, we see that a representing pair of objects consists precisely of morphisms $p_{A_0} : \Gamma_0.A_0 \to \Gamma_0$ and $p_{A_1} : p_{A_0}^*(\Gamma_1).A_1 \to p_{A_0}^*(\Gamma_1)$.

Theorem 2.14. *As defined in Definition 2.10,* Mod *forms a representable indexed presheaf over* Sig*.*

Proof. It is straightforward to use the base and fiberwise representableness of Tm_0 and Tm_1 to establish this. □

If Definition 2.10 did not already convince the reader of the tedium involved in the indexed approach, proving Theorem 2.14 surely has. Fortunately, now that we have a notion of representable indexed presheaf family we can switch to working internally to indexed presheaves, with

$$\mathsf{Sig} : \mathsf{Jdg}$$
$$\mathsf{Mod} : \mathsf{Sig} \to \mathsf{Jdg}^+$$

just as before.

However, once we are working internally, we need an synthetic way of accessing the static and dynamic parts of a module signature. This can be accomplished incredibly simply, by just adding a single proposition to our metatheory, which is interpreted as an indexed presheaf in the following way.

Definition 2.15. The **static open** $\fatbslash\!\!\!\bigcirc_{\mathsf{st}}$ in a category of indexed presheaves is defined by

$$(\fatbslash\!\!\!\bigcirc_{\mathsf{st}})_0(\Gamma_0) = \top$$
$$(\fatbslash\!\!\!\bigcirc_{\mathsf{st}})_1(\Gamma_0, \Gamma_1, \_) = \bot$$

where $\top$ is the unit type and $\bot$ is the empty type.

If we start from System F, then we should have $\mathsf{Type} : \mathsf{Sig}$, the universe of types. The static part of Type is $\star : \mathsf{Ty}_0(\cdot)$, the type universe, and the dynamic part is just the unit type.[2] We can access the fact that the dynamic part is trivial via adding synthetically the axiom

$$\mathsf{Type\text{-}is\text{-}static} : (A\, A' : \mathsf{Mod}\,\mathsf{Type}) \to (\fatbslash\!\!\!\bigcirc_{\mathsf{st}} \to (A = A')) \to A = A'$$

That is, if two types are equal when postulating the static open, they must already be equal.

Conversely, if Int is an element of $\mathsf{Ty}_1(\cdot, \cdot)$, we also can have $\mathsf{Int} : \mathsf{Sig}$, where the static part of Int is the unit. To access synthetically the fact that a value of type Int has no static relevance, we add the axioms

$$\mathsf{Int\text{-}is\text{-}dynamic} : \fatbslash\!\!\!\bigcirc_{\mathsf{st}} \to \mathsf{IsContr}\,(\mathsf{Mod}\,\mathsf{Int})$$

where

$$\mathsf{IsContr}\, A := (a : A) \times ((a' : A) \to a = a')$$

We express this by saying that Int is *statically contractible*.

Now, there is actually a slight technical problem with $\mathsf{Type\text{-}is\text{-}static}$. Specifically, we exponentiated $A = A'$ by $\fatbslash\!\!\!\bigcirc_{\mathsf{st}}$, but we only have $\fatbslash\!\!\!\bigcirc_{\mathsf{st}} : \mathsf{Jdg}$, not $\fatbslash\!\!\!\bigcirc_{\mathsf{st}} : \mathsf{Jdg}^+$. And this is important, because we do not wish $\fatbslash\!\!\!\bigcirc_{\mathsf{st}}$ to be representable (this is impossible because a representable indexed presheaf cannot have uniformly empty fiber).

This problem may be resolved by adding $\fatbslash\!\!\!\bigcirc_{\mathsf{st}} \to -$ directly as a modality $\mathsf{Jdg} \to \mathsf{Jdg}$; we call this operation $\bigcirc_{\mathsf{st}}$. The semantics of $\bigcirc_{\mathsf{st}}$ in a category of indexed presheaves is precisely $\fatbslash\!\!\!\bigcirc_{\mathsf{st}} \to -$, but more explicitly

$$\bigcirc_{\mathsf{st}}(P)_0 = P_0$$
$$\bigcirc_{\mathsf{st}}(P)_1 = \mathsf{const}\,\top$$

Using $\bigcirc_{\mathsf{st}}$, we can instead write

$$\mathsf{Type\text{-}is\text{-}static} : (A\, A' : \mathsf{Mod}\,\mathsf{Type}) \to \bigcirc_{\mathsf{st}}(A = A') \to A = A'.$$

We can be more precise, however. $\bigcirc_{\mathsf{st}}$ describes a *reflexive sub-universe* of Jdg, which means that $\bigcirc_{\mathsf{st}}$ is an idempotent monadic modality that preserves limits (Σ-types and equality types). We can then make the following definition

Definition 2.16 (Staticness of a judgment)**.** Let $\mathsf{IsStatic}$ be defined in the following way.

$$\mathsf{IsStatic} : \mathsf{Jdg} \to \mathsf{Prop}$$
$$\mathsf{IsStatic}\, A := \mathsf{IsEquiv}\, \eta_A^{\bigcirc_{\mathsf{st}}}$$

Intuitively, P is static if $P_1 \cong \mathsf{const}\,\top$.

[2] In a traditional ML-style module system, this would have to be a module signature with a single type field; it is one of the frustrating parts of the traditional system that there is not a simpler module signature for just a single type.

Finally, we have: Type-is-static : IsStatic (Mod Type)

This leads us to the first main contribution of the paper: extending the theory of SOGATs to include a reflexive sub-universe. Unfortunately, in order to do this we must be slightly more precise about what a SOGAT is.

We can do this in two ways: syntactic and semantic. The syntactic approach is to essentially formalizing the type theory that we have been using, including the $\bigcirc_{\mathsf{st}}$ operator. The semantic approach to SOGATs is *categories with representables*.

