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. In addition, this model formally connects gradual typing to the seminal work of Scott and Lambek who showed that the untyped λ-calculus can be considered as typed using retracts, and that the type λ-calculus can be modeled in a cartesian closed category respectively. Following this we extract from our model a new simple type system which combines static and dynamic typing called Core Grady. Then we develop a gradually typed surface language for Core Grady, and show that it can be translated into the core such that the gradual guarantee holds. In addition, to show that the wider area of gradual type systems can benefit from our model we show that Siek and Taha's gradual simply typed λ-calculus can be modeled by the proposed semantics. Finally, while a gradual type system allows for type casts to be left implicit we show that explicit casts can be derived in the gradually typed surface language, and using the explicit casts we show that more programs can be typed. For example, we define a typed fixpoint operator that can only be defined due to the explicit casts in the gradually typed surface language.

CCS Concepts •**Theory of computation → Denotational semantics; Categorical semantics;** *Type theory; Functional constructs; Type structures;*

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

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),* 21 pages.
DOI: 10.1145/nnnnnnn.nnnnnnn

1 Introduction

Gradual typing [23, 24] is a way to combine static and dynamic typing within the same language. For example, one can structure their programs so that the safety critical parts are as statically typed as possible to catch the most errors at compile time, while rapidly prototyping other parts of their program using dynamic typing.

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

The design of a gradual type system consists of two languages: a core language and a surface language. Both languages start as a static type system with a new type called the unknown type, denoted in this paper by ?, whose inhabitants are all untyped programs. The core language then has an explicit mechanism for casting types to and from the unknown type, and an operational semantics. However, the surface language only consists of a type checking algorithm that is designed so that casts are left implicit, and then after type checking succeeds a surface language program is translated into the core language by a cast insertion algorithm.

Programming in the surface language requires the ability to implicitly cast data between consistent types during eliminations. For example, $(\lambda(x : ?).(\text{succ}\ (\text{succ}\ x)))\ 3$ should type check with type Nat even though 3 has type Nat and x has type ?. Now not every cast should work, for example, $(\lambda(x : \text{Bool}).t)\ 3$ should not type check, because it is inconsistent to allow different atomic types to be cast between each other. Therefore, the surface language must be able to decide which casts are consistent and which are not. This is done by extending the type checking algorithm with a binary relation called *type consistency* which determines which types are castable between each other, e.g. every type will be consistent with the unknown type.

Gradual type systems must satisfy the metatheoretic property called the *gradual guarantee*. The gradual guarantee states that any well-typed program can slide between being more statically typed or being more dynamically typed by inserting or removing casts without changing the meaning or the behavior of the program. The formal statement of the gradual guarantee is given in Section 7. This property was first proposed by Siek et al. [24] to set apart systems that simply combine dynamic and static typing and gradual type systems.

Just as Siek et al. report [22–24] there are a number of programming languages that combine static and dynamic typing with implicit casting to and from the unknown type, for example, Boo [7], Bigloo [3, 20], Cecil [5], Visual Basic .Net, C# [16], Professor J [9], and many more. In addition, there are languages that combine static and dynamic typing with explicit casting only. Abadi et al. [1] combine dynamic and static typing by adding a new type called Dynamic along with a new case construct for pattern matching on types, and Henglein [10] defines the dynamic λ-calculus by adding a new type Dyn to the simply typed λ-calculus and then adding primitive casting operations called tagging and check-and-untag. Please see the introduction to Siek et al. [24] for a more complete list.

As we can see the combination of dynamic and static typing, as well as gradual type systems, are important to both industry and academia. Thus, expanding gradual type systems with new features is an increasingly important line of research [8, 12, 15, 22, 23][1]. Therefore, it is of the utmost importance that we be able to formally reason about gradual type systems so as to insure new designs and extensions are correct. However, there are no categorical models, in fact not many mathematical models at all, to aid in the design of new gradual type systems or their extensions.

In this paper we propose a new categorical model of the core casting calculus of gradual type systems based on the seminal work of Scott [21] and Lambek [14], thus merging the categorical model of the statically typed λ-calculus with the categorical model of the untyped λ-calculus into a new model that combines the two. Our categorical model leads to a new and simple type theory that combines dynamic and static typing with explicit casts called Core Grady. Furthermore, it is has a less complex language design compared to existing core languages for gradual typing, for example, Siek and Taha's system [22], because it does not depend on type consistency where theirs does. Our new model also gives a rigorous framework that can be used while developing new gradual type systems or extensions of existing gradual type systems.

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 [13]; on the syntactic side these two theories are equivalent as well due to Howard [11]; 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 programming 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 [4] was a big advocate for the use of category theory in programming language research for this reason. We agree with Brookes et al. (from p. 3 of [4]) that the following quote, originally from [18], 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 [17, 26] on using monads – a purely categorical notion – to add side effects to pure functional programming languages. Thus, we believe that developing these types of models for new language designs and features can be hugely beneficial.

1.1 Overview

We now give a brief overview of our main results, but from a typed λ-calculus perspective, but we will transition to category theory in Section 2. Suppose we add the unknown type, ?, and two functions $\mathsf{squash} : (? \to ?) \to ?$ and $\mathsf{split} : ? \to (? \to ?)$ to the simply typed λ-calculus with the natural numbers. Furthermore, we require that for any program, t, of type $? \to ?$, we have $\mathsf{split}\ (\mathsf{squash}\ t) \rightsquigarrow t$. Categorically split and squash define what is called a retract. Scott [19] showed that this is enough to encode the untyped λ-calculus into a statically typed setting:

$$\begin{array}{rclrcl} \lceil x \rceil & = & x & \lceil x\ t_2 \rceil & = & (\mathsf{split}\ x)\ \lceil t_2 \rceil \\ \lceil \lambda x.t \rceil & = & \lambda(x : ?).\lceil t \rceil & \lceil t_1\ t_2 \rceil & = & \lceil t_1 \rceil\ (\mathsf{squash}\ \lceil t_2 \rceil) \end{array}$$

For example, if $\Omega = \lambda x.x\ x$, then $|\Omega| = \lambda(x : ?).(\mathsf{split}\ x)\ x$, and $|\Omega\ \Omega| = |\Omega|\ (\mathsf{squash}\ |\Omega|)$ is the typical diverging term.

We have at this point a typed functional programming language with two fragments: the statically typed λ-calculus and the untyped λ-calculus. However, they are just sitting side-by-side. Now suppose for any atomic type A, excluding the unknown type, we add two new functions $\mathsf{box}_A : A \to ?$ and $\mathsf{unbox}_A : ? \to A$ such that for any term, t, of atomic type A, we have that $\mathsf{unbox}_A\ (\mathsf{box}_A\ t) \rightsquigarrow t$ – a second retract. This defines the bridge between the typed fragment and the untyped fragment. We will show in the next section that both box and unbox can be generalized to arbitrary types, and thus, they will subsume split and squash as well, hence, reducing all explicit casts to just two functions simplifying the language even further – in the rest of this section we only use box and unbox.

