The Combination of Dynamic and Static Typing from a Categorical Perspective

Harley Eades III
Computer Science
Augusta University
Augusta, GA, USA
harley.eades@gmail.com

Michael Townsend
Computer Science
Augusta University
Augusta, GA, USA
mitownsend@augusta.edu

Abstract

In this paper we introduce a new categorical model based on retracts that combines static and dynamic typing. This model is initially based on the seminal work of Scott who showed that the untyped λ-calculus can be considered as typed using retracts. Following this we define a new simple type system which combines static and dynamic typing called Grady that corresponds to our model. Then we develop a gradually typed surface language for Grady, and show that it can be translated into Grady such that the gradual guarantee holds. Finally, to illustrate how this system can be extended with new features we extend both the surface and the core languages with bounded quantification such that the gradual guarantee is preserved.

CCS Concepts •**Software and its engineering** → **General programming languages;** •**Social and professional topics** → *History of programming languages;*

Keywords static typing, dynamic typing, gradual typing, categorical semantics, retract,typed lambda-calculus, untyped lambda-calculus, functional programming, bounded quantification

ACM Reference format:
Harley Eades III and Michael Townsend. 2016. The Combination of Dynamic and Static Typing from a Categorical Perspective. In *Proceedings of ACM Conference, Washington, DC, USA, July 2017 (Conference'17),* 7 pages.
DOI: 10.1145/nnnnnnn.nnnnnnn

1 Introduction

2 The Categorical Model

One strength and main motivation for giving a categorical model to a programming language is that it can expose the fundamental structure of the language. This arises because a lot of the language features that often cloud the picture go away, for example, syntactic notions like variables disappear. This can often simplify things and expose the underlying structure. For example, when giving the simply typed λ-calculus a categorical model we see that it is a cartesian closed category, but we also know that intuitionistic logic has the same model due to Lambek [8]; on the syntactic side these two theories are equivalent as well due to Howard [7]; this is known as the Curry-Howard-Lambek correspondence.

The previous point highlights one of the most powerful features of category theory: its ability to relate seemingly unrelated theories. It is quite surprising that the typed λ-calculus and intuitionistic logic share the same model. Thus, defining a categorical model for a particular programming language may reveal new and interesting relationships with existing work. In fact, one of the contributions of this paper is the new connection between Scott and Lambek's work to the new area of gradual typing and combing static and dynamic typing.

However, that motivation places defining a categorical model as an after thought. The Grady languages developed here were designed from the other way around. We started with the question, how do we combine static and dynamic typing categorically? Then after developing the model we use it to push us toward the correct language design. Reynolds [2] was a big advocate for the use of category theory in programming language research for this reason. We think the following quote – reported by Brookes et al. [2] – makes this point nicely:

> *Programming language semanticists should be the obstetricians of programming languages, not their coroners.*
>
> — *John C. Reynolds*

Categorical semantics provides a powerful tool to study language extensions. For example, purely functional programming in Haskell would not be where it is without the seminal work of Moggi and Wadler [10, 16] on using monads – a purely categorical notion – to add side effects to to pure functional programming languages. Thus, we believe that developing these types of models for new language designs and features can be hugely beneficial.

The model we develop here builds on the seminal work of Lambek [8] and Scott [11]. Lambek [8] showed that the typed λ-calculus can be modeled by a cartesian closed category. In the same volume as Lambek, Scott essentially showed that the untyped λ-calculus is actually typed. That is, typed theories are more fundamental than untyped ones. He accomplished this by adding a single type, ?, and two functions $\mathsf{squash} : (? \to ?) \to ?$ and $\mathsf{split} : ? \to (? \to ?)$, such that, $\mathsf{squash}; \mathsf{split} = \mathsf{id} : (? \to ?) \to (? \to ?)$, to a cartesian closed category. At this point he was able to translate the untyped λ-calculus into this unityped one.

Categorically, Scott modeled split and squash as the morphisms in a retract within a cartesian closed category – the same model as typed λ-calculus.

Both authors where supported by the National Science Foundation CRII CISE Research Initiation grant, "CRII:SHF: A New Foundation for Attack Trees Based on Monoidal Categories", under Grant No. 1565557.
Conference'17, Washington, DC, USA
2016. 978-x-xxxx-xxxx-x/YY/MM…$15.00
DOI: 10.1145/nnnnnnn.nnnnnnn

Definition 2.1. Suppose C is a category. Then an object A is a **retract** of an object B if there are morphisms $i : A \longrightarrow B$ and $r : B \longrightarrow A$ such that $i; r = \mathsf{id}_A$.

Thus, $? \to ?$ is a retract of ?, but we also require that $? \times ?$ be a retract of ?; this is not new, see Lambek and Scott [9]. Putting this together we obtain Scott's model of the untyped λ-calculus.

Definition 2.2. An **untyped λ-model**, $(C, ?, \mathsf{split}, \mathsf{squash})$, is a cartesian closed category C with a distinguished object ? and morphisms $\mathsf{squash} : S \longrightarrow ?$ and $\mathsf{split} : ? \longrightarrow S$ making the object S a retract of ?, where S is either $? \to ?$ or $? \times ?$.

Theorem 2.3 (Scott [11])**.** *An untyped λ-model is a sound and complete model of the untyped λ-calculus.*

So far we know how to model static types (typed λ-calculus) and unknown types (the untyped λ-calculus). The to make the Grady languages a bit more interesting we add natural numbers, but we will need a way to model these in a cartesian closed category.

We model the natural numbers with their (non-recursive) eliminator using what we call a non-recursive natural number object. This is a simplification of the traditional natural number object; see Lambek and Scott [9].

Definition 2.4. Suppose C is a cartesian closed category. A **non-recursive natural number object (NRNO)** is an object Nat of C and morphisms $\mathsf{z} : 1 \longrightarrow \mathsf{Nat}$ and $\mathsf{succ} : \mathsf{Nat} \longrightarrow \mathsf{Nat}$ of C, such that, for any morphisms $f : Y \longrightarrow X$ and $g : Y \times \mathsf{Nat} \longrightarrow X$ of C there is an unique morphism $\mathsf{case}_X\langle f, g\rangle : Y \times \mathsf{Nat} \longrightarrow X$ such that the following hold:

$$\langle \mathsf{id}_Y, \diamond_Y; \mathsf{z}\rangle; \mathsf{case}_{Y,X}\langle f, g\rangle = f \quad \langle \mathsf{id}_Y \times \mathsf{succ}\rangle; \mathsf{case}_{Y,X}\langle f, g\rangle = g$$

Informally, the two equations essentially assert that we can define $\mathsf{case}_{Y,X}$ as follows:

$$\mathsf{case}_{Y,X}\langle f, g\rangle\, y\, 0 = f\, y \qquad \mathsf{case}_{Y,X}\langle f, g\rangle\, y\, (\mathsf{succ}\, n) = g\, y\, n$$

At this point we can model both static and unknown types with natural numbers in a cartesian closed category, but we do not have any way of moving typed data into the untyped part and vice versa to obtain dynamic typing. To accomplish this we add two new morphisms $\mathsf{box}_C : C \longrightarrow ?$ and $\mathsf{unbox}_C : ? \longrightarrow C$ such that each atomic type, C, is a retract of ?. This enforces that the only time we can cast ? to another type is if it were boxed up in the first place. Combining all of these insights we obtain the complete categorical model.