Definition 2.17 (Category with representables)**.** A **category with representables** (CwR) is a category $\mathcal{J}$ with finite limits equipped with a class $R_{\mathcal{J}}$ of morphisms called representable maps, such that

- $R_{\mathcal{J}}$ is stable under pullback
- Each map in $R_{\mathcal{J}}$ is exponentiable

A morphism of CwRs is a functor that preserves finite limits and the class of representables.

Example 2.18. For a category $\mathcal{C}$, the category $\mathcal{J} = \mathsf{Psh}(\mathcal{C})$ has a canonical CwR structure where the class of representable maps are the *unstraightenings* of representable presheaf families.

Definition 2.19 (Unstraightening)**.** If Q is a presheaf family over P, then its **unstraightening** is the first projection from the presheaf $(i : P) \times Q(i)$ to P.

We use $\mathcal{J}$ as our name for a generic CwR instead of $\mathcal{C}$ so as to remain consistent with the use of $\mathcal{C}$ as the category of contexts; when we have $\mathcal{J} = \mathsf{Psh}(\mathcal{C})$, we think of $\mathcal{J}$ as the category of *judgments*.

The relationship between SOGATs and CwRs is analogous to the relationship between GATs and finite limit categories. Each SOGAT presents a small CwR, and models of that SOGAT are equivalent to CwR-morphisms out of that CwR into a presheaf CwR.

Definition 2.20 (Category with representables equipped with open modality)**.** Define a $\bigcirc$-CwR to be a CwR $(\mathcal{J}, R_{\mathcal{J}})$, equipped with an idempotent monad $\bigcirc_{\mathcal{J}} : (\mathcal{J}, R_{\mathcal{J}}) \to (\mathcal{J}, R_{\mathcal{J}})$ that we call the static modality. A morphism of $\bigcirc$-CwRs is a CwR morphism that preserves the static modality.

Example 2.21. If $\mathcal{C}$ is an indexed category, then $\mathsf{IPsh}(\mathcal{C})$ is a $\bigcirc$-CwR where the static subuniverse is given by $\bigcirc_{\mathsf{st}}$.

Just as Uemura defines "type theory" to be a small CwR $\mathcal{T}$, and a model of that type theory consists of a category $\mathcal{C}$ and a CwR-morphism from $\mathcal{T}$ to $\mathsf{Psh}(\mathcal{C})$, we may define "type theory with a static/dynamic phase distinction" to be a small $\bigcirc$-CwR $\mathcal{T}$, and a model consists of an indexed category $\mathcal{C}$ with a $\bigcirc$-CwR morphism from $\mathcal{T}$ to $\mathsf{IPsh}(\mathcal{C})$.

In the next section, we will define a type theory in logical framework style; this should be semantically understood as presenting a small $\bigcirc$-CwR.

3 Hippogriff

Hippogriff is a small language meant to showcase the feasibility of formulating a module system synthetically via the phase distinction and then incorporating language features like unrestricted

recursion which are classically not permitted in a dependently typed language by restricting them to the dynamic fragment.

While small (the elaborator and interpreter is implemented in around 1500 lines of Haskell), Hippogriff is a non-trivial language with a higher-order module system, sum types, and unrestricted recursion; we believe this serves as evidence that a direct implementation of the synthetic phase distinction is a practical and simple approach to modularity features.

In this section, we define the type theory for Hippogriff alongside code samples that typecheck and run in the current Hippogriff implementation. We could work in the most minimal syntax for SOGATs, but it will be more elegant to assume various sugar such as inductive datatypes as arguments (which should be thought of as postulating a *family* of judgments) and implicit arguments. These will be explained as they are used.

3.1 Energy and the basic judgments

Nominal types are an essential tool for organizing a code base and for providing readable error messages; any programmer would prefer to see a type error mentioning `List Int` rather than `fix A. ('nil | 'cons Int A)`. Hippogriff uses an approach to nominality that is stolen from the Narya source code [6]. We start off by parametrizing our standard representable family by *energy type*. We give Energy as an inductive type, but we only ever consider maps out of it, so it does not actually increase the power of our language, it is merely a convenience feature.

$$
\begin{aligned}
&\textbf{data}\ \mathsf{Energy} : \mathsf{Set}\ \textbf{where}\\
&\quad \mathsf{kinetic} : \mathsf{Energy}\\
&\quad \mathsf{potential} : \mathsf{Energy}\\
\\
&\mathsf{Ty} : \mathsf{Energy} \to \mathsf{Jdg}\\
&\mathsf{Ty}_k := \mathsf{Ty}\ \mathsf{kinetic}\\
&\mathsf{Ty}_p := \mathsf{Ty}\ \mathsf{potential}\\
&\mathsf{El} : \mathsf{Energy} \to \mathsf{Ty}_k \to \mathsf{Jdg}^+\\
&\mathsf{El}_k := \mathsf{El}\ \mathsf{kinetic}\\
&\mathsf{El}_p := \mathsf{El}\ \mathsf{potential}
\end{aligned}
$$

The idea here is that we have a normal type theory for $(\mathsf{Ty}_k, \mathsf{El}_k)$. However, at certain points we introduce fresh terms that *behave* in a certain way (in the sense of having certain universal properties). This is done in the following way (we reuse the same notation for types and elements).

$$
\begin{aligned}
&\_ \rightsquigarrow \_ : \mathsf{Ty}_k \to \mathsf{Ty}_p \to \mathsf{Jdg}\\
&\_ \rightsquigarrow \_ : \{A : \mathsf{Ty}_k\} \to \mathsf{El}_k\, A \to \mathsf{El}_p\, A \to \mathsf{Jdg}^+\\
&\mathsf{let}_p : \{A\, B : \mathsf{Ty}_k\} \to (a' : \mathsf{El}_p\, A) \to ((a : \mathsf{El}_k\, A) \to a \rightsquigarrow a' \to \mathsf{El}_k\, B) \to \mathsf{El}_k\, B
\end{aligned}
$$

Within the body of the let binding, a is fresh; it is not equal to any previously defined term. However, we know that it *behaves like* a', and this allows us to use it as, for instance, a sum type.