At this point we have basically built up Core Grady (Section 3) the corresponding type theory to our categorical model (Section 2). We can move statically typed data in between the two fragments. An example may help solidify the previous point.

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, and hence, in Core Grady. First, we have a full implementation of every language in this paper available[2]. All examples in this paper can be typed and ran in the implementation, and thus, we make use of Core Grady's

[1] There are even more examples in the list of accepted papers for ICFP 2017, for example, extending gradual type systems with session types and polymorphism.

[2] 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.

concrete syntax which is very similar to Haskell's and does not venture too far from the mathematical syntax we will introduce in Section 3.

The definition of the Y combinator in Core Grady is as follows:

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

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

Using fix we can define the usual arithmetic operations in Core Grady, but we use a typed version of fix – we have developed many more examples in Core Grady please see the implementation (Footnote 2).

```
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))));

add : Nat → Nat → Nat
add = \(m:Nat) → fixNat
     (\( r:Nat → Nat) →
      \(n:Nat) → case n of 0 → m, (succ n') → succ (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 Nat → Nat, thus, it must take in an argument, f : (Nat → Nat) → (Nat → Nat), and produce a function of type Nat → 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 : (Nat → Nat) → (Nat → Nat) into a function of type (? → ?) → ? and we do this by η-expanding f and casting the input and output using box and unbox to arrive at the function $\lambda(y : ?).\text{box}_{(\text{Nat}\to\text{Nat})}\ (f\ (\text{unbox}_{(\text{Nat}\to\text{Nat})}\ y)) : ? \to ?$. Finally, we can apply fix , and then unbox its output to the type Nat → Nat. Thus, the definition of fixNat moves typed data to the untyped fragment using box and then moves it back using unbox.

The terms box and unbox correspond to explicit casts. In Section 4 we develop a gradually typed surface language called Surface Grady that allows for the casts to be left implicit. Then we show that both Surface and Core Grady can be soundly interpreted into our categorical model in Section 5. Furthermore, we prove the gradual guarantee for Surface and Core Grady in Section 7.

To insure that our categorical model can be used to study other gradual type systems we show that Siek and Taha's gradually typed λ-calculus [23, 24] can also be soundly modeled by our semantics in Section 6.

Finally, while a gradual type system allows for type casts to be left implicit we show, in Section 8, that explicit casts can be derived in the gradually typed surface language, and using the explicit casts we show that more programs can be typed. For example, we define a typed fixpoint operator that can only be defined due to the explicit casts in the gradually typed surface language.

2 The Categorical Model

The model we develop here builds on the seminal work of Lambek [13] and Scott [19]. Lambek [13] 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 object – or type – ?, and two morphisms $\text{squash} : (? \to ?) \to ?$ and $\text{split} : ? \to (? \to ?)$, such that, $\text{squash}; \text{split} = \text{id} : (? \to ?) \to (? \to ?)$, to a cartesian closed category[3]. 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.

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 $i; r = \text{id}_A$.

Thus, $? \to ?$ is a retract of ?, but we also require that $? \times ?$ be a retract of ?; this is not new, see Lambek and Scott [14]. Putting this together we obtain Scott's model of the untyped λ-calculus.

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

Theorem 2.3 (Scott [19])**.** *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). 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 [14].

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 $\text{z} : 1 \longrightarrow \text{Nat}$ and $\text{succ} : \text{Nat} \longrightarrow \text{Nat}$ of C, such that, for any morphisms $f : Y \longrightarrow X$ and $g : Y \times \text{Nat} \longrightarrow X$ of C there is an unique morphism $\text{case}_{Y,X}\langle f, g\rangle : Y \times \text{Nat} \longrightarrow X$ such that the following hold:

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

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

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

[3]We use diagrammatic notation for composition of morphisms. If $f : A \longrightarrow B$ and $g : B \longrightarrow C$, then their composition is denoted by $f; g : A \longrightarrow C$.

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**, $(\mathcal{T}, \mathcal{C}, ?, \mathsf{T}, \mathsf{split}, \mathsf{squash}, \mathsf{box}, \mathsf{unbox}, \mathsf{error})$, where $\mathcal{T}$ is a discrete category with at least two objects Nat and Unit, $\mathcal{C}$ is a cartesian closed category with an NRNO, $(\mathcal{C}, ?, \mathsf{split}, \mathsf{squash})$ is an untyped λ-model, $\mathsf{T} : \mathcal{T} \longrightarrow \mathcal{C}$ 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 $\mathcal{C}$, 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 $\mathcal{C}$ 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 $(\mathcal{T}, \mathcal{C}, ?, \mathsf{T}, \mathsf{split}, \mathsf{squash}, \mathsf{box}, \mathsf{unbox}, \mathsf{error})$ is a gradual λ-model. Then the following equations hold:*

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

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

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

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

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:

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

The definition of both $\widehat{\mathsf{box}}$ or $\widehat{\mathsf{unbox}}$ use the functor $- \to - : \mathcal{C}^{\mathsf{op}} \times \mathcal{C} \longrightarrow \mathcal{C}$ 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.

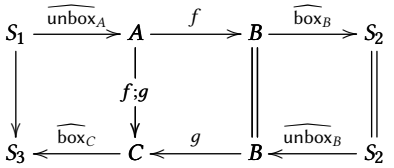