Definition 2.5. A **gradual λ-model**, $(\mathcal{T}, C, ?, \mathsf{T}, \mathsf{split}, \mathsf{squash}, \mathsf{box}, \mathsf{unbox}, \mathsf{error})$, where $\mathcal{T}$ is a discrete category with at least two objects Nat and Unit, C is a cartesian closed category with a NRNO, $(C, ?, \mathsf{split}, \mathsf{squash})$ is an untyped λ-model, $\mathsf{T} : \mathcal{T} \longrightarrow C$ is an embedding – a full and faithful functor that is injective on objects – and for every object A of $\mathcal{T}$ there are morphisms $\mathsf{box}_A : TA \longrightarrow ?$ and $\mathsf{unbox}_A : ? \longrightarrow TA$ making TA a retract of ?. Furthermore, to model dynamic type errors, there is a morphism, $\mathsf{err}_A : \mathsf{Unit} \longrightarrow A$ of C, such that, the following equations hold w.r.t. $\mathsf{error}_{A,B} = A \xrightarrow{\mathsf{triv}_A} \mathsf{Unit} \xrightarrow{\mathsf{err}_B} B$:

$$\begin{aligned}
\mathsf{box}_{TA}; \mathsf{unbox}_{TB} &= \mathsf{error}_{TA,TB}, \text{ where } A \neq B\\
\mathsf{squash}_{S_1}; \mathsf{split}_{S_2} &= \mathsf{error}_{S_1,S_2}, \text{ where } S_1 \neq S_2\\
f; \mathsf{error}_{B,C} &= \mathsf{error}_{A,C}, \text{ where } f : A \longrightarrow B\\
\mathsf{error}_{A,B}; f &= \mathsf{error}_{A,C}, \text{ where } f : B \longrightarrow C\\
\langle \mathsf{error}_{A,B}, f\rangle &= \mathsf{error}_{A,B\times C}, \text{ where } f : A \longrightarrow C\\
\langle f, \mathsf{error}_{A,C}\rangle &= \mathsf{error}_{A,B\times C}, \text{ where } f : A \longrightarrow B\\
\mathsf{curry}(\mathsf{error}_{A\times B,C}) &= \mathsf{error}_{A,B\to C}
\end{aligned}$$

We call the category $\mathcal{T}$ the category of atomic types. We call an object, A, **atomic** iff there is some object A' in $\mathcal{T}$ such that $A = \mathsf{T}A'$. Note that we do not consider ? an atomic type.

Triggering dynamic type errors is a fundamental property of the criteria for gradually typed languages, and thus, the model must capture this. The new morphism $\mathsf{err}_A : \mathsf{Unit} \longrightarrow A$ is combined with the terminal morphism, $\mathsf{triv}_A : A \longrightarrow \mathsf{Unit}$, which is a unique morphism guaranteed to exist because C is cartesian closed, to define the morphism $\mathsf{error}_{A,B} : A \longrightarrow B$ that signifies that one tried to unbox or split at the wrong type resulting in a dynamic type error; this is captured by the first and second equations in the definition. If we view morphisms as programs, then the other equations are congruence rules that trigger a dynamic type error for the whole program when one of its subparts trigger a dynamic type error. The following extends the error equations to the functors $- \times -$ and $- \to -$:

Lemma 2.6 (Extended Errors)**.** *Suppose* $(\mathcal{T}, C, ?, \mathsf{T}, \mathsf{split}, \mathsf{squash}, \mathsf{box}, \mathsf{unbox}, \mathsf{error})$ *is a gradual λ-model. Then the following equations hold:*

$$\begin{aligned}
f \times \mathsf{error}_{B,C} &= \mathsf{error}_{A\times B,C\times D}, \textit{ where } f : A \longrightarrow C\\
\mathsf{error}_{A,C} \times f &= \mathsf{error}_{A\times B,C\times D}, \textit{ where } f : B \longrightarrow D\\
f \to \mathsf{error}_{B,C} &= \mathsf{error}_{A\to B,C\to D}, \textit{ where } f : C \longrightarrow A\\
\mathsf{error}_{C,A} \to f &= \mathsf{error}_{A\to B,C\to D}, \textit{ where } f : B \longrightarrow D
\end{aligned}$$

Proof. The following define the morphism part of the two functors $f \times g : (A \times B) \longrightarrow (C \times D)$ and $f \to g : (A \to B) \longrightarrow (C \to D)$:

$$\begin{aligned}
f \times g &= \langle \mathsf{fst}; f, \mathsf{snd}; g\rangle,\\
&\quad \text{where } f : A \longrightarrow C \text{ and } g : B \longrightarrow D\\
f \to g &= \mathsf{curry}((\mathsf{id}_{A\to B} \times f); \mathsf{app}_{A,B}; g),\\
&\quad \text{where } f : C \longrightarrow A \text{ and } g : B \longrightarrow D
\end{aligned}$$

First, note that $\mathsf{fst} : (A \times B) \longrightarrow A$, $\mathsf{snd} : (A \times B) \longrightarrow B$, and $\mathsf{app}_{A,B} : ((A \to B) \times A) \longrightarrow B$ all exist by the definition of a cartesian closed category.

It is now quite obvious that if either f or g is error in the previous two definitions, then by using the equations from the definition of a gradual λ-model (Definition 2.5) the application of either of the functors will result in error. □

As the model is defined it is unclear if we can cast any type to ?, and vice versa, but we must be able to do this in order to model full dynamic typing. In the remainder of this section we show that we can build up such casts in terms of the basic features of our model. To cast any type A to ? we will build casting morphisms that first take the object A to its skeleton, and then takes the skeleton to ?.

Definition 2.7. Suppose $(\mathcal{T}, C, ?, \mathsf{T}, \mathsf{split}, \mathsf{squash}, \mathsf{box}, \mathsf{unbox}, \mathsf{error})$ is a gradual λ-model. Then we call any morphism defined completely in terms of id, the functors $- \times -$ and $- \to -$, split and squash, and box and unbox a **casting morphism**.

Definition 2.8. Suppose $(\mathcal{T}, C, ?, \mathsf{T}, \mathsf{split}, \mathsf{squash}, \mathsf{box}, \mathsf{unbox}, \mathsf{error})$ is a gradual λ-model. The **skeleton** of an object A of C is an object S that is constructed by replacing each atomic type in A with ?. Given an object A we denote its skeleton by skeleton A.

One should think of the skeleton of an object as the supporting type structure of the object, but we do not know what kind of data is actually in the structure. For example, the skeleton of the object Nat is ?, and the skeleton of $(\mathsf{Nat} \times \mathsf{Unit}) \to \mathsf{Nat} \to \mathsf{Nat}$ is $(? \times ?) \to ? \to ?$.

The next definition defines a means of constructing a casting morphism that casts a type A to its skeleton and vice versa. This definition is by mutual recursion on the input type.