Example 3.1. In the following code snippet, we demonstrate pattern matching against a list. All top-level bindings in Hippogriff use let_p, so in `head`, `List A` is a fresh type that behaves like a sum type.

```
def List (A : Type) : Type := sum
  'nil
  'cons A (List A)
end

def head (A : Type) (xs : List A) : A := match xs
  'nil => abandon
  'cons x _ => x
end
```

In order to support definitions which are not nominal, we use the following construct.

$$\mathsf{realize} : \{A : \mathsf{Ty}_k\} \to \mathsf{El}_k\, A \to \mathsf{El}_p\, A$$
$$\mathsf{realize\text{-}behavior} : \{A : \mathsf{Ty}_k\} \to (a_0\, a_1 : \mathsf{El}_k\, A) \to (a_0 \rightsquigarrow \mathsf{realize}\, a_1) \to a_0 = a_1$$

Example 3.2. In the following code snippet, we define `MyInt` to be `Int` directly, and the elaborator inserts a realize node automatically. Then later on, `MyInt` is just an alias of `Int`.

```
def MyInt : Type := Int

def MyInt/make (x : Int) : MyInt := x
```

Finally, crucially, we add the axiom that the type judgment is purely static, using Definition 2.16.

$$\mathsf{Ty\text{-}is\text{-}static} : \{e : \mathsf{Energy}\} \to \mathsf{IsStatic}\,(\mathsf{Ty}\, e)$$

3.2 Levels and universes

Hippogriff is meant to be a modular programming language, so we need a "type of small types." However, it turns out to be convenient to go beyond this and also have a "type of large types"; this makes it easier to support features like module signatures with arguments. We formalize this in the following way, inspired by Jon Sterling's treatment of levels for Pterodactyl [7]. First, we introduce a poset of levels.

$$\begin{aligned}
&\textbf{data}\ \mathsf{Level} : \mathsf{Set}\ \textbf{where}\\
&\quad \mathsf{small} : \mathsf{Level}\\
&\quad \mathsf{large} : \mathsf{Level}\\
&\quad \mathsf{top} : \mathsf{Level}
\end{aligned}$$

$$\begin{aligned}
&_ \le _ : \mathsf{Level} \to \mathsf{Level} \to \mathsf{Prop}\\
&\mathsf{small} \le _ := \top\\
&\mathsf{large} \le \mathsf{small} := \bot\\
&\mathsf{large} \le _ := \top\\
&\mathsf{top} \le \mathsf{small} := \bot\\
&\mathsf{top} \le \mathsf{large} := \bot\\
&\mathsf{top} \le \mathsf{top} := \top
\end{aligned}$$

Then, we have a judgment which says that a type sits at a certain level. This is functorial in the level, so if A@small then A@large.

$$\_@\_ : \{e : \mathsf{Energy}\} \to \mathsf{Ty}\, e \to \mathsf{Level} \to \mathsf{Prop}$$
$$@/\mathsf{functorial} : \{e : \mathsf{Energy}\} \to \{\ell_0\, \ell_1 : \mathsf{Level}\} \to \ell_0 \leq \ell_1 \to \{A : \mathsf{Ty}\, e\} \to A@\ell_0 \to A@\ell_1$$

Finally, we internalize types at certain levels into terms via universes.

$$\begin{aligned}&\textbf{data}\ \mathsf{Universe} : \mathsf{Level} \to \mathsf{Level} \to \mathsf{Set}\ \textbf{where}\\ &\quad \mathsf{Universe/small} : \mathsf{Universe\ small\ large}\\ &\quad \mathsf{Universe/large} : \mathsf{Universe\ large\ top}\end{aligned}$$

$$\mathsf{U} : \{\ell_0\, \ell_1 : \mathsf{Level}\} \to \mathsf{Universe}\, \ell_0\, \ell_1 \to \mathsf{Ty}_k$$
$$\mathsf{U/level} : \{\ell_0\, \ell_1 : \mathsf{Level}\} \to (u : \mathsf{Universe}\, \ell_0\, \ell_1) \to (\mathsf{U}\, u)@\ell_1$$
$$(\mathsf{code}, \mathsf{decode}, \mathsf{U}/\beta, \mathsf{U}/\eta) : \{\ell_0\, \ell_1 : \mathsf{Level}\} \to \{u : \mathsf{Universe}\, \ell_0\, \ell_1\} \to \{e : \mathsf{Energy}\} \to$$
$$(A : \mathsf{Ty}\, e) \times (A@\ell_0) \cong \mathsf{El}\, e\, (\mathsf{U}\, u)$$

The two parameters to Universe are the levels that the universe *decodes into* and *lives at*, respectively. So, for instance, U Universe/small stores codes for small types (hence decodes into small types) and is a large type.

Example 3.3. In Hippogriff, the appropriate coding and decoding is done automatically by the elaborator. For example, consider the polymorphic identity.

```
def id (A : Type) (a : A) : A := a
```

The term `A` of type `Type` (which is the syntax for the small universe) is automatically decoded into a type when used on the right hand side of a colon.

In fact, all syntax in Hippogriff is elaborated as terms; there is no direct syntax for types. This makes the elaborator much easier to write, because elaborating a type is merely checking at a universe. Elements of the large universe are introduced with `theory`

```
theory Eq := sig
  t : Type
  eq : t -> t -> t
end
```

The only difference between `def` and `theory` is that `def` has a return type annotation. If we allowed `Theory` as a term, we could have

```
def Eq : Theory := sig
  t : Type
  eq : t -> t -> t
end
```

However, that would either require `Theory` to be an element of a further universe (which seems overkill), or mean that type elaboration could not be implemented by checking at a universe, and in any case it seems good to have an indication that a top-level binding is a theory.