Lemma 2.10 (Boxing and Unboxing Lifted Retract)**.** *Suppose* $(\mathcal{T}, C, ?, \mathsf{T}, \mathsf{split}, \mathsf{squash}, \mathsf{box}, \mathsf{unbox}, \mathsf{error})$ *is a gradual* λ*-model. Then for any object* A, $\widehat{\mathsf{box}}_A; \widehat{\mathsf{unbox}}_A = \mathsf{id}_A : A \longrightarrow A$. *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 B.1 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 $S_1 = \mathsf{skeleton}\, A$ and $S_2 = \mathsf{skeleton}\, B$. 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:



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 $\mathsf{S} : C \longrightarrow \mathcal{S}$:

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

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

Proof. This proof follows from the definition of S and Lemma 2.10. For the full proof see Appendix B.2. □

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:

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

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

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 $\mathsf{C} : C \longrightarrow \mathcal{U}$ that injects all of C into the object ?.

Lemma 2.14 (Casting to ?)**.** *Suppose* $(\mathcal{T}, C, ?, \mathsf{T}, \mathsf{split}, \mathsf{squash}, \mathsf{box}, \mathsf{unbox}, \mathsf{error})$ *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* $\mathsf{C} = C \xrightarrow{\mathsf{S}} \mathcal{S} \xrightarrow{\mathsf{U}} \mathcal{U}$.

In a way we can think of $\mathsf{C} : C \longrightarrow \mathcal{U}$ 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 C, because we need more information. However, given a type A we can always obtain a casting morphism from ? to A by $(\widehat{\mathsf{split}}_{(\mathsf{skeleton}\, A)}); (\widehat{\mathsf{unbox}}_A) : ? \longrightarrow A$. Therefore, we have the following result.

Lemma 2.15 (Casting Morphisms to ?)**.** *Suppose* $(\mathcal{T}, C, ?, \mathsf{T}, \mathsf{split}, \mathsf{squash}, \mathsf{box}, \mathsf{unbox}, \mathsf{error})$ *is a gradual* λ*-model, and* A *is an object of* 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:

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

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

$$
\begin{array}{lll}
\text{(types)} & A, B, C ::= \mathsf{Unit} \mid \mathsf{Nat} \mid ? \mid A \times B \mid A \to B \\
\text{(skeletons)} & S, K, U ::= ? \mid S_1 \times S_2 \mid S_1 \to S_2 \\
\text{(terms)} & t ::= x \mid \mathsf{triv} \mid 0 \mid \mathsf{succ}\, t \mid (t_1, t_2) \mid \mathsf{fst}\, t \mid \mathsf{snd}\, t \\
 & \quad \mid \lambda(x : A).t \mid t_1\, t_2 \mid \mathsf{case}\, t\colon \mathsf{Nat\ of}\ 0 \to t_1, (\mathsf{succ}\, x) \to t_2 \\
 & \quad \mid \mathsf{box}_A \mid \mathsf{unbox}_A \mid \mathsf{error}_A \\
\text{(values)} & v ::= \lambda(x : A).t \\
\text{(evaluation contexts)} & \mathcal{E} ::= \Box\, t_2 \mid \mathsf{unbox}_A\, \Box \mid \mathsf{succ}\, \Box \mid \mathsf{fst}\, \Box \mid \mathsf{snd}\, \Box \mid (\Box, t) \\
 & \quad \mid (t, \Box) \mid \mathsf{case}\, \Box\colon \mathsf{Nat\ of}\ 0 \to t_1, (\mathsf{succ}\, x) \to t_2 \\
\text{(contexts)} & \Gamma ::= \cdot \mid x : A \mid \Gamma_1, \Gamma_2
\end{array}
$$

Figure 1. Syntax for Core Grady

$$
\frac{x : A \in \Gamma}{\Gamma \vdash_{\mathsf{CG}} x : A}\,\mathsf{var} \qquad \frac{}{\Gamma \vdash_{\mathsf{CG}} \mathsf{box}_A : A \to ?}\,\mathsf{box}
$$

$$
\frac{}{\Gamma \vdash_{\mathsf{CG}} \mathsf{unbox}_A : ? \to A}\,\mathsf{unbox} \qquad \frac{}{\Gamma \vdash_{\mathsf{CG}} \mathsf{triv} : \mathsf{Unit}}\,\mathsf{Unit}
$$

$$
\frac{}{\Gamma \vdash_{\mathsf{CG}} 0 : \mathsf{Nat}}\,\mathsf{zero} \qquad \frac{\Gamma \vdash_{\mathsf{CG}} t : \mathsf{Nat}}{\Gamma \vdash_{\mathsf{CG}} \mathsf{succ}\, t : \mathsf{Nat}}\,\mathsf{succ}
$$

$$
\frac{\Gamma \vdash_{\mathsf{CG}} t : \mathsf{Nat} \quad \Gamma \vdash_{\mathsf{CG}} t_1 : A \quad \Gamma, x : \mathsf{Nat} \vdash_{\mathsf{CG}} t_2 : A}{\Gamma \vdash_{\mathsf{CG}} \mathsf{case}\, t\colon \mathsf{Nat\ of}\ 0 \to t_1, (\mathsf{succ}\, x) \to t_2 : A}\,\mathsf{Nat}_e
$$

$$
\frac{\Gamma \vdash_{\mathsf{CG}} t_1 : A_1 \quad \Gamma \vdash_{\mathsf{CG}} t_2 : A_2}{\Gamma \vdash_{\mathsf{CG}} (t_1, t_2) : A_1 \times A_2}\times_i \qquad \frac{\Gamma \vdash_{\mathsf{CG}} t : A_1 \times A_2}{\Gamma \vdash_{\mathsf{CG}} \mathsf{fst}\, t : A_1}\times_{e_1}
$$

$$
\frac{\Gamma \vdash_{\mathsf{CG}} t : A_1 \times A_2}{\Gamma \vdash_{\mathsf{CG}} \mathsf{snd}\, t : A_2}\times_{e_2} \qquad \frac{\Gamma, x : A \vdash_{\mathsf{CG}} t : B}{\Gamma \vdash_{\mathsf{CG}} \lambda(x : A).t : A \to B}\to_i
$$

$$
\frac{\Gamma \vdash_{\mathsf{CG}} t_1 : A \to B \quad \Gamma \vdash_{\mathsf{CG}} t_2 : A}{\Gamma \vdash_{\mathsf{CG}} t_1\, t_2 : B}\to_e \qquad \frac{}{\Gamma \vdash_{\mathsf{CG}} \mathsf{error}_A : A}\,\mathsf{error}
$$

Figure 2. Typing rules for Core Grady

$$
\frac{}{\mathsf{unbox}_A\, (\mathsf{box}_A\, t) \rightsquigarrow t}\,\mathsf{retract} \qquad \frac{A \neq B}{\mathsf{unbox}_A\, (\mathsf{box}_B\, t) \rightsquigarrow \mathsf{error}_A}\,\mathsf{raise}
$$

$$
\frac{x : B \vdash_{\mathsf{CG}} \mathcal{E}[x] : A}{\mathcal{E}[\mathsf{error}_B] \rightsquigarrow \mathsf{error}_A}\,\mathsf{error}
$$

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

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

$$
\frac{}{(\lambda(x : A_1).t_2)\, t_1 \rightsquigarrow [t_1/x]t_2}\,\beta \qquad \frac{}{\mathsf{fst}\, (t_1, t_2) \rightsquigarrow t_1}\times_{e_1}
$$

$$
\frac{}{\mathsf{snd}\, (t_1, t_2) \rightsquigarrow t_2}\times_{e_2} \qquad \frac{t_1 \rightsquigarrow t_2}{\mathcal{E}[t_1] \rightsquigarrow \mathcal{E}[t_2]}\,\mathsf{cong}
$$

Figure 3. Reduction rules for Core Grady

The previous result has a number of implications. It completely brings together the static and dynamic fragments of the gradual λ-model, and thus, fully relating the combination of dynamic and static typing to the past work of Lambek and Scott [13, 19]. It will allow for the definition of casting morphisms between arbitrary objects. Finally, from a practical perspective it will simplify our corresponding type systems derived from this model, because $\mathsf{Box}_S = \widehat{\mathsf{squash}}_S$ and $\mathsf{Unbox}_S = \widehat{\mathsf{split}}_S$ when S is a skeleton, and hence, we will only need a single retract in the corresponding type systems.

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 [13, 27]. Objects become types, and morphisms, $t : \Gamma \longrightarrow\!\!\!\!\!\rightarrow A$, 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 5.