Definition 2.9. Suppose $(\mathcal{T}, \mathcal{C}, ?, \mathsf{T}, \mathsf{split}, \mathsf{squash}, \mathsf{box}, \mathsf{unbox}, \mathsf{error})$ is a gradual λ-model. Then for any object A whose skeleton is S we define the morphisms $\widehat{\mathsf{box}}_A : A \longrightarrow S$ and $\widehat{\mathsf{unbox}}_A : S \longrightarrow A$ by mutual recursion on A as follows:

$$
\begin{array}{l|l}
\widehat{\mathsf{box}}_A = \mathsf{box}_A & \widehat{\mathsf{unbox}}_A = \mathsf{unbox}_A \\
\quad \text{when } A \text{ is atomic} & \quad \text{when } A \text{ is atomic} \\
\widehat{\mathsf{box}}_? = \mathsf{id}_? & \widehat{\mathsf{unbox}}_? = \mathsf{id}_? \\
\widehat{\mathsf{box}}_{(A_1 \to A_2)} = \widehat{\mathsf{unbox}}_{A_1} \to \widehat{\mathsf{box}}_{A_2} & \widehat{\mathsf{unbox}}_{(A_1 \to A_2)} = \widehat{\mathsf{box}}_{A_1} \to \widehat{\mathsf{unbox}}_{A_2} \\
\widehat{\mathsf{box}}_{(A_1 \times A_2)} = \widehat{\mathsf{box}}_{A_1} \times \widehat{\mathsf{box}}_{A_2} & \widehat{\mathsf{unbox}}_{(A_1 \times A_2)} = \widehat{\mathsf{unbox}}_{A_1} \times \widehat{\mathsf{unbox}}_{A_2}
\end{array}
$$

The definition of both $\widehat{\mathsf{box}}$ or $\widehat{\mathsf{unbox}}$ uses the functor $- \to - : \mathcal{C}^{\mathrm{op}} \times \mathcal{C} \longrightarrow \mathcal{C}$ which is contravariant in its first argument, and thus, in that contravariant position we must make a recursive call to the opposite function, and hence, they must be mutually defined. Every call to either $\widehat{\mathsf{box}}$ or $\widehat{\mathsf{unbox}}$ in the previous definition is on a smaller object than the input object. Thus, their definitions are well founded. Furthermore, $\widehat{\mathsf{box}}$ and $\widehat{\mathsf{unbox}}$ form a retract between A and S.

Lemma 2.10 (Boxing and Unboxing Lifted Retract)**.** *Suppose* $(\mathcal{T}, \mathcal{C}, ?, \mathsf{T}, \mathsf{split}, \mathsf{squash}, \mathsf{box}, \mathsf{unbox}, \mathsf{error})$ *is a gradual λ-model. Then for any object A,* $\widehat{\mathsf{box}}_A; \widehat{\mathsf{unbox}}_A = \mathsf{id}_A : A \longrightarrow A$*. Furthermore, for any objects A and B such that $A \neq B$,* $\widehat{\mathsf{box}}_A; \widehat{\mathsf{unbox}}_B = \mathsf{error}_{A,B}$.

Proof. This proof holds by induction on the form A. Please see Appendix **??** for the complete proof. □

As an example, suppose we wanted to cast the type $(\mathsf{Nat} \times ?) \to \mathsf{Nat}$ to its skeleton $(? \times ?) \to ?$. Then we can obtain a casting morphisms that will do this as follows:

$$\widehat{\mathsf{box}}_{((\mathsf{Nat} \times ?) \to \mathsf{Nat})} = (\mathsf{unbox}_{\mathsf{Nat}} \times \mathsf{id}_?) \to \mathsf{box}_{\mathsf{Nat}}$$

We can also cast a morphism $A \xrightarrow{f} B$ to a morphism $\widehat{\mathsf{unbox}}_A; f; \widehat{\mathsf{box}}_A : S_1 \longrightarrow S_2$ where $S_1 = \text{skeleton}\, A$ and $S_2 = \text{skeleton}\, B$. Now if we have a second $\widehat{\mathsf{unbox}}_B; g; \widehat{\mathsf{box}}_C : S_2 \longrightarrow S_3$ then their composition reduces to composition at the typed level:

$$
\begin{tikzcd}
S_1 \arrow[r, "\widehat{\mathsf{unbox}}_A"] \arrow[d] & A \arrow[r, "f"] \arrow[d, "f;g"] & B \arrow[r, "\widehat{\mathsf{box}}_B"] \arrow[d, equal] & S_2 \arrow[d, equal] \\
S_3 & C \arrow[l, "\widehat{\mathsf{box}}_C"'] & B \arrow[l, "g"'] & S_2 \arrow[l, "\widehat{\mathsf{unbox}}_B"']
\end{tikzcd}
$$

The right most diagram commutes because B is a retract of S_2, and the left unannotated arrow is the composition $\widehat{\mathsf{unbox}}_A; f; g; \widehat{\mathsf{box}}_C$. This tells us that we have a functor $\mathsf{S} : \mathcal{C} \longrightarrow \mathcal{S}$:

$$
\begin{aligned}
&\mathsf{S}A = \text{skeleton}\, A \\
&\mathsf{S}(f : A \longrightarrow B) = \widehat{\mathsf{unbox}}_A; f; \widehat{\mathsf{box}}_A
\end{aligned}
$$

where $\mathcal{S}$ is the full subcategory of $\mathcal{C}$ consisting of the skeletons and morphisms between them, that is, $\mathcal{S}$ is a cartesian closed category with one basic object ? such that $(\mathcal{S}, ?, \mathsf{split}, \mathsf{squash})$ is an untyped λ-model. The following turns out to be true.

Lemma 2.11 (S is faithful)**.** *Suppose* $(\mathcal{T}, \mathcal{C}, ?, \mathsf{T}, \mathsf{split}, \mathsf{squash}, \mathsf{box}, \mathsf{unbox}, \mathsf{error})$ *is a gradual λ-model, and* $(\mathcal{S}, ?, \mathsf{split}, \mathsf{squash})$ *is the category of skeletons. Then the functor* $\mathsf{S} : \mathcal{C} \longrightarrow \mathcal{S}$ *is faithful.*

Proof. This proof follows from the definition of S and Lemma 2.10. For the full proof see Appendix **??**. □

Thus, we can think of the functor S as an injection of the typed world into the untyped one.

Now that we can cast any type into its skeleton we must show that every skeleton can be cast to ?. We do this similarly to the above and lift split and squash to arbitrary skeletons.

Definition 2.12. Suppose $(\mathcal{S}, ?, \mathsf{split}, \mathsf{squash})$ is the category of skeletons. Then for any skeleton S we define the morphisms $\widehat{\mathsf{squash}}_S : S \longrightarrow ?$ and $\widehat{\mathsf{split}}_S : ? \longrightarrow S$ by mutual recursion on S as follows:

$$
\begin{array}{l}
\widehat{\mathsf{squash}}_? = \mathsf{id}_? \\
\widehat{\mathsf{squash}}_{(S_1 \to S_2)} = (\widehat{\mathsf{split}}_{S_1} \to \widehat{\mathsf{squash}}_{S_2}); \mathsf{squash}_{? \to ?} \\
\widehat{\mathsf{squash}}_{(S_1 \times S_2)} = (\widehat{\mathsf{squash}}_{S_1} \times \widehat{\mathsf{squash}}_{S_2}); \mathsf{squash}_{? \times ?} \\
\hline
\widehat{\mathsf{split}}_? = \mathsf{id}_? \\
\widehat{\mathsf{split}}_{(S_1 \to S_2)} = \mathsf{split}_{? \to ?}; (\widehat{\mathsf{squash}}_{S_1} \to \widehat{\mathsf{split}}_{S_2}) \\
\widehat{\mathsf{split}}_{(S_1 \times S_2)} = \mathsf{split}_{? \times ?}; (\widehat{\mathsf{split}}_{S_1} \times \widehat{\mathsf{split}}_{S_2})
\end{array}
$$

As an example we will construct the casting morphism that casts the skeleton $(? \times ?) \to ?$ to ?:

$$\widehat{\mathsf{squash}}_{(? \times ?) \to ?} = (\mathsf{split}_{? \times ?} \to \mathsf{id}_?); \mathsf{squash}_{? \to ?}.$$

Just as we saw above, splitting and squashing forms a retract.