We have to also say how decoding interacts with behavior.

$$(\rightsquigarrow/\mathsf{code}, \rightsquigarrow/\mathsf{decode}, \rightsquigarrow/\mathsf{U}/\beta, \rightsquigarrow/\mathsf{U}/\eta) : \{\ell_0\, \ell_1 : \mathsf{Level}\} \to \{u : \mathsf{Universe}\, \ell_0\, \ell_1\} \to$$
$$(c : \mathsf{El}_k\, (\mathsf{U}\, u)) \to (c' : \mathsf{El}_p\, (\mathsf{U}\, u)) \to (\mathsf{decode}\, c) \rightsquigarrow (\mathsf{decode}\, c') \cong c \rightsquigarrow c'$$

Finally, we add an axiom that says that the elements of a small type are statically contractible.

$$\mathsf{Small\text{-}dynamic} : \{A : \mathsf{Ty}_k\} \to \{e : \mathsf{Energy}\} \to A@\mathsf{small} \to \text{🔓}_{\mathsf{st}} \to \mathsf{IsContr}\,(\mathsf{El}\,e\,A)$$

3.3 Dependent function types

Function types are more or less the standard thing, adapted to support both kinetic and potential elements.

$$\Pi : (A : \mathsf{Ty}_k) \to (\mathsf{El}_k\,A \to \mathsf{Ty}_k) \to \mathsf{Ty}_k$$
$$(\mathsf{lam}, \mathsf{app}, \Pi/\beta, \Pi/\eta) : \{A : \mathsf{Ty}_k\} \to \{B : \mathsf{El}_k\,A \to \mathsf{Ty}_k\} \to \{e : \mathsf{Energy}\} \to$$
$$((a : \mathsf{El}_k\,A) \to \mathsf{El}\,e\,(B\,a)) \cong \mathsf{El}\,e\,(\Pi\,A\,B)$$

We then need some axioms for how application interacts with $\rightsquigarrow$.

$$(\rightsquigarrow/\mathsf{lam}, \rightsquigarrow/\mathsf{app}, \rightsquigarrow/\Pi/\beta, \rightsquigarrow/\Pi/\eta) : \{A : \mathsf{Ty}_k\} \to \{B : \mathsf{El}_k\,A \to \mathsf{Ty}_k\} \to \{e : \mathsf{Energy}\} \to$$
$$(f : \mathsf{El}_k\,(\Pi\,A\,B)) \to (f' : \mathsf{El}_p\,(\Pi\,A\,B)) \to ((a : \mathsf{El}_k\,A) \to \mathsf{app}\,f\,a \rightsquigarrow \mathsf{app}\,f'\,a) \cong f \rightsquigarrow f'$$

Example 3.4. After evaluating the definition

```
def Maybe (A : Type) : Type := sum
  'just A
  'nothing
end
```

we have `Maybe` bound to a neutral of type `Type -> Type` with behavior given by the lambda

```
A => sum
  'just A
  'nothing
end
```

Following $\rightsquigarrow/\mathsf{app}$, the behavior of `Maybe Int` is then `sum { 'just Int; 'nothing }`.

Finally, we must give an axiom that characterizes the level of Π types. Here we have a choice to make. At the very least, in order for $\Pi\,A\,B$ to be at level ℓ we must have $(B\,a)@\ell$ for all $a : \mathsf{El}_k\,A$; otherwise we have nonsensical results like $\Pi\,1\,B$ being at a smaller level than B. If this is the only restriction, then we get impredicativity, just like in System F_ω, as for instance `Type -> Int` becomes a small type. With the current treewalking interpreter we have for Hippogriff, this is not a problem, however making this choice may preclude a monomorphizing compilation scheme later on and necessitate a uniform value representation. A more conservative approach, which is the approach we currently take, is to require A and B to both be at a certain level for $\Pi\,A\,B$ to be at that level.

$$\Pi/\mathsf{level} : \{A : \mathsf{Ty}_k\} \to \{B : \mathsf{El}\,A \to \mathsf{Ty}_k\} \to \{\ell : \mathsf{Level}\} \to$$
$$A@\ell \to ((x : \mathsf{El}\,A) \to (B\,a)@\ell) \to (\Pi\,A\,B)@\ell$$

3.4 Record types

Typically in dependent type theory papers, only the rules for binary Σ types are presented and the extension to general records is left as an exercise. Fortunately, one advantage of working in a

rich metatheory is that we can do general record types without needing a careful use of imprecise ellipses.

Assume that Name is some fixed set, such as the set of non-empty utf8 strings. In fact, for our purposes, it could well be the unit type; the task of determining from user syntax which field is meant is a task for the elaborator, not for the core language.

We then define records as an internalization of telescopes. Note that the use of the inductive set Names is only in argument position, so this can really be thought of as an collection of axiom families, one for each list of names.