The syntax for Core Grady is defined in Figure 1. The syntax is a straightforward extension of the simply typed λ-calculus. Arbitrary programs or terms are denoted by t and values by v. The latter are used to influence the evaluation strategy used by Core Grady. 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 $\mathsf{case}\, t\colon \mathsf{Nat\ of}\ 0 \to t_1, (\mathsf{succ}\, x) \to t_2$. 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. That result shows that these can actually be defined in terms of $\widehat{\mathsf{box}}_A$, $\widehat{\mathsf{unbox}}_A$, $\widehat{\mathsf{split}}_S$, and $\widehat{\mathsf{squash}}_S$ when A is any type and S is a skeleton, but we take the general versions as primitive, because they are the most useful from a programming perspective. In addition, as we mentioned above Box_A and Unbox_A divert to $\widehat{\mathsf{squash}}_A$ and $\widehat{\mathsf{split}}_A$ respectively when A is a skeleton. This implies that we no longer need two retracts, and hence, simplifies the language.

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. This implies that applying either box_A or unbox_A to some other term corresponds to function application as opposed to $\mathsf{succ}\, t$ which cannot be used without its argument. This fact is used in the definition of the evaluation strategy.

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 (or evaluates) in one step to the term t_2." The reduction relation is defined in Figure 3, and we denote the least reflexive and transitive closure of $\rightsquigarrow$ as $\rightsquigarrow^*$. Core Grady's reduction strategy is an extended version of call-by-name. It is specified using evaluation contexts that are denoted by $\mathcal{E}$ and are defined in Figure 1.

An evaluation context is essentially a term with a hole, denoted by $\Box$, in it. The hole can be filled (or plugged) with a term and is denoted by $\mathcal{E}[t]$. Note that plugging the hole of an

Syntax:

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

Metafunctions:

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

$$\begin{array}{l}\mathsf{fun}(?) = ? \to ? \\ \mathsf{fun}(A \to B) = A \to B\end{array}$$

Figure 4. Syntax and Metafunctions for Surface Grady

evaluation context results in a term. Evaluation contexts are used to give a compact definition of an evaluation strategy by first specifying the reduction axioms (Figure 3), then defining the evaluation contexts by placing a hole within the syntax of terms that specifies where evaluation is allowed to take place (Figure 1), finally, the following reduction rule is then added:

$$\frac{t_1 \rightsquigarrow t_2}{\mathcal{E}[t_1] \rightsquigarrow \mathcal{E}[t_2]}\mathrm{cong}$$

This rule states that evaluation can take place in the locations of the holes given in the definition of evaluation contexts (Figure 1).

How we define the syntax of values and evaluation contexts, and the evaluation rules determines the evaluation strategy. We consider as values λ-abstractions. Thus, the expression $\lambda(x : A).\Box$ is not an evaluation context, and hence, there is no evaluation under λ-abstractions. Similarly, we have no evaluation contexts which allow evaluation under the branches of a case-expression. In addition, the evaluation context $\mathsf{unbox}_A\ \Box$ allows for reduction in the argument position of an application of unbox_A, but we do not allow reduction in the argument position of an application of box_A. These restrictions are used to prevent infinite reduction from occurring in those positions. We want evaluation to make as much overall progress as possible.

Perhaps the most interesting reduction rules from Figure 3 are the first three: retract, raise, and error. The first two handle dynamic type casts and the third preserves dynamic type errors that have been raised in an evaluation position. The error reduction rule depends on typing which is necessary to insure that the type annotation is correct. This insures that type preservation will hold. Practically speaking, this dependency on typing is not significant, because we only evaluate closed well-typed programs anyway.

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 [23, 24].

4 Surface Grady

In this section we introduce the gradually typed surface language Surface Grady. Surface Grady is a small extension of

Typing Rules:

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

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

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

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

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

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

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

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

Type Consistency:

$$\frac{}{A \sim A}\ \mathrm{refl} \qquad \frac{}{A \sim ?}\ \mathrm{box} \qquad \frac{}{? \sim A}\ \mathrm{unbox}$$

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

Figure 5. Typing rules for Surface Grady

the surface language given by Siek et al. [24]. We have added natural numbers with their eliminator as well as cartesian products. The Surface Grady syntax is defined in Figure 4, and it corresponds to Core Grady's syntax (Figure 1), but without the explicit casts. The syntax for types and typing contexts do not change, and so we do not repeat them here.

The metafunctions $\mathsf{nat}(A)$, $\mathsf{prod}(A)$, and $\mathsf{fun}(A)$ are partial functions that will be used to determine when to use box in the elimination type checking rules for natural numbers, cartesian products, and function applications respectively. For example, if $\mathsf{nat}(A) = \mathsf{Nat}$, then the type A must have been either ? or Nat, and if it were the former then we know we can cast A to Nat via $\mathsf{box}_{\mathsf{Nat}}$. If $\mathsf{prod}(A) = B \times C$, then either $A = ?$ and $B \times C = ? \times ?$ or $A = B \times C$ for some other types B and C. This implies that if the former is true, then we can cast A to $B \times C$ via $\mathsf{box}_{(?\times ?)}$. The case is similar for $\mathsf{fun}(A)$.