Lemma 2.13 (Splitting and Squashing Lifted Retract)**.** *Suppose* $(\mathcal{S}, ?, \mathsf{split}, \mathsf{squash})$ *is the category of skeletons. Then for any skeleton S,* $\widehat{\mathsf{squash}}_S; \widehat{\mathsf{split}}_S = \mathsf{id}_S : S \longrightarrow S$*. Furthermore, for any skeletons S_1 and S_2 such that $S_1 \neq S_2$,* $\widehat{\mathsf{squash}}_{S_1}; \widehat{\mathsf{split}}_{S_2} = \mathsf{error}_{S1,S2}$.

Proof. The proof is similar to the proof of the boxing and unboxing lifted retract (Lemma 2.10). □

There is also a faithful functor from $\mathcal{S}$ to $\mathcal{U}$ where $\mathcal{U}$ is the full subcategory of $\mathcal{S}$ that consists of the single object ? and all its morphisms between it:

$$
\begin{aligned}
&\mathsf{U}S = ? \\
&\mathsf{U}(f : S_1 \longrightarrow S_2) = \widehat{\mathsf{split}}_{S_1}; f; \widehat{\mathsf{squash}}_{S_2}
\end{aligned}
$$

This finally implies that there is a functor $\mathsf{C} : \mathcal{C} \longrightarrow \mathcal{U}$ that injects all of $\mathcal{C}$ into the object ?.

$$
\begin{array}{llll}
\text{(types)} & A, B, C & ::= & \mathsf{Unit} \mid \mathsf{Nat} \mid ? \mid A \times B \mid A \to B \\[1ex]
\text{(skeletons)} & S, K, U & ::= & ? \mid S_1 \times S_2 \mid S_1 \to S_2 \\[1ex]
\text{(terms)} & t & ::= & x \mid \mathsf{triv} \mid 0 \mid \mathsf{succ}\; t \mid (t_1, t_2) \mid \mathsf{fst}\; t \mid \mathsf{snd}\; t \\
 & & \mid & \lambda(x : A).t \mid t_1\; t_2 \mid \mathsf{case}\; t \colon \mathsf{Nat}\; \mathsf{of}\; 0 \to t_1, (\mathsf{succ}\; x) \to t_2 \\
 & & \mid & \mathsf{box}_A \mid \mathsf{unbox}_A \mid \mathsf{squash}_S \mid \mathsf{split}_S \mid \mathsf{error}_A \\[2ex]
\text{(contexts)} & \Gamma & ::= & \cdot \mid x : A \mid \Gamma_1, \Gamma_2
\end{array}
$$

Figure 1. Syntax for Core Grady

Lemma 2.14 (Casting to ?)**.** *Suppose* $(\mathcal{T}, \mathcal{C}, ?, \mathsf{T}, \mathsf{split}, \mathsf{squash}, \mathsf{box}, \mathsf{unbox}, \mathsf{error})$ *is a gradual λ-model,* $(\mathcal{S}, ?, \mathsf{split}, \mathsf{squash})$ *is the full subcategory of skeletons, and* $(\mathcal{U}, ?)$ *is the full subcategory containing only ? and its morphisms. Then there is a faithful functor* $\mathsf{C} = \mathcal{C} \xrightarrow{\mathsf{S}} \mathcal{S} \xrightarrow{\mathsf{U}} \mathcal{U}$.

In a way we can think of $\mathsf{C} : \mathcal{C} \longrightarrow \mathcal{U}$ as a forgetful functor. It forgets the type information.

Getting back the typed information is harder. There is no nice functor from $\mathcal{U}$ to $\mathcal{C}$, because we need more information. However, given a type A we can always obtain a casting morphism from ? to A by $(\widehat{\mathsf{split}}_{(\mathsf{skeleton}\; A)}); (\widehat{\mathsf{unbox}_A}) : ? \longrightarrow A$. Finally, we have the following result.

Lemma 2.15 (Casting Morphisms to ?)**.** *Suppose* $(\mathcal{T}, \mathcal{C}, ?, \mathsf{T}, \mathsf{split}, \mathsf{squash}, \mathsf{box}, \mathsf{unbox}, \mathsf{error})$ *is a gradual λ-model, and A is an object of $\mathcal{C}$. Then there exists casting morphisms from A to ? and vice versa that make A a retract of ?.*

Proof. The two morphisms are as follows:

$$
\begin{array}{l}
\mathsf{Box}_A := \widehat{\mathsf{box}_A}; \widehat{\mathsf{squash}}_{(\mathsf{skeleton}\; A)} : A \longrightarrow ? \\
\mathsf{Unbox}_A := \widehat{\mathsf{split}}_{(\mathsf{skeleton}\; A)}; \widehat{\mathsf{unbox}_A} : ? \longrightarrow A
\end{array}
$$

The fact the these form a retract between A and ?, and raise dynamic type errors holds by Lemma 2.10 and Lemma 2.13. □

3 Core Grady

Just as the simply typed λ-calculus corresponds to cartesian closed categories our categorical model has a corresponding type theory we call Core Grady. It consists of all of the structure found in the model. To move from the model to Core Grady we apply the Curry-Howard-Lambek correspondence [8, 17]. Objects become types, and morphisms, $t : \Gamma \longrightarrow A$, become programs in context usually denoted by $\Gamma \vdash_{\mathsf{CG}} t : A$ which corresponds to Core Grady's type checking judgment. We will discuss this correspondence in detail in Section 4.1.

The syntax for Core Grady is defined in Figure 1. The syntax is a straightforward extension of the simply typed λ-calculus. Natural numbers are denoted by 0 and $\mathsf{succ}\; t$ where the latter is the successor of t. The non-recursive natural number eliminator is denoted by $\mathsf{case}\; t \colon \mathsf{Nat}\; \mathsf{of}\; 0 \to t_1, (\mathsf{succ}\; x) \to t_2$. The most interesting aspect of the syntax is that box_A and unbox_A are not restricted to atomic types, but actually correspond to Box_A and Unbox_A from Lemma 2.15. The categorical model shows that these can actually be defined in terms of box_A and unbox_A when A is atomic, but we take the general

$$
\begin{array}{cc}
\dfrac{x : A \in \Gamma}{\Gamma \vdash_{\mathsf{CG}} x : A}\,\mathsf{var} & \dfrac{}{\Gamma \vdash_{\mathsf{CG}} \mathsf{box}_A : A \to ?}\,\mathsf{box} \\[3ex]
\dfrac{}{\Gamma \vdash_{\mathsf{CG}} \mathsf{unbox}_A : ? \to A}\,\mathsf{unbox} & \dfrac{}{\Gamma \vdash_{\mathsf{CG}} \mathsf{squash}_S : S \to ?}\,\mathsf{squash} \\[3ex]
\dfrac{}{\Gamma \vdash_{\mathsf{CG}} \mathsf{split}_S : ? \to S}\,\mathsf{split} & \dfrac{}{\Gamma \vdash_{\mathsf{CG}} \mathsf{triv} : \mathsf{Unit}}\,\mathsf{Unit} \\[3ex]
\dfrac{}{\Gamma \vdash_{\mathsf{CG}} 0 : \mathsf{Nat}}\,\mathsf{zero} & \dfrac{\Gamma \vdash_{\mathsf{CG}} t : \mathsf{Nat}}{\Gamma \vdash_{\mathsf{CG}} \mathsf{succ}\; t : \mathsf{Nat}}\,\mathsf{succ}
\end{array}
$$