$$
\begin{aligned}
&\textbf{data}\ \mathsf{Names} : \mathsf{Set}\ \textbf{where}\\
&\quad \mathsf{empty} : \mathsf{Names}\\
&\quad \mathsf{cons} : \mathsf{Name} \to \mathsf{Names} \to \mathsf{Names}\\
\\
&\mathsf{Tele} : \mathsf{Names} \to \mathsf{Jdg}\\
&\mathsf{Tele}\ \mathsf{empty} := []\\
&\mathsf{Tele}\ (\mathsf{cons}\ x\ xs) := [\mathsf{head} : \mathsf{Ty}_k, \mathsf{tail} : \mathsf{El}_k\ \mathsf{head} \to \mathsf{Tele}\ xs]\\
\\
&\mathsf{Record} : \{xs : \mathsf{Names}\} \to \mathsf{Tele}\ xs \to \mathsf{Ty}_p\\
\\
&\mathsf{Elts} : (e : \mathsf{Energy}) \to \{xs : \mathsf{Names}\} \to \mathsf{Tele}\ xs \to \mathsf{Jdg}^+\\
&\mathsf{Elts}\ \_\ \{\mathsf{empty}\}\ \_ = []\\
&\mathsf{Elts}\ e\ \{\mathsf{cons}\ x\ xs\}\ t = [\mathsf{head} : \mathsf{El}\ e\ t.\mathsf{head}, \mathsf{tail} : \mathsf{Elts}\ e\ (t.\mathsf{tail}\ \mathsf{head})]\\
\\
&(\mathsf{struct}, \mathsf{destruct}, \mathsf{Record}/\beta, \mathsf{Record}/\eta) : \{e : \mathsf{Energy}\} \to \{t : \mathsf{Tele}_k\} \to \{A : \mathsf{Ty}_k\} \to\\
&\quad \{A \rightsquigarrow^{\mathsf{Ty}} \mathsf{Record}\ t\} \to \mathsf{Elts}\ e\ t \cong \mathsf{El}\ e\ A
\end{aligned}
$$

In this scheme, Record returns a potential type; this means that records are only allowed to be nominal. Because of this, the universal property takes in an arbitrary type with a witness that that type behaves like a record. It would not be greatly different to also allow "anonymous" record types; before the introduction of the kinetic/potential system in Hippogriff this was how record types worked.

Example 3.5. Consider the following code.

```
theory PointedType := sig
  t : Type
  pt : t
end

def Unit : PointedType := struct
  t := sum
    'tt
  end
  pt := 'tt
end
```

This declares a new large type that behaves like a record type, and then an element of that large type whose first field behaves like a singleton, and whose second field is the unique point in that singleton. This crucially relies on the ability to form a potential struct; if structs were only kinetic then they could not have nominal fields.

We then specify how the eliminator interacts with $\rightsquigarrow$.

$$\rightsquigarrow/\mathsf{Elts} : \{xs : \mathsf{Names}\} \to \{t : \mathsf{Tele}\, xs\} \to \mathsf{Elts}\, \mathsf{kinetic}\, t \to \mathsf{Elts}\, \mathsf{potential}\, t \to \mathsf{Jdg}^+$$
$$\rightsquigarrow/\mathsf{Elts}\, \{\mathsf{empty}\}\, \_\, \_ = []$$
$$\rightsquigarrow/\mathsf{Elts}\, \{\mathsf{cons}\, x\, xs\}\, e\, e' = [\mathsf{head} : e.\mathsf{head} \rightsquigarrow e'.\mathsf{head}, \mathsf{tail} : \rightsquigarrow/\mathsf{Elts}\, e.\mathsf{tail}\, e'.\mathsf{tail}]$$

$$(\rightsquigarrow/\mathsf{struct}, \rightsquigarrow/\mathsf{destruct}, \rightsquigarrow/\mathsf{Record}/\beta, \rightsquigarrow/\mathsf{Record}/\eta) : \{e : \mathsf{Energy}\} \to \{t : \mathsf{Tele}_k\} \to$$
$$\{A : \mathsf{Ty}_k\} \to \{A \rightsquigarrow \mathsf{Record}\, t\} \to (a : \mathsf{El}_k\, A) \to (a' : \mathsf{El}_p\, A) \to$$
$$\rightsquigarrow/\mathsf{Elts}\, (\mathsf{destruct}\, a)\, (\mathsf{destruct}\, a') \cong a \rightsquigarrow a'$$

And finally, we specify what level a record type lives at; if all of the fields of a record are at a level then the record is at that level.

$$@/\mathsf{Tele} : \{xs : \mathsf{Names}\} \to \mathsf{Tele}\, xs \to \mathsf{Level} \to \mathsf{Prop}$$
$$@/\mathsf{Tele}\{\mathsf{empty}\}\, \_\, \_ = \top$$
$$@/\mathsf{Tele}\{\mathsf{cons}\, x\, \mathsf{xs}\}\, t\, \ell = [\mathsf{head} : t.\mathsf{head}@\ell, \mathsf{tail} : @/\mathsf{Tele}\, t.\mathsf{tail}\, \ell]$$

$$\mathsf{Record}/\mathsf{level} : \{\ell : \mathsf{Level}\} \to \{xs : \mathsf{Names}\} \to (t : \mathsf{Tele}\, xs) \to (A : \mathsf{Ty}_k) \to$$
$$(A \rightsquigarrow \mathsf{Record}\, t) \to @/\mathsf{Tele}\, t\, \ell \to A@\ell$$

3.5 Sum types

Sum types differ in several ways from the three types (universes, dependent function types, record types) that we have considered so far. First of all, if we are to sustain the static phase distinction, we must be very careful about levels. Specifically, as small types are statically contractible, we must be careful that we can only eliminate into other statically contractible types.

Example 3.6. In Hippogriff, it is perfectly valid to have a function `LandOrSea -> Type`, where

```
def LandOrSea : Type := sum
  'land
  'sea
end
```

However, it is a key invariant of the system that the only functions that one can actually write down of that type are constant. Specifically, a function

```
def paul-revere (method : LandOrSea) : Type := match method
  'land => Unit
  'sea => Bool
end
```

would be rejected with the error message "cannot eliminate from a sum type into the large type `Type`."

While this means that Hippogriff is potentially vulnerable to British surprise attacks, without this restriction the interaction between sum types, the "type judgment is static" axiom and the "small types are statically contractible" judgments would end trivializing the system, in the sense of making all types equal.

On the other hand, it is conceivable that a future version of Hippogriff could support "static sum types" which would not be small, and which could eliminate into large types, which would be similar to DataKinds in Haskell (cite). However, how these would interact with recursion is currently an open question, so we have deferred this matter for now.