The type checking and type consistency rules are given in Figure 5. Similarly to Core Grady the typing judgment is denoted by $\Gamma \vdash_{\mathrm{SG}} t : A$. Type checking depends on the notion of type consistency; first proposed by Siek and Taha [23]. This is a reflexive and symmetric, but non-transitive, relation on types denoted by $A \sim B$ which can be read as "the type A is consistent with the type B." The lack of transitivity is important, because if type consistency were transitive, then all types would be consistent, but this is too general. Consider an example, type consistency is responsible for the function application $(\lambda(x : ?).(\mathsf{succ}\ x))\ 3$ being typable in the surface language, because type Nat is consistent with the type ?. This implies that the elimination rule for function types must be extended with type consistency.

Type consistency states when two types are safely castable between each other when inserting explicit casts, and so, from a semantical perspective if $A \sim B$ holds, then there are casting morphisms (Definition 2.7) $c_1 : A \longrightarrow\!\!\!\!\!\rightarrow B$ and $c_2 : B \longrightarrow\!\!\!\!\!\rightarrow A$; see Lemma 5.2 in Section 5.

The typing rules for Surface Grady are a conservative extension of the typing rules for Core Grady (Figure 2). The extension is the removal of explicit casts and the addition of type consistency and the metafunctions from Figure 4. Each rule is modified in positions where casting is likely to occur which would be in all of the elimination rules as well as the typing rule for successor, because it is a type of application. Consider the elimination rule for function applications:

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

This rule has been extended with type consistency. The type of t_1 is allowed to be either ? or a function type $A_1 \to B_1$, by the definition of $\mathsf{fun}(C)$, if the former is true, then $A_1 \to B_1 = ? \to ?$ and A_2 can be any type at all, but if $C = A_1 \to B_1$, then A_2 must be consistent with A_1. Notice that if $C = A_1 \to B_1$ and $A_2 = A_1$, then this rule is equivalent to the usual rule for function application. We can now see that our example program $(\lambda(x : ?).(\mathsf{succ}\ x))\ 3$ is typable in Surface Grady. Similar reasoning can be used to understand the other typing rules as well.

Surface Grady is translated into Core Grady using the cast insertion algorithm detailed in Figure 6. The cast insertion judgment is denoted by $\Gamma \vdash t_1 \Rightarrow t_2 : A$ which is read as "the Surface Grady program t_1 of type A is translated into the Core Grady program t_2 of type A in context Γ." This algorithm is type directed, and is dependent on the partial metafunction $\mathsf{caster}(A, B)$ that constructs the casting morphism – in Core Grady – of type $A \to B$:

$$\begin{aligned}
&\mathsf{caster}(A, A) = \lambda(x : A).x\\
&\mathsf{caster}(A, ?) = \mathsf{box}_A\\
&\mathsf{caster}(?, B) = \mathsf{unbox}_B\\
&\mathsf{caster}((A_1 \times B_1), (A_2 \times B_2)) = \mathsf{caster}(A_1, A_2) \times \mathsf{caster}(B_1, B_2)\\
&\mathsf{caster}((A_1 \to B_1), (A_2 \to B_2)) = \mathsf{caster}(A_2, A_1) \to \mathsf{caster}(B_1, B_2)
\end{aligned}$$

The previous definition uses the following derivable functor rules:

$$\frac{\Gamma \vdash_{\mathsf{CG}} t_1 : A_1 \to A_2 \qquad \Gamma \vdash_{\mathsf{CG}} t_2 : B_1 \to B_2}{\Gamma \vdash_{\mathsf{CG}} t_1 \times t_2 : (A_1 \times B_1) \to (A_2 \times B_2)}$$

$$\frac{\Gamma \vdash_{\mathsf{CG}} t_1 : A_2 \to A_1 \qquad \Gamma \vdash_{\mathsf{CG}} t_2 : B_1 \to B_2}{\Gamma \vdash_{\mathsf{CG}} t_1 \to t_2 : (A_1 \to B_1) \to (A_2 \to B_2)}$$

They are defined as follows:

$$\begin{aligned}
t_1 \times t_2 &= \lambda(x : A_1 \times B_1).(t_1\ (\mathsf{fst}\ x),\ t_2\ (\mathsf{snd}\ x))\\
t_1 \to t_2 &= \lambda(x : A_1 \to B_1).\lambda(y : A_2).t_2\ (x\ (t_1\ y))
\end{aligned}$$

The definition of $\mathsf{caster}(A, B)$ is based on the definition of type consistency.

Lemma 4.1 (Type Consistency and Caster)**.** *If $A \sim B$, then $\Gamma \vdash_{\mathsf{CG}} \mathsf{caster}(A, B) : A \to B$.*

Proof. This proof holds by induction on $A \sim B$, but is vary routine, and so we omit its proof. □

$$\frac{x : A \in \Gamma}{\Gamma \vdash x \Rightarrow x : A} \qquad \frac{}{\Gamma \vdash 0 \Rightarrow 0 : \mathsf{Nat}} \qquad \frac{}{\Gamma \vdash \mathsf{triv} \Rightarrow \mathsf{triv} : \mathsf{Unit}}$$

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

$$\frac{\begin{array}{lll} & A_1 \sim A & \\ \Gamma \vdash t \Rightarrow t' : ? & A_2 \sim A & t_1'' = (c_1\ t_1') \\ \Gamma \vdash t_1 \Rightarrow t_1' : A_1 & \mathsf{caster}(A_1, A) = c_1 & t_2'' = (c_2\ t_2') \\ \Gamma, x : \mathsf{Nat} \vdash t_2 \Rightarrow t_2' : A_2 & \mathsf{caster}(A_2, A) = c_2 & t'' = (\mathsf{unbox}_{\mathsf{Nat}}\ t') \end{array}}{\Gamma \vdash (\mathsf{case}\ t\ \mathsf{of}\ 0 \to t_1, (\mathsf{succ}\ x) \to t_2) \Rightarrow (\mathsf{case}\ t''\ \mathsf{of}\ 0 \to t_1'', (\mathsf{succ}\ x) \to t_2'') : A}$$

$$\frac{\begin{array}{lll} & A_1 \sim A & \\ \Gamma \vdash t \Rightarrow t' : \mathsf{Nat} & A_2 \sim A & \\ \Gamma \vdash t_1 \Rightarrow t_1' : A_1 & \mathsf{caster}(A_1, A) = c_1 & t_1'' = (c_1\ t_1') \\ \Gamma, x : \mathsf{Nat} \vdash t_2 \Rightarrow t_2' : A_2 & \mathsf{caster}(A_2, A) = c_2 & t_2'' = (c_2\ t_2') \end{array}}{\Gamma \vdash (\mathsf{case}\ t\ \mathsf{of}\ 0 \to t_1, (\mathsf{succ}\ x) \to t_2) \Rightarrow (\mathsf{case}\ t'\ \mathsf{of}\ 0 \to t_1'', (\mathsf{succ}\ x) \to t_2'') : A}$$

$$\frac{\Gamma \vdash t_1 \Rightarrow t_3 : A_1 \quad \Gamma \vdash t_2 \Rightarrow t_4 : A_2}{\Gamma \vdash (t_1, t_2) \Rightarrow (t_3, t_4) : A_1 \times A_2}$$

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

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

$$\frac{\Gamma, x : A_1 \vdash t_1 \Rightarrow t_2 : A_2}{\Gamma \vdash \lambda(x : A_1).t_1 \Rightarrow \lambda(x : A_1).t_2 : A_1 \to A_2}$$

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

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

Figure 6. Cast Insertion Algorithm

Notice that for each typing rule that uses one of the metafunctions from Figure 4 there are two cast insertion rules corresponding to the typing rule.

The cast insertion algorithm is designed around where explict casts need to be inserted. This is accomplished by case splitting on the input to each metafunction from Figure 4 resulting in two rules per elimination rule. The first is the case where the input to the metafunction is ?, and the second is the case where the input to the metafunction is a type of the appropriate structure. For example, in the case of the elimination rule for function application, $\to_e$, from Figure 5, there are two cast insertion rules, the last two in Figure 6, where the first is when the type of the function, t_1, is ?, and the second when the type of t_1 is an arrow type. The former requires the type ? to be split into $? \to ?$ using $\mathsf{unbox}_{(?\to?)}$, and a casting morphism to cast the argument to the appropriate input type. The second cast insertion rule only needs to cast the argument type, because t_1 already has a function type.