$$
\dfrac{\begin{array}{l}\Gamma \vdash_{\mathsf{CG}} t : \mathsf{Nat} \\ \Gamma \vdash_{\mathsf{CG}} t_1 : A \quad \Gamma, x : \mathsf{Nat} \vdash_{\mathsf{CG}} t_2 : A\end{array}}{\Gamma \vdash_{\mathsf{CG}} \mathsf{case}\; t \colon \mathsf{Nat}\; \mathsf{of}\; 0 \to t_1, (\mathsf{succ}\; x) \to t_2 : A}\,\mathsf{Nat}_e
$$

$$
\begin{array}{cc}
\dfrac{\Gamma \vdash_{\mathsf{CG}} t_1 : A_1 \quad \Gamma \vdash_{\mathsf{CG}} t_2 : A_2}{\Gamma \vdash_{\mathsf{CG}} (t_1, t_2) : A_1 \times A_2}\,\times_i & \dfrac{\Gamma \vdash_{\mathsf{CG}} t : A_1 \times A_2}{\Gamma \vdash_{\mathsf{CG}} \mathsf{fst}\; t : A_1}\,\times_{e_1} \\[3ex]
\dfrac{\Gamma \vdash_{\mathsf{CG}} t : A_1 \times A_2}{\Gamma \vdash_{\mathsf{CG}} \mathsf{snd}\; t : A_2}\,\times_{e_2} & \dfrac{\Gamma, x : A \vdash_{\mathsf{CG}} t : B}{\Gamma \vdash_{\mathsf{CG}} \lambda(x : A).t : A \to B}\,\to_i \\[3ex]
\dfrac{\Gamma \vdash_{\mathsf{CG}} t_1 : A \to B \quad \Gamma \vdash_{\mathsf{CG}} t_2 : A}{\Gamma \vdash_{\mathsf{CG}} t_1\; t_2 : B}\,\to_e & \dfrac{}{\Gamma \vdash_{\mathsf{CG}} \mathsf{error}_A : A}\,\mathsf{error}
\end{array}
$$

Figure 2. Typing rules for Core Grady

$$
\begin{array}{cc}
\dfrac{}{\mathsf{unbox}_A\; (\mathsf{box}_A\; t) \rightsquigarrow t}\,\mathsf{retract}_1 & \dfrac{A \neq B}{\mathsf{unbox}_A\; (\mathsf{box}_B\; t) \rightsquigarrow \mathsf{error}_A}\,\mathsf{error} \\[3ex]
\dfrac{t \rightsquigarrow t'}{\mathsf{unbox}_A\; t \rightsquigarrow \mathsf{unbox}_A\; t'}\,\mathsf{unbox} & \dfrac{}{\mathsf{split}_S\; (\mathsf{squash}_S\; t) \rightsquigarrow t}\,\mathsf{retract}_2 \\[3ex]
\dfrac{t \rightsquigarrow t'}{\mathsf{split}_S\; t \rightsquigarrow \mathsf{split}_S\; t'}\,\mathsf{split} & \dfrac{t \rightsquigarrow t'}{\mathsf{succ}\; t \rightsquigarrow \mathsf{succ}\; t'}\,\mathsf{succ}
\end{array}
$$

$$
\dfrac{}{\mathsf{case}\; 0 \colon \mathsf{Nat}\; \mathsf{of}\; 0 \to t_1, (\mathsf{succ}\; x) \to t_2 \rightsquigarrow t_1}\,\mathsf{Nat}_{e_1}
$$

$$
\dfrac{}{\mathsf{case}\; (\mathsf{succ}\; t) \colon \mathsf{Nat}\; \mathsf{of}\; 0 \to t_1, (\mathsf{succ}\; x) \to t_2 \rightsquigarrow [t/x]t_2}\,\mathsf{Nat}_{e_2}
$$

$$
\dfrac{t \rightsquigarrow t'}{\mathsf{case}\; t \colon \mathsf{Nat}\; \mathsf{of}\; 0 \to t_1, (\mathsf{succ}\; x) \to t_2 \rightsquigarrow \mathsf{case}\; t' \colon \mathsf{Nat}\; \mathsf{of}\; 0 \to t_1, (\mathsf{succ}\; x) \to t_2}\,\mathsf{ce}
$$

$$
\begin{array}{ccc}
\dfrac{}{\mathsf{fst}\; (t_1, t_2) \rightsquigarrow t_1}\,\times_{e_1} & \dfrac{}{\mathsf{snd}\; (t_1, t_2) \rightsquigarrow t_2}\,\times_{e_2} & \dfrac{t \rightsquigarrow t'}{\mathsf{fst}\; t \rightsquigarrow \mathsf{fst}\; t'}\,\mathsf{fst}
\end{array}
$$

$$
\begin{array}{cc}
\dfrac{t \rightsquigarrow t'}{\mathsf{snd}\; t \rightsquigarrow \mathsf{snd}\; t'}\,\mathsf{snd} & \dfrac{t_1 \rightsquigarrow t_1'}{(t_1, t_2) \rightsquigarrow (t_1', t_2)}\,\mathsf{pair}_1 \\[3ex]
\dfrac{t_2 \rightsquigarrow t_2'}{(t_1, t_2) \rightsquigarrow (t_1, t_2')}\,\mathsf{pair}_2 & \dfrac{}{(\lambda(x : A_1).t_2)\; t_1 \rightsquigarrow [t_1/x]t_2}\,\beta \\[3ex]
\dfrac{t_1 \rightsquigarrow t_1'}{t_1\; t_2 \rightsquigarrow t_1'\; t_2}\,\mathsf{app}_1 & \dfrac{t_2 \rightsquigarrow t_2'}{v\; t_2 \rightsquigarrow v\; t_2'}\,\mathsf{app}_2
\end{array}
$$

Figure 3. Reduction rules for Core Grady

versions as primitive, because they are the most useful from a programming perspective. The case is similar for split_S and squash_S.

Multisets of pairs of variables and types, denoted by $x : A$, called a typing context or just a context is denoted by Γ. The empty context is denoted by $\cdot$, and the union of contexts Γ_1 and Γ_2 is denoted by Γ_1, Γ_2. Typing contexts are used to keep track of the types of free variables during type checking.

The typing judgment is denoted by $\Gamma \vdash_{\mathsf{CG}} t : A$, and is read "the term t has type A in context Γ." The typing judgment is defined by the type checking rules in Figure 2. The type checking rules are an extension of the typing rules for the simply typed λ-calculus. The casting terms are all typed as axioms with their expected types.

Computing with terms is achieved by defining a reduction relation denoted by $t_1 \rightsquigarrow t_2$ and is read as "the term t_1 reduces in one step to the term t_2." The reduction relation is defined in Figure 3. Reduction for Core Grady differs from the simply typed λ-calculus in that it is an extended formulation of call-by-name. We only allow reduction under unbox and split, and we do not allow reduction under the branches of case. The former insures that when casting progress towards applying the retract rules, $\mathrm{retract}_1$ and $\mathrm{retract}_2$, is always possible. Disallowing reduction in arguments and in the branches of case expressions prevents infinite reductions from occurring without the overall program diverging.

Just as Abadi et al. [1] argue it is quite useful to have access to the untyped λ-calculus. We give some example Core Grady programs utilizing this powerful feature. We have a full implementation of every language in this paper available[1]. All examples in this section can be typed and ran in the implementation, and thus, we make use of Core Grady's concrete syntax which is very similar to Haskell's and does not venture too far from the mathematical syntax given above.