With that preamble out of the way, we get to the axioms for sum types. Just like record types were parameterized by a list of names, sum types are parameterized by a shape.

$$\begin{aligned}
&\textbf{data}\ \mathsf{Shape} : \mathsf{Set}\ \textbf{where}\\
&\quad \mathsf{empty} : \mathsf{Shape}\\
&\quad \mathsf{cons} : \mathsf{Name} \to \mathsf{Nat} \to \mathsf{Shape} \to \mathsf{Shape}
\end{aligned}$$

$$\mathsf{Ty}_e^{\mathsf{small}} := (A : \mathsf{Ty}_e) \times (A@\mathsf{small})$$

$$\begin{aligned}
&\mathsf{STele} : \mathsf{Nat} \to \mathsf{Jdg}\\
&\mathsf{STele}\, 0 = []\\
&\mathsf{STele}\, (n+1) = [\mathsf{head} : \mathsf{Ty}_k^{\mathsf{small}}, \mathsf{tail} : \mathsf{STele}\, n]
\end{aligned}$$

STele stands for simple telescope; unlike a normal telescope the types in this telescope are all small and don't depend on each other. We can then define the input to the sum type former as follows.

$$\begin{aligned}
&\mathsf{Variants} : \mathsf{Shape} \to \mathsf{Jdg}\\
&\mathsf{Variants}\, \mathsf{empty} := []\\
&\mathsf{Variants}\, (\mathsf{cons}\, x\, n\, s) := [\mathsf{head} : \mathsf{STele}\, n, \mathsf{tail} : \mathsf{Variants}\, s]
\end{aligned}$$

$$\mathsf{Sum} : \{s : \mathsf{Shape}\} \to \mathsf{Variants}\, s \to \mathsf{Ty}_p^{\mathsf{small}}$$

In order to give the introduction rule for Sum, we muse make some auxiliary definitions which describe a well-formed application of a tag in the sum type to its proper arguments, according to

a Variants specification.

$$
\begin{aligned}
&\mathsf{data}\ \mathsf{Tag} : \mathsf{Shape} \to \mathsf{Set}\ \mathbf{where}\\
&\quad \mathsf{here} : \mathsf{Tag}\,(\mathsf{cons}\,x\,n\,s)\\
&\quad \mathsf{there} : \mathsf{Tag}\,s \to \mathsf{Tag}\,(\mathsf{cons}\,x\,s)\\
\\
&\mathsf{SElts} : \{n : \mathsf{Nat}\} \to \mathsf{STele}\,n \to \mathsf{Jdg}^+\\
&\mathsf{SElts}\,\{0\}\,\_\,\_ := []\\
&\mathsf{SElts}\,\{n+1\}\,e\,t := [\mathsf{head} : \mathsf{El}_k\,t.\mathsf{head}, \mathsf{tail} : \mathsf{SElts}_k\,t.\mathsf{tail}]\\
\\
&\mathsf{Args} : \{s : \mathsf{Shape}\} \to (t : \mathsf{Tag}\,s) \to \mathsf{Variants}\,s \to \mathsf{Jdg}^+\\
&\mathsf{Args}\,\mathsf{here}\,(\mathsf{cons}\,x\,n\,\_)\,v := \mathsf{SElts}\,v.\mathsf{head}\\
&\mathsf{Args}\,(\mathsf{there}\,t)\,(\mathsf{cons}\,\_\,\_\,s)\,v := \mathsf{Args}\,t\,s\,v.\mathsf{tail}
\end{aligned}
$$

Finally, we have the introduction, elimination, and computation rules. These are in a different form from other types that we have considered. This is because in other types the rules formed an isomorphism, but SOGATs do not permit a function *into* a meta-level sum type, only out of one (because a function out of a meta-level sum type is really just a collection of functions, using the universal property). Thus, we cannot have $\mathsf{El}_k\,A \cong (t : \mathsf{Tag}\,s) \times \mathsf{Args}\,t\,v$. Instead, the eliminator takes in a continuation out of $(t : \mathsf{Tag}\,s) \times \mathsf{Args}\,t\,v$ which is equivalent to providing a function for each tag.

$$
\begin{aligned}
&\mathsf{tag} : \{s : \mathsf{Shape}\} \to \{v : \mathsf{Variants}\,s\} \to \{A : \mathsf{Ty}_k\} \to \{A \rightsquigarrow \mathsf{Sum}\,v\} \to\\
&\quad (t : \mathsf{Tag}\,s) \to \mathsf{Args}\,t\,v \to \mathsf{El}_k\,A\\
&\mathsf{match} : \{s : \mathsf{Shape}\} \to \{v : \mathsf{Variants}\,s\} \to \{A : \mathsf{Ty}_k\} \to \{A \rightsquigarrow \mathsf{Sum}\,v\} \to\\
&\quad \mathsf{El}_k\,A \to \{B : \mathsf{Ty}_k^{\mathsf{small}}\} \to ((t : \mathsf{Tag}\,s) \to \mathsf{Args}\,t\,v \to \mathsf{El}_k\,B) \to \mathsf{El}_k\,B\\
&\mathsf{Sum}/\beta : \{s : \mathsf{Shape}\} \to \{v : \mathsf{Variants}\,s\} \to \{A : \mathsf{Ty}_k\} \to \{A \rightsquigarrow \mathsf{Sum}\,v\} \to\\
&\quad (B : \mathsf{Ty}_k^{\mathsf{small}}) \to (f : (t : \mathsf{Tag}\,s) \to \mathsf{Args}\,t\,v \to \mathsf{El}_k\,B) \to (t : \mathsf{Tag}\,s) \to (a : \mathsf{Args}\,t\,v) \to\\
&\quad \mathsf{match}\,(\mathsf{tag}\,t\,a)\,f = f\,t\,a
\end{aligned}
$$