The cast insertion algorithm preserves the type of the program.

Lemma 4.2 (Cast Insertion Preserves the Type)**.** *If $\Gamma \vdash_{\mathsf{SG}} t_1 : A$ and $\Gamma \vdash t_1 \Rightarrow t_2 : A$, then $\Gamma \vdash_{\mathsf{CG}} t_2 : A$.*

Proof. This proof holds by induction on $\Gamma \vdash_{\mathsf{SG}} t_1 : A$ which will determine which of the cast insertion rules need to be considered. At that point, a case split over the input to any

metafunctions from Figure 4 used in the typing rule may be necessary. We omit the proof in the interest of brevity. □

Finally, cast insertion also plays a role when interpreting Surface Grady into the categorical model. The next section gives the details.

5 Interpreting Surface and Core Grady in the Model

Interpreting a programming language into a categorical model requires three steps. First, the types are interpreted as objects. Then programs are interpreted as morphisms in the category, but this is a simplification. Every morphism, f, in a category has a source object and a target object, we usually denote this by $f : A \longrightarrow B$. Thus, in order to interpret programs as morphisms the program must have a source and target. So instead of interpreting raw terms as morphisms we interpret terms in their typing context. That is, we must show how to interpret every $\Gamma \vdash_{\mathsf{SG}} t : A$ as a morphism $t : [\![\Gamma]\!] \longrightarrow [\![A]\!]$. The third step is to show that whenever one program reduces to another their interpretations are isomorphic in the model. This means that whenever $\Gamma \vdash_{\mathsf{CG}} t_1 : A$, $\Gamma \vdash_{\mathsf{CG}} t_2 : A$, and $t_1 \rightsquigarrow t_2$, then $[\![t_1]\!] = [\![t_2]\!] : [\![\Gamma]\!] \longrightarrow [\![A]\!]$. The goal of this section is to prove these two facts for Surface Grady and Core Grady. This section heavily depends on Section 2, Section 3, and Section 4.

First, we must give the interpretation of types and contexts, but this interpretation is obvious, because we have been making sure to match the names of types and objects throughout this paper.

Definition 5.1. Suppose $(\mathcal{T}, C, ?, \mathsf{T}, \mathsf{split}, \mathsf{squash}, \mathsf{box}, \mathsf{unbox}, \mathsf{error})$ is a gradual λ-model. Then we define the interpretation of types into C as follows:

$$[\![?]\!] = ? \qquad [\![A_1 \to A_2]\!] = [\![A_1]\!] \to [\![A_2]\!]$$
$$[\![\mathsf{Unit}]\!] = \mathsf{Unit} \qquad [\![A_1 \times A_2]\!] = [\![A_1]\!] \times [\![A_2]\!]$$
$$[\![\mathsf{Nat}]\!] = \mathsf{Nat}$$

We extend this interpretation to typing contexts as follows:

$$[\![\cdot]\!] = \mathsf{Unit} \qquad [\![\Gamma, x : A]\!] = [\![\Gamma]\!] \times [\![A]\!]$$

Throughout the remainder of this paper we will drop the interpretation symbols around types.

Before we can interpret the typing rules of Surface and Core Grady we must show how to interpret the type consistency relation from Figure 4. These will correspond to casting morphisms (Definition 2.7).

Lemma 5.2 (Type Consistency in the Model)**.** *Suppose* $(\mathcal{T}, C, ?, \mathsf{T}, \mathsf{split}, \mathsf{squash}, \mathsf{box}, \mathsf{unbox}, \mathsf{error})$ *is a gradual* λ*-model, and* $A \sim B$ *for some types* A *and* B*. Then there are two casting morphisms* $c_1 : A \longrightarrow B$ *and* $c_2 : B \longrightarrow A$*.*

Proof. This proof holds by induction on the form $A \sim B$ using the morphisms $\mathsf{Box}_A : A \longrightarrow ?$ and $\mathsf{Unbox}_A : ? \longrightarrow A$ from Lemma 2.14. Please see Appendix B.3 for the complete proof. □

Showing that both c_1 and c_2 exist corresponds to the fact that $A \sim B$ is symmetric.

$[\![\Gamma_1, x : A_i, \Gamma_2 \vdash_{\mathsf{SG}} x : A_i]\!] = \pi_i$
$[\![\Gamma \vdash_{\mathsf{SG}} \mathsf{triv} : \mathsf{Unit}]\!] = \mathsf{triv}_{[\![\Gamma]\!]}$
$[\![\Gamma \vdash_{\mathsf{SG}} 0 : \mathsf{Nat}]\!] = \mathsf{triv}_{[\![\Gamma]\!]}; z$
$[\![\Gamma \vdash_{\mathsf{SG}} \mathsf{succ}\ t : \mathsf{Nat}]\!] = [\![t]\!]; c; \mathsf{succ}$ where $\Gamma \vdash_{\mathsf{SG}} t : A$ and $c : A \longrightarrow \mathsf{Nat}$
$[\![\Gamma \vdash_{\mathsf{SG}} \mathsf{case}\ t\ \mathsf{of}\ 0 \to t_1, (\mathsf{succ}\ x) \to t_2 : A]\!]$ $= \langle \mathsf{id}_{[\![\Gamma]\!]}, [\![t]\!]; c_1 \rangle; \mathsf{case}_{[\![\Gamma]\!], [\![A]\!]} \langle [\![t_1]\!]; c_2, [\![t_2]\!]; c_3 \rangle$ where $\Gamma \vdash_{\mathsf{SG}} t : C$, $\Gamma \vdash_{\mathsf{SG}} t_1 : A_1$, $\Gamma, x : \mathsf{Nat} \vdash_{\mathsf{SG}} t_2 : A_2$, $c_1 : C \longrightarrow \mathsf{Nat}$, $c_2 : A_1 \longrightarrow A$, and $c_3 : A_2 \longrightarrow A$
$[\![\Gamma \vdash_{\mathsf{SG}} (t_1, t_2) : A_1 \times A_2]\!] = \langle [\![t_1]\!], [\![t_2]\!] \rangle$ where $\Gamma \vdash_{\mathsf{SG}} t_1 : A_1$ and $\Gamma \vdash_{\mathsf{SG}} t_2 : A_2$
$[\![\Gamma \vdash_{\mathsf{SG}} \mathsf{fst}\ t : A_1]\!] = [\![t]\!]; c; \pi_1$ where $\Gamma \vdash_{\mathsf{SG}} t : B$ and $c : B \longrightarrow (A_1 \times A_2)$
$[\![\Gamma \vdash_{\mathsf{SG}} \mathsf{snd}\ t : A_2]\!] = [\![t]\!]; c; \pi_2$ where $\Gamma \vdash_{\mathsf{SG}} t : B$ and $c : B \longrightarrow (A_1 \times A_2)$
$[\![\Gamma \vdash_{\mathsf{SG}} \lambda(x : A).t : A \to B]\!] = \mathsf{curry}([\![t]\!])$ where $\Gamma, x : A \vdash_{\mathsf{SG}} t : B$
$[\![\Gamma \vdash_{\mathsf{SG}} t_1\ t_2 : B_1]\!] = \langle [\![t_1]\!]; c_1, [\![t_2]\!]; c_2 \rangle; \mathsf{app}_{A_1, B_1}$ where $\Gamma \vdash_{\mathsf{SG}} t_1 : C$, $\Gamma \vdash_{\mathsf{SG}} t_2 : A_2$, $c_1 : C \longrightarrow (A_1 \to B_1)$, and $c_2 : A_2 \longrightarrow A_1$
$[\![\Gamma \vdash_{\mathsf{CG}} \mathsf{box}_A : A \to ?]\!] = \mathsf{triv}_{[\![\Gamma]\!]}; \mathsf{curry}(\mathsf{Box}_A)$
$[\![\Gamma \vdash_{\mathsf{CG}} \mathsf{unbox}_A : ? \to A]\!] = \mathsf{triv}_{[\![\Gamma]\!]}; \mathsf{curry}(\mathsf{Unbox}_A)$

Figure 7. Interpretation of Terms