Core Grady does not have a primitive notion of recursion, but it is well-known that we can define the Y combinator in the untyped λ-calculus. Its definition in Core Grady is as follows:

```
omega : (? → ?) → ?
omega = \(x : ? → ?) → (x (squash (? → ?) x));

fix : (? → ?) → ?
fix = \(f : ? → ?) → omega (\(x:?) → f ((split (? → ?) x) x));
```

Using fix we can define the usual arithmetic operations in Core Grady, but we use a typed version of fix.

```
fixNat : ((Nat → Nat) → (Nat → Nat)) → (Nat → Nat)
fixNat = \(f:(Nat → Nat) → (Nat → Nat)) →
   unbox⟨Nat → Nat⟩ (fix (\(y:?)→ box⟨Nat → Nat⟩ (f (unbox⟨Nat → Nat⟩ y))));

pred : Nat → Nat
pred = \(n:Nat) → case n of 0 → 0, (succ n') → n';

add : Nat → Nat → Nat
add = \(m:Nat) → fixNat
    (\(r:Nat → Nat) →
     \(n:Nat) → case n of 0 → m, (succ n') → succ (r n'));

sub : Nat → Nat → Nat
sub = \(m:Nat) → fixNat
    (\(r:Nat → Nat) →
     \(n:Nat) → case n of 0 → m, (succ n') → pred (r n'));

mult : Nat → Nat → Nat
mult = \(m:Nat) → fixNat
    (\(r:Nat → Nat) →
     \(n:Nat) → case n of 0 → 0, (succ n') → add m (r n'));
```

The function fixNat is defined so that it does recursion on the type $\mathsf{Nat} \to \mathsf{Nat}$, thus, it must take in an argument, $f : (\mathsf{Nat} \to \mathsf{Nat}) \to (\mathsf{Nat} \to \mathsf{Nat})$, and produce a function of type $\mathsf{Nat} \to \mathsf{Nat}$. However, we already have fix defined in the untyped fragment, and so, we can define fixNat using fix by boxing up the typed data. This means we must cast $f : (\mathsf{Nat} \to \mathsf{Nat}) \to (\mathsf{Nat} \to \mathsf{Nat})$ into a function of type $(? \to ?) \to ?$ and we do this by η-expanding f and casting the input and output using box and unbox to arrive at the function $\lambda(y : ?).\mathsf{box}_{(\mathsf{Nat}\to\mathsf{Nat})}\ (f\ (\mathsf{unbox}_{(\mathsf{Nat}\to\mathsf{Nat})}\ y)) : ? \to ?$. Finally, we can apply fix, and then unbox its output to the type $\mathsf{Nat} \to \mathsf{Nat}$.

Extending Grady with polymorphism would allow for the definition of fixNat to be abstracted, and then we could do statically typed recursion at any type. We extend Core Grady with bounded polymorphism in Section 6.

From a programming perspective Core Grady has a lot going for it, but it is unfortunate the programmer is required to insert explicit casts when wanting to program dynamically. This implies that it is not possible to program in dynamic style when using Core Grady. In the next section we fix this problem by developing a gradually typed surface language for Core Grady in the spirit of Siek and Taha's gradually typed λ-calculus [13].

Syntax:

$$\begin{array}{lrl}
(\text{terms}) & t ::= & x \mid \mathsf{triv} \mid 0 \mid \mathsf{succ}\ t \mid (t_1, t_2) \mid \mathsf{fst}\ t \mid \mathsf{snd}\ t \\
& \mid & \lambda(x : A).t \mid t_1\ t_2 \mid \mathsf{case}\ t\ \mathsf{of}\ 0 \to t_1, (\mathsf{succ}\ x) \to t_2
\end{array}$$

Metafunctions:

$$\begin{array}{ll}
\mathsf{nat}(?) = \mathsf{Nat} & \mathsf{list}(?) = \mathsf{List}\ ? \\
\mathsf{nat}(\mathsf{Nat}) = \mathsf{Nat} & \mathsf{list}(\mathsf{List}\ A) = \mathsf{List}\ A \\
\\
\mathsf{prod}(?) = ? \times ? & \mathsf{fun}(?) = ? \to ? \\
\mathsf{prod}(A \times B) = A \times B & \mathsf{fun}(A \to B) = A \to B
\end{array}$$

Figure 4. Syntax and Metafunctions for Surface Grady

Typing Rules:

$$\frac{x : A \in \Gamma}{\Gamma \vdash_{SG} x : A}\ \mathsf{var} \qquad \frac{}{\Gamma \vdash_{SG} \mathsf{triv} : \mathsf{Unit}}\ \mathsf{Unit} \qquad \frac{}{\Gamma \vdash_{SG} 0 : \mathsf{Nat}}\ \mathsf{zero}$$

$$\frac{\Gamma \vdash_{SG} t : A \quad \mathsf{nat}(A) = \mathsf{Nat}}{\Gamma \vdash_{SG} \mathsf{succ}\ t : \mathsf{Nat}}\ \mathsf{succ}$$

$$\frac{\begin{array}{c}\Gamma \vdash_{SG} t : C \quad \mathsf{nat}(C) = \mathsf{Nat} \quad \Gamma \vdash A_1 \sim A \\ \Gamma \vdash_{SG} t_1 : A_1 \quad \Gamma, x : \mathsf{Nat} \vdash_{SG} t_2 : A_2 \quad \Gamma \vdash A_2 \sim A\end{array}}{\Gamma \vdash_{SG} \mathsf{case}\ t\ \mathsf{of}\ 0 \to t_1, (\mathsf{succ}\ x) \to t_2 : A}\ \mathsf{Nat}_e$$

$$\frac{\Gamma \vdash_{SG} t_1 : A_1 \quad \Gamma \vdash_{SG} t_2 : A_2}{\Gamma \vdash_{SG} (t_1, t_2) : A_1 \times A_2}\ \times_i$$

$$\frac{\Gamma \vdash_{SG} t : B \quad \mathsf{prod}(B) = A_1 \times A_2}{\Gamma \vdash_{SG} \mathsf{fst}\ t : A_1}\ \times_{e_1}$$

$$\frac{\Gamma \vdash_{SG} t : B \quad \mathsf{prod}(B) = A_1 \times A_2}{\Gamma \vdash_{SG} \mathsf{snd}\ t : A_2}\ \times_{e_2}$$

$$\frac{\Gamma, x : A \vdash_{SG} t : B}{\Gamma \vdash_{SG} \lambda(x : A).t : A \to B}\ \to_i$$

$$\frac{\begin{array}{c}\Gamma \vdash_{SG} t_1 : C \\ \Gamma \vdash_{SG} t_2 : A_2 \quad \Gamma \vdash A_2 \sim A_1 \quad \mathsf{fun}(C) = A_1 \to B_1\end{array}}{\Gamma \vdash_{SG} t_1\ t_2 : B_1}\ \to_e$$

Type Consistency:

$$\frac{}{\Gamma \vdash A \sim A}\ \mathsf{refl} \qquad \frac{}{\Gamma \vdash A \sim ?}\ \mathsf{box} \qquad \frac{}{\Gamma \vdash ? \sim A}\ \mathsf{unbox}$$

$$\frac{\Gamma \vdash A_2 \sim A_1 \quad \Gamma \vdash B_1 \sim B_2}{\Gamma \vdash (A_1 \to B_1) \sim (A_2 \to B_2)}\ \to \qquad \frac{\Gamma \vdash A_1 \sim A_2 \quad \Gamma \vdash B_1 \sim B_2}{\Gamma \vdash (A_1 \times B_1) \sim (A_2 \times B_2)}\ \times$$

Figure 5. Typing rules for Surface Grady

[1] Please see https://ct-gradual-typing.github.io/Grady/ for access to the implementation as well as full documentation on how to install and use it.