Example 3.7. We can possibly still give warning about the British as long as it is acceptable to return numerals rather than cardinals. We use > and = to indicate the input and output of a command at a REPL.

```
def paul-revere (method : LandOrSea) : Int := match method
  'land => 1
  'sea => 2
end

> paul-revere 'land
= 1
```

3.6 Abandonment

In "The Error Model," Joe Duffy argues that there are (at least) two types of errors that one encounters in general-purpose programming. One type of error indicates the failure of an

operation that is known to possibly fail, such as the non-existence of a file whose path the user typed in. The other type of error indicates that an invariant that would have been maintained if the rest of the program were correct has in fact not been maintained. Duffy argues that often the only sensible thing to do in this situation is halt the program as soon as possible, as once an error of this sort is detected the extent of the bug is unknowable and therefore recovery is impossible. Duffy calls this kind of halting "abandoning." Of course, there are situations in which a higher-level process can reset to a known-good state, but in general abandonment is an essential component of software that is not completely statically checked. As with current technology, full verification of every interesting invariant that might appear in a general-purpose program is at least a burden to the developer, if not impossible, abandonment is an essential component of a general-purpose programming language.

Fortunately, unlike in a typical dependently typed language, in Hippogriff abandonment does not compromise type checking, because we restrict abandonment to small types and so abandonment is statically evaluated to the opaque element, just like any other operation.

The rule for abandonment is very simple.

$$\mathsf{abandon} : \{A : \mathsf{Ty}_k^{\mathsf{small}}\} \to \mathsf{El}_k\, A$$

Example 3.8. We can define an analogue of Rust's `.unwrap()` operation as follows.

```
def unwrap (A : Type) (ma : Maybe A) : A := match ma
  'just a => a
  'nothing => abandon
end
```

3.7 Recursion, guarded and unguarded

This section is relatively more conjectural than the rest. The techniques for recursion described here are implemented in Hippogriff and seem to work, but require a slightly more advanced metatheory to describe. We include it in the current paper as a demonstration of the kind of thing that the phase distinction could enable.

A naive attempt to do recursion would look like the following.

$$\mathsf{letfix}_p : \{A\, B : \mathsf{Ty}_k\} \to (f : \mathsf{El}_k\, A \to \mathsf{El}_p\, A) \to ((a : \mathsf{El}_k\, A) \to (a \rightsquigarrow f\, a) \to \mathsf{El}_k\, B) \to \mathsf{El}_k\, B$$

This would permit definition of types like

```
def List (A : Type) : Type := sum
  'empty
  'cons A (List A)
end
```

After this definition, `List` would be a neutral that behaved like

```
A => sum { 'empty; 'cons A (List A) }
```

If we did not have realize (which allowed converting a kinetic element to a potential element), then this definition for letfix_p would be fine, as the only way to give a potential element would be to use Sum or Record. Thus, the algorithm to determine the behavior of a neutral introduced with letfix would always give us at least one "behavior node" to work with, which would allow elaboration to progress. However, realize permits us to write down

```
def List (A : Type) : Type := List A
```

which is not desirable.

So instead, we implement *guarded recursion*. To do this, we assume an applicative modality $\triangleright$ on judgments (cite). We can then fix our earlier definition.

$$\mathsf{letfix}_p : \{A\, B : \mathsf{Ty}_k\} \to (f : \triangleright(\mathsf{El}_k\, A) \to \mathsf{El}_p\, A) \to ((a : \mathsf{El}_k\, A) \to (a \rightsquigarrow f\,(\mathsf{pure}\, a)) \to \mathsf{El}_k\, B) \to \mathsf{El}_k\, B$$

All top-level definitions in Hippogriff are wrapped in letfix_p.

Then we introduce two ways of "unguarding." The first way is by modifying the definition of Variants for sum types to take a delayed simple telescope.

$$\begin{aligned}
&\mathsf{Variants} : \mathsf{Shape} \to \mathsf{Jdg}\\
&\mathsf{Variants}\,\mathsf{empty} := []\\
&\mathsf{Variants}\,(\mathsf{cons}\, x\, n\, s) := [\mathsf{head} : \triangleright(\mathsf{STele}\, n), \mathsf{tail} : \mathsf{Variants}\, s]
\end{aligned}$$

This allows recursive use of the top-level definition that we are defining, so long as we are producing a type for one of the constructor arguments in a sum type.

Example 3.9. Mutual recursion is achieved via recursion at a record type.

```
theory EvenAndOdd/typeof := sig
  even : Type
  odd : Type
end

def EvenAndOdd : EvenAndOdd/typeof := struct
  even := sum
    'zero
    'succ EvenAndOdd.odd
  end

  odd := sum
    'succ EvenAndOdd.even
  end
end

def three : EvenAndOdd.odd := 'succ ('succ ('succ 'zero))
```

The second is we always allow recursion at a small type. Just like with abandonment, this does not impact type checking because any recursion is just evaluated to the opaque element.

$$\mathsf{rec} : \{A : \mathsf{Ty}_k^{\mathsf{small}}\} \to \triangleright(\mathsf{El}_k\, A) \to \mathsf{El}_k\, A$$

Example 3.10. We can define `map` for lists in the following way.

```
def map (A : Type) (B : Type) (f : A -> B) (xs : List A) : List B := match xs
  'empty => 'empty
  'cons x xs => 'cons (f x) (rec (map A B f xs))
end
```

```
> map Int Int (n => n + 1) ('cons 1 ('cons 2 'empty))
= 'cons 2 ('cons 3 'empty)
```

In order to prove that type checking for Hippogriff is decidable, in addition to a normalization result we must also prove a "productivity" result, which is that the question of what behavior a neutral has (if any) is decidable; we conjecture that guarded recursion helps with this (given that it has made an implementation feasible); we do not do this in the current paper but it is certainly a good area for future work.