At this point we have everything we need to show our main result which is that typing in both Surface and Core Grady, and evaluation in Core Grady can be interpreted into the categorical model. The interpretation of terms used in the following proofs is summarized in Figure 7. We only summarize the interpretation of the Surface Grady programs and just box and unbox in Core Grady, because the interpretation of Core Grady is equivalent to the interpretation of Surface Grady where all casting morphisms have been replaced with the identity morphism.

Theorem 5.3 (Interpretation of Typing)**.** *Suppose* $(\mathcal{T}, C, ?, \mathsf{T}, \mathsf{split}, \mathsf{squash}, \mathsf{box}, \mathsf{unbox}, \mathsf{error})$ *is a gradual* λ*-model. If* $\Gamma \vdash_{\mathsf{SG}} t : A$ *or* $\Gamma \vdash_{\mathsf{CG}} t : A$*, then there is a morphism* $[\![t]\!] : [\![\Gamma]\!] \longrightarrow A$ *in* C*.*

Proof. Both parts of the proof hold by induction on the form of the assumed typing derivation, and uses most of the results we have developed up to this point. Please see Appendix B.4 for the complete proof. □

Theorem 5.4 (Interpretation of Evaluation)**.** *Suppose* $(\mathcal{T}, C, ?, \mathsf{T}, \mathsf{split}, \mathsf{squash}, \mathsf{box}, \mathsf{unbox}, \mathsf{error})$ *is a gradual* λ*-model. If* $\Gamma \vdash_{\mathsf{CG}} t_1 : A$, $\Gamma \vdash_{\mathsf{CG}} t_2 : A$*, and* $t_1 \rightsquigarrow t_2$*, then* $[\![t_1]\!] = [\![t_2]\!] : [\![\Gamma]\!] \longrightarrow A$*.*

Proof. This proof holds by induction on the form of $t_1 \rightsquigarrow t_2$, and uses Theorem 5.3, Lemma 5.2, Corollary B.1, and Lemma B.2. Please see Appendix B.5 for the complete proof. □

One can see a direct connection between the proof of interpretation of typing (Theorem 5.3) and the cast insertion algorithm (Figure 6). During the proof – summarized in Figure 7 – we construct casting morphisms from type consistency which is essentially the semantic equivalent to $\mathsf{caster}(A, B)$.

Syntax:

$$\begin{array}{rrl} \text{(Atomic Types)} & T & ::= \mathsf{Unit} \mid \mathsf{Nat} \\ \text{(Ground Types)} & R & ::= T \mid ? \to ? \\ \text{(values)} & v & ::= \lambda x : A.t \\ \text{(terms)} & t & ::= \cdots \mid t : A \Rightarrow B \end{array}$$

Typing Rules:

$$\cdots \qquad \frac{\Gamma \vdash t : A \quad A \sim B}{\Gamma \vdash (t : A \Rightarrow B) : B}\ \text{cast}$$

Reduction Relation:

$$\cdots \qquad \frac{}{v : T \Rightarrow T \rightsquigarrow v}\ \text{id-atom} \qquad \frac{}{v : ? \Rightarrow ? \rightsquigarrow v}\ \text{id-U}$$

$$\frac{}{v : R \Rightarrow ? \Rightarrow R \rightsquigarrow v}\ \text{succeed} \qquad \frac{R_1 \neq R_2}{v : R_1 \Rightarrow ? \Rightarrow R_2 \rightsquigarrow \mathsf{error}_{R_2}}\ \text{fail}$$

$$\frac{}{(v_1 : (A_1 \to B_1) \Rightarrow (A_2 \to B_2))\, v_2 \rightsquigarrow v_1\, (v_2 : A_2 \Rightarrow A_1) : B_1 \Rightarrow B_2}\ \to\Rightarrow$$

$$\frac{A \sim R \quad A \neq R \quad A \neq ?}{v : A \Rightarrow ? \rightsquigarrow v : A \Rightarrow R \Rightarrow ?}\ \text{expand}_1$$

$$\frac{A \sim R \quad A \neq R \quad A \neq ?}{v : ? \Rightarrow A \rightsquigarrow v : ? \Rightarrow R \Rightarrow A}\ \text{expand}_2$$

Figure 8. The core casting calculus: $\lambda^{\rightrightarrows}_{\to}$

6 Modeling Siek and Taha's Gradual λ-Calculus

In this section we show that Siek and Taha's gradual λ-calculus [23, 24] can be modeled in a gradual λ-model. Thus showing that other gradual type systems can benefit from our semantics.

We only consider Siek and Taha's casting calculus, called $\lambda^{\rightrightarrows}_{\to}$, because their surface language is essentially Surface Grady. The complete language specification is summarized in Figure 8. The casting calculus $\lambda^{\rightrightarrows}_{\to}$ is Core Grady where box A and unbox A have been replaced with the explicit cast $t : A \Rightarrow B$. In addition, the typing rules for box A and unbox A have been replaced with the cast typing rule. The syntax of types are the same as for Core Grady; see Figure 1. We do not consider cartesian products in $\lambda^{\rightrightarrows}_{\to}$, but they can be added to $\lambda^{\rightrightarrows}_{\to}$ in the same way that they are defined for Core Grady. One interesting aspect of $\lambda^{\rightrightarrows}_{\to}$ is that it depends on type consistency, used in the cast rule, but it is defined in the same way as in Figure 5, however without cartesian products.

The explicit cast, $t : A \Rightarrow B$, should be understand as casting the term t whose type is A to the type B. Thus, boxing a term, t, of type A is defined by $t : A \Rightarrow ?$, and unboxing is defined by $t : ? \Rightarrow A$. Semantically, type consistency corresponds to casting morphisms, and because their morphisms they compose, but type consistency is not transitive. However, using the explicit cast we can compose type consistency. Suppose $A \sim B$, $B \sim C$, and $\Gamma \vdash t : A$, then using the rule, cast, of the casting calculus we may type $\Gamma \vdash t : A \Rightarrow B \Rightarrow C : C$. This composition is the reason why we interpret type consistency as casting morphisms in the model.

The reduction rules for $\lambda^{\rightrightarrows}_{\to}$ includes all of the rules from Core Grady except for the retract rules and the rules for cartesian products in addition to the new casting rules given in Figure 8. It is easy to see that the new casting rules, succeed and fail, correspond to the Core Grady reduction rules, retract and raise, from Figure 3. The other new casting rules are congruence rules to prevent stuck terms and push casting towards the succeed and fail rules.

We now prove the two main properties for modeling type systems in a categorical model just as we did in the previous section. The interpretation of types and contexts remain the same as in Definition 5.1.

Theorem 6.1 (Interpretation of Typing for $\lambda^{\rightrightarrows}_{\to}$)**.** *Suppose* ($\mathcal{T}$, $\mathcal{C}$, ?, T, split, squash, box, unbox, error) *is a gradual λ-model. If $\Gamma \vdash t : A$, then there is a morphism $[\![t]\!] : [\![\Gamma]\!] \longrightarrow A$ in $\mathcal{C}$.*

Proof. This is a proof by induction on $\Gamma \vdash t : A$. We only show the case for the explicit cast, because all others are equivalent to the interpretation of Core Grady.

Case: $\dfrac{\Gamma \vdash t : A \quad A \sim B}{\Gamma \vdash (t : A \Rightarrow B) : B}\ \text{cast}$