$$\frac{\Gamma \vdash t_1 \Rightarrow t_2 : ?}{\Gamma \vdash \mathsf{succ}\ t_1 \Rightarrow \mathsf{succ}\ (\mathsf{unbox}_{\mathsf{Nat}}\ t_2) : \mathsf{Nat}} \qquad \frac{\Gamma \vdash t_1 \Rightarrow t_2 : \mathsf{Nat}}{\Gamma \vdash \mathsf{succ}\ t_1 \Rightarrow \mathsf{succ}\ t_2 : \mathsf{Nat}}$$

$$\frac{\Gamma \vdash t_1 \Rightarrow t_2 : ?}{\Gamma \vdash \mathsf{fst}\ t_1 \Rightarrow \mathsf{fst}\ (\mathsf{split}_{(? \times ?)}\ t_2) : ?} \qquad \frac{\Gamma \vdash t_1 \Rightarrow t_2 : A_1 \times A_2}{\Gamma \vdash \mathsf{fst}\ t_1 \Rightarrow \mathsf{fst}\ t_2 : A_1}$$

$$\frac{\Gamma \vdash t_1 \Rightarrow t_2 : ?}{\Gamma \vdash \mathsf{snd}\ t_1 \Rightarrow \mathsf{snd}\ (\mathsf{split}_{(? \times ?)}\ t_2) : ?} \qquad \frac{\Gamma \vdash t_1 \Rightarrow t_2 : A \times B}{\Gamma \vdash \mathsf{snd}\ t_1 \Rightarrow \mathsf{snd}\ t_2 : B}$$

$$\frac{\begin{array}{c}\Gamma \vdash t_1 \Rightarrow t_1' : ? \\ \Gamma \vdash t_2 \Rightarrow t_2' : A_2 \quad \mathsf{caster}(A_2, ?) = c\end{array}}{\Gamma \vdash t_1\ t_2 \Rightarrow (\mathsf{split}_{(? \to ?)}\ t_1')\ (c\ t_2') : ?}$$

$$\frac{\begin{array}{l}\Gamma \vdash t_2 \Rightarrow t_2' : A_2 \\ \Gamma \vdash t_1 \Rightarrow t_1' : A_1 \to B \quad \Gamma \vdash A_2 \sim A_1 \quad \mathsf{caster}(A_2, A_1) = c\end{array}}{\Gamma \vdash t_1\ t_2 \Rightarrow t_1'\ (c\ t_2') : B}$$

Figure 6. Cast Insertion Algorithm

Type Precision:

$$\frac{}{A \sqsubseteq ?}\,? \qquad \frac{}{A \sqsubseteq A}\,\mathrm{refl} \qquad \frac{A \sqsubseteq C \quad B \sqsubseteq D}{(A \to B) \sqsubseteq (C \to D)}\,\to$$

$$\frac{A \sqsubseteq C \quad B \sqsubseteq D}{(A \times B) \sqsubseteq (C \times D)}\,\times$$

Context Precision:

$$\frac{}{\Gamma \sqsubseteq \Gamma}\,\mathrm{refl} \qquad \frac{\Gamma_1 \sqsubseteq \Gamma_2 \quad A \sqsubseteq A' \quad \Gamma_3 \sqsubseteq \Gamma_4}{\Gamma_1, x : A, \Gamma_3 \sqsubseteq \Gamma_2, x : A', \Gamma_4}\,\mathrm{ext}$$

Figure 7. Type and Context Precision

Term Precision for Surface Grady:

$$\frac{}{t \sqsubseteq t}\,\mathrm{refl} \qquad \frac{t_1 \sqsubseteq t_2}{(\mathsf{succ}\ t_1) \sqsubseteq (\mathsf{succ}\ t_2)}\,\mathrm{succ}$$

$$\frac{t_1 \sqsubseteq t_4 \quad t_2 \sqsubseteq t_5 \quad t_3 \sqsubseteq t_6}{(\mathsf{case}\ t_1\ \mathsf{of}\ 0 \to t_2, (\mathsf{succ}\ x) \to t_3) \sqsubseteq (\mathsf{case}\ t_4\ \mathsf{of}\ 0 \to t_5, (\mathsf{succ}\ x) \to t_6)}\,\mathsf{Nat}$$

$$\frac{t_1 \sqsubseteq t_3 \quad t_2 \sqsubseteq t_4}{(t_1, t_2) \sqsubseteq (t_3, t_4)}\,\times_i \qquad \frac{t_1 \sqsubseteq t_2}{(\mathsf{fst}\ t_1) \sqsubseteq (\mathsf{fst}\ t_2)}\,\times_{e_1}$$

$$\frac{t_1 \sqsubseteq t_2}{(\mathsf{snd}\ t_1) \sqsubseteq (\mathsf{snd}\ t_2)}\,\times_{e_2} \qquad \frac{t_1 \sqsubseteq t_2 \quad A_1 \sqsubseteq A_2}{(\lambda(x : A_1).t) \sqsubseteq (\lambda(x : A_2).t_2)}\,\to_i$$

$$\frac{t_1 \sqsubseteq t_3 \quad t_2 \sqsubseteq t_4}{(t_1\ t_2) \sqsubseteq (t_3\ t_4)}\,\to_2$$

Term Precision for Core Grady:

$$\frac{\Gamma \vdash_{CG} t : ?}{\Gamma \vdash (\mathsf{unbox}_A\ t) \sqsubseteq t}\,\mathsf{box} \qquad \frac{\Gamma \vdash_{CG} t : A}{\Gamma \vdash t \sqsubseteq (\mathsf{box}_A\ t)}\,\mathsf{unbox}$$

$$\frac{\Gamma \vdash_{CG} t : ?}{\Gamma \vdash (\mathsf{split}_S\ t) \sqsubseteq t}\,\mathsf{split} \qquad \frac{\Gamma \vdash_{CG} t : S}{\Gamma \vdash t \sqsubseteq (\mathsf{squash}_S\ t)}\,\mathsf{squash}$$

$$\frac{\Gamma \vdash_{CG} t : B \quad A \sqsubseteq B}{\Gamma \vdash \mathsf{error}_A \sqsubseteq t}\,\mathsf{error}$$

Figure 8. Term Precision

Subtyping for Core Grady:

$$\frac{}{\Gamma \vdash A <: A}\,\mathrm{refl} \qquad \frac{}{\Gamma \vdash A <: \top}\,\mathrm{top} \qquad \frac{X <: A \in \Gamma}{\Gamma \vdash X <: A}\,\mathrm{var}$$

$$\frac{}{\Gamma \vdash \top <: \mathbb{S}}\,\top_{\mathcal{S}} \qquad \frac{}{\Gamma \vdash \mathsf{Nat} <: \mathbb{S}}\,\mathsf{Nat}_{\mathcal{S}} \qquad \frac{}{\Gamma \vdash \mathsf{Unit} <: \mathbb{S}}\,\mathsf{Unit}_{\mathcal{S}}$$

$$\frac{\Gamma \vdash A <: \mathbb{S}}{\Gamma \vdash \mathsf{List}\ A <: \mathbb{S}}\,\mathsf{List}_{\mathcal{S}} \qquad \frac{\Gamma \vdash A <: \mathbb{S} \quad \Gamma \vdash B <: \mathbb{S}}{\Gamma \vdash A \to B <: \mathbb{S}}\,\to_{\mathcal{S}}$$

$$\frac{\Gamma \vdash A <: \mathbb{S} \quad \Gamma \vdash B <: \mathbb{S}}{\Gamma \vdash A \times B <: \mathbb{S}}\,\times_{\mathcal{S}} \qquad \frac{\Gamma \vdash A <: B}{\Gamma \vdash \mathsf{List}\ A <: \mathsf{List}\ B}\,\mathsf{List}$$

$$\frac{\Gamma \vdash A_1 <: A_2 \quad \Gamma \vdash B_1 <: B_2}{\Gamma \vdash A_1 \times B_1 <: A_2 \times B_2}\,\times \qquad \frac{\Gamma \vdash A_2 <: A_1 \quad \Gamma \vdash B_1 <: B_2}{\Gamma \vdash A_1 \to B_1 <: A_2 \to B_2}\,\to$$

$$\frac{\Gamma, X <: A \vdash B_1 <: B_2}{\Gamma \vdash \forall(X <: A).B_1 <: \forall(X <: A).B_2}\,\forall$$