4 Implementation notes

The thesis of this paper is that a direct implementation of the synthetic account of the phase distinction as found in "Logical Relations as Types" is a sensible and useful technique for integrating modularity features into a programming language.

In this section, we provide evidence for this thesis by describing the elaboration algorithm in the Hippogriff implementation.

4.1 A note on notation

4.2 A normalization by evaluation refresher

Normalization by evaluation is a technique to decide definitional equality of open types and terms in extensions of the lambda calculus. In this section we walk through a basic implementation of normalization by evaluation for the untyped lambda calculus. For added flavor, we give the implementation in Hippogriff so it also serves as an example of what hippogriff looks like on a non-trivial problem.

We start with some basic utilities: natural numbers and backwards lists (also called snoc lists).

```
def Nat : Type := sum
  'zero
  'succ Nat
end

def Bwd (A : Type) : Type := sum
  'nil
  'snoc (Bwd A) A
end

def Bwd/elemAt (A : Type) (xs : Bwd A) (n : Nat) : A := match xs
  'nil => abandon
  'snoc xs x => match n
    'zero => x
    'succ n => rec (Bwd/elemAt A xs n)
  end
end
```

We then define types for syntax and values.

```
# backwards index, 0 is the last element in context
```

```
def BId : Type := Nat
# forwards index, 0 is the first element in context
def FId : Type := Nat

def Stx : Type := sum
  'var BId
  'app Stx Stx
  'lam Stx
end

def Val : Type := sum
  'neu FId (Bwd Val)
  'lam (Val -> Val)
end
```

Normalization by evaluation happens in two stages. The first stage *evaluates* syntax to produce values, and then the second stage *quotes* the values to produce syntax again, but the syntax is in a normal form. The process of evaluating computes "as much as possible until it gets stuck."

```
def Env : Type := Bwd Val

def app (v0 : Val) (v1 : Val) : Val := match v0
  'neu i vs => 'neu i ('snoc vs v1)
  'lam f => f v1
end

def eval (e : Env) (t : Stx) : Val := match t
  'var i => Bwd/elemAt Val e i
  'app t0 t1 => app (rec (eval e t0)) (rec (eval e t1))
  'lam t => 'lam (v => rec (eval ('snoc e v) t))
end
```

Then quoting instantiates lambda terms with fresh values, in order to re-constitute syntax which contains no meta-level functions.

```
theory Quoter/typeof := sig
  spine : Bwd Val -> Stx -> Stx
  val : Val -> Stx
end

def Quoter (n : Nat) : Quoter/typeof := struct
  spine := sp => t => match sp
    'nil => t
    'snoc sp v => 'app (rec (Quoter n .spine sp t)) (rec (Quoter n .val v))
  end

  val := v => match v
    'neu i vs => rec (Quoter n .spine vs ('var i))
    'lam f => 'lam (rec (Quoter ('succ n) .val (f ('neu n 'nil))))
  end
end
```

Here `spine` and `val` are mutually recursive, so we need to put them in a struct. Interestingly, Hippogriff allows us to factor out their common argument `n : Nat` into the constructor for that struct; this can be thought of as a way of doing something like "object oriented programming" where `Quoter n` is a object that knows how to quote things in a context of length `n`.

Finally, normalization can be achieved by combining evaluation and quoting.

```
def norm (t : Stx) : Stx := Quote 'zero .val (eval 'nil t)

> norm ('lam ('app ('lam ('var 'zero)) ('var 'zero)))
= 'lam ('var 'zero)
```

Classically, definitional equality between two terms is checked by normalizing and then structurally comparing the resulting syntax. However in practice it is possible to write an algorithm on values directly which "fuses" quoting and the structural equality check; this is much more efficient because it can return early without fully normalizing once it detects non-equality.

4.3 Bidirectional elaboration with normalization by evaluation

4.4 Normalization by evaluation for static equivalence

4.5 Energy and normalization by evaluation

4.6 Tackling recursion

5 Future work

References

[1] Steve Awodey. Natural models of homotopy type theory. *Mathematical Structures in Computer Science*, 28(2):241–286, February 2018. doi:10.1017/S0960129516000268.

[2] Rafaël Bocquet. *Relative Induction Principles for Second-Order Generalized Algebraic Theories*. PhD thesis, Eötvös Loránd Tudományegyetem, 2025. URL: https://rafaelbocquet.gitlab.io/pdfs/thesis.pdf.

[3] John Cartmell. Generalised algebraic theories and contextual categories. *Annals of Pure and Applied Logic*, 32:209–243, 1986. doi:10.1016/0168-0072(86)90053-9.

[4] B. Jacobs. *Categorical Logic and Type Theory*. Number 141 in Studies in Logic and the Foundations of Mathematics. North Holland, Amsterdam, 1999.

[5] Ambrus Kaposi and Szumi Xie. Second-Order Generalised Algebraic Theories: Signatures and First-Order Semantics. In Jakob Rehof, editor, *9th International Conference on Formal Structures for Computation and Deduction (FSCD 2024)*, volume 299 of *Leibniz International Proceedings in Informatics (LIPIcs)*, pages 10:1–10:24, Dagstuhl, Germany, 2024. Schloss Dagstuhl – Leibniz-Zentrum für Informatik. doi:10.4230/LIPIcs.FSCD.2024.10.

[6] Michael Shulman. Narya. Gwaith-i-Mírdain, March 2026. URL: https://github.com/gwaithimirdain/narya.

[7] Jon Sterling. Fuss-free universe hierarchies, 2025. URL: https://www.jonmsterling.com/01HX/.

[8] Jonathan Sterling and Robert Harper. Logical Relations as Types: Proof-Relevant Parametricity for Program Modules. *Journal of the ACM*, 68(6):1–47, December 2021. doi:10.1145/3474834.

[9] Paul Taylor. *Recursive Domains, Indexed Category Theory and Polymorphism*. PhD thesis, Cambridge University, 1986.