Consistent Subtyping for Surface Grady:

$$\frac{}{\Gamma \vdash A \lesssim A}\,\mathrm{refl} \qquad \frac{}{\Gamma \vdash A \lesssim \top}\,\mathrm{top} \qquad \frac{X <: A' \in \Gamma \quad \Gamma \vdash A' \sim A}{\Gamma \vdash X \lesssim A}\,\mathrm{var}$$

$$\frac{\Gamma \vdash A \lesssim \mathbb{S}}{\Gamma \vdash A \lesssim ?}\,\mathsf{box} \qquad \frac{\Gamma \vdash A \lesssim \mathbb{S}}{\Gamma \vdash ? \lesssim A}\,\mathsf{unbox} \qquad \frac{}{\Gamma \vdash ? \lesssim \mathbb{S}}\,?_{\mathcal{S}}$$

$$\frac{}{\Gamma \vdash \top \lesssim \mathbb{S}}\,\top_{\mathcal{S}} \qquad \frac{}{\Gamma \vdash \mathsf{Nat} \lesssim \mathbb{S}}\,\mathsf{Nat}_{\mathcal{S}} \qquad \frac{}{\Gamma \vdash \mathsf{Unit} \lesssim \mathbb{S}}\,\mathsf{Unit}_{\mathcal{S}}$$

$$\frac{\Gamma \vdash A \lesssim \mathbb{S}}{\Gamma \vdash \mathsf{List}\ A \lesssim \mathbb{S}}\,\mathsf{List}_{\mathcal{S}} \qquad \frac{\Gamma \vdash A \lesssim \mathbb{S} \quad \Gamma \vdash B \lesssim \mathbb{S}}{\Gamma \vdash A \times B \lesssim \mathbb{S}}\,\times_{\mathcal{S}}$$

$$\frac{\Gamma \vdash A \lesssim \mathbb{S} \quad \Gamma \vdash B \lesssim \mathbb{S}}{\Gamma \vdash A \to B \lesssim \mathbb{S}}\,\to_{\mathcal{S}} \qquad \frac{\Gamma \vdash A \lesssim B}{\Gamma \vdash (\mathsf{List}\ A) \lesssim (\mathsf{List}\ B)}\,\mathsf{List}$$

$$\frac{\Gamma \vdash A_1 \lesssim A_2 \quad \Gamma \vdash B_1 \lesssim B_2}{\Gamma \vdash (A_1 \times B_1) \lesssim (A_2 \times B_2)}\,\times \qquad \frac{\Gamma \vdash A_2 \lesssim A_1 \quad \Gamma \vdash B_1 \lesssim B_2}{\Gamma \vdash (A_1 \to B_1) \lesssim (A_2 \to B_2)}\,\to$$

$$\frac{\Gamma, X <: A \vdash B_1 \lesssim B_2}{\Gamma \vdash (\forall(X <: A).B_1) \lesssim (\forall(X <: A).B_2)}\,\forall$$

Figure 9. Subtyping for Core and Surface Grady

4 Surface Grady

4.1 Interpreting Surface Grady in the Model

5 The Gradual Guarantee

6 Bounded Quantification

7 Conclusion

References

[1] M. Abadi, L. Cardelli, B. Pierce, and G. Plotkin. 1989. Dynamic Typing in a Statically-typed Language. In *Proceedings of the 16th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL '89)*. ACM, New York, NY, USA, 213–227.

[2] Stephen Brookes, Peter W. O'Hearn, and Uday Reddy. 2014. The Essence of Reynolds. *POPL '14* (January 2014).

[3] Roy L. Crole. 1994. *Categories for Types.* Cambridge University Press. DOI: http://dx.doi.org/10.1017/CBO9781139172707

[4] Ronald Garcia, Alison M. Clark, and Éric Tanter. 2016. Abstracting Gradual Typing. In *Proceedings of the 43rd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL '16)*. ACM, New York, NY, USA, 429–442.

[5] Jean-Yves Girard, Yves Lafont, and Paul Taylor. 1989. *Proofs and Types (Cambridge Tracts in Theoretical Computer Science).* Cambridge University Press.

[6] Fritz Henglein. 1994. Dynamic typing: syntax and proof theory. *Science of Computer Programming* 22, 3 (1994), 197 – 230.

[7] W. A. Howard. 1980. The Formulae-as-Types Notion of Construction. *To H. B. Curry: Essays on Combznatory Logic, Lambda-Calculus, and Formalism* (1980), 479–490.

[8] Joachim Lambek. 1980. From lambda calculus to Cartesian closed categories. *To H. B. Curry: Essays on Combinatory Logic, Lambda Calculus and Formalism* (1980), 376–402.

[9] J. Lambek and P.J. Scott. 1988. *Introduction to Higher-Order Categorical Logic.* Cambridge University Press.

[10] Eugenio Moggi. 1989. Notions of Computation and Monads. *Information and Computation* 93 (1989), 55–92.

[11] Dana Scott. 1980. Relating Theories of the lambda-Calculus. In *To H.B. Curry: Essays on Combinatory Logic, Lambda-Calculus and Formalism (eds. Hindley and Seldin).* Academic Press, 403–450.

[12] Peter Sewell, Francesco Zappa Nardelli, Scott Owens, Gilles Peskine, Thomas Ridge, Susmit Sarkar, and Rok Strnisa. 2010. Ott: Effective tool support for the working semanticist. In *Journal of Functional Programming (JFP)*, Vol. 20. 71–122.

[13] Jeremy G Siek and Walid Taha. 2006. Gradual typing for functional languages. In *Scheme and Functional Programming Workshop (1)*, Vol. 6. 81–92.

[14] Jeremy G. Siek, Michael M. Vitousek, Matteo Cimini, and John Tang Boyland. 2015. Refined Criteria for Gradual Typing. In *1st Summit on Advances in Programming Languages (SNAPL 2015) (Leibniz International Proceedings in Informatics (LIPIcs))*, Thomas Ball, Rastislav Bodik, Shriram Krishnamurthi, Benjamin S. Lerner, and Greg Morrisett (Eds.), Vol. 32. Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik, Dagstuhl, Germany, 274–293.

[15] The Univalent Foundations Program. 2013. *Homotopy Type Theory: Univalent Foundations of Mathematics.* https://homotopytypetheory.org/book, Institute for Advanced Study.

[16] Philip Wadler. 1995. *Monads for functional programming.* Springer Berlin Heidelberg, Berlin, Heidelberg, 24–52.

[17] Philip Wadler. 2015. Propositions As Types. *Commun. ACM* 58, 12 (Nov. 2015), 75–84. DOI: http://dx.doi.org/10.1145/2699407