By the induction hypothesis there is a morphism $[\![t]\!] : [\![\Gamma]\!] \longrightarrow A$, and by type consistency in the model (Lemma 5.2) there is a casting morphism $c_1 : A \longrightarrow B$. So take $[\![t : A \Rightarrow B]\!] = [\![t]\!]; c_1 : [\![\Gamma]\!] \longrightarrow B$.

□

Theorem 6.2 (Interpretation of Evaluation for $\lambda^{\rightrightarrows}_{\to}$)**.** *Suppose* ($\mathcal{T}$, $\mathcal{C}$, ?, T, split, squash, box, unbox, error) *is a gradual λ-model. If $\Gamma \vdash t_1 : A$, $\Gamma \vdash t_2 : A$, and $t_1 \rightsquigarrow t_2$, then $[\![t_1]\!] = [\![t_2]\!] : [\![\Gamma]\!] \longrightarrow A$.*

Proof. This proof holds by induction on the form of $t_1 \rightsquigarrow t_2$, and uses Theorem 6.1, Lemma 5.2, Corollary B.1, and inversion for typing whose definition we omit in the interest of space.

We show only one of the most interesting cases here, but please see Appendix B.6 for the complete proof.

Case: $\dfrac{}{v : R \Rightarrow ? \Rightarrow R \rightsquigarrow v}\ \text{succeed}$

By inversion for typing the typing derivation for $v : R \Rightarrow ? \Rightarrow R$ is as follows:

$$\dfrac{\dfrac{\Gamma \vdash v : R \quad R \sim ?}{\Gamma \vdash v : R \Rightarrow ? : ?} \qquad ? \sim R}{\Gamma \vdash v : R \Rightarrow ? \Rightarrow R : R}$$

By the induction hypothesis we have the morphism $[\![v]\!] : [\![\Gamma]\!] \longrightarrow R$. As we can see we will first use Box_R and then Unbox_R based on Corollary B.1. Thus, $[\![v : R \Rightarrow ? \Rightarrow R]\!] = [\![v]\!]; \mathsf{Box}_R; \mathsf{Unbox}_R = [\![v]\!]$, because Box_R and Unbox_R form a retract (Lemma 2.15).

□

7 The Gradual Guarantee

We now turn to proving that the gradual guarantee – see Theorem 7.1 – holds for Grady. This is the defining property of every gradual type system. In fact, Siek et al. [24] argue that the gradual guarantee is what separates type systems that simply combine dynamic and static typing from systems that not only combine dynamic and static typing, but also allow the

Type Precision:

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

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

Context Precision:

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

Figure 9. Type and Context Precision

programmer to program in dynamic style without the need to insert explicit casts. Intuitively, the gradual guarantee states that a gradually typed program should preserve its type and behavior when explicit casts are either inserted or removed.

Our proof follows the scheme adopted by Siek et al. [24] in their proof of the gradual guarantee for the gradual simply typed λ-calculus. That is, we prove the exact same results as they do. We will call a type **static** if it does not mention the unknown type. The first step in proving the gradual guarantee is making rigorous the characterization of when one type is more static than another type, and when one program is more dynamic than another program. This is done by defining the notion of type and term precision.

Type precision is denoted by $A \sqsubseteq B$ and is read "the type A is more precise than type B." It is defined in Figure 9 with its extension to typing contexts. Type precision is a preorder on types where as one travels up a chain the types tend toward the unknown type as opposed to when one travels down a chain the type tends toward some static type. This implies that if $A \sqsubseteq B$, then A is more static than B, but B is more dynamic than A. The direction of type precision, and term precision which will be defined next, may seem backward, but one can consider the unknown type as a universe of types [8], and so, it is natural to consider it as a top element in the preorder.

Term precision is similar to type precision. It is denoted by $t_1 \sqsubseteq t_2$, and is read "the program t_1 is more precise than the program t_2." That is, t_1 is more static while t_2 is more dynamic. Term precision is defined in Figure 10 for both Surface Grady and Core Grady programs. The definition of term precision for Core Grady includes similar rules defining term precision for Surface Grady in addition to the ones given in Figure 10, and so we do not repeat them. The term precision rules for Core Grady are also annotated with typing contexts to keep track of the types of free variables. This is needed because the rules depend on typing.

Perhaps the most interesting rules are the ones for box, unbox, and error. Since the job of $\text{unbox}_A\ t$ is to specialize the type of t at a more specific type, then $\Gamma \vdash (\text{unbox}_A\ t) \sqsubseteq t$. Dually, since the job of $\text{box}_A\ t$ is to generalize the type of t to ?, then $\Gamma \vdash t \sqsubseteq (\text{box}_A\ t)$.

At this point we can now rigorously state and prove the gradual guarantee for the Grady languages, where we denote a diverging term by $t \uparrow$.

Theorem 7.1 (Gradual Guarantee)**.**

i. If $\cdot \vdash_{\text{SG}} t : A$ and $t \sqsubseteq t'$, then $\cdot \vdash_{\text{SG}} t' : B$ and $A \sqsubseteq B$.

ii. Suppose $\cdot \vdash_{\text{CG}} t : A$ and $\cdot \vdash t \sqsubseteq t'$. Then

Term Precision for Surface Grady:

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

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

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

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

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

Term Precision for Core Grady:

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

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

Figure 10. Term Precision

a. if $t \rightsquigarrow^* v$, then $t' \rightsquigarrow^* v'$ and $\cdot \vdash v \sqsubseteq v'$,

b. if $t \uparrow$, then $t' \uparrow$,

c. if $t' \rightsquigarrow^* v'$, then $t \rightsquigarrow^* v$ where $\cdot \vdash v \sqsubseteq v'$, or $t \rightsquigarrow^* \text{error}_A$, and

d. if $t' \uparrow$, then $t \uparrow$ or $t \rightsquigarrow^* \text{error}_A$.

Proof. This result follows from the same proof as [24], and so, we only give a brief summary. Part i. holds by Lemma 7.2, and Part ii. follows from simulation of more precise programs (Lemma 7.3). □

Part one states that one may insert casts into a closed gradual program, t, yielding a less precise program, t', and the program will remain typable, but at a less precise type. This part follows from the following generalization:

Lemma 7.2 (Gradual Guarantee Part One)**.** *If $\Gamma \vdash_{\text{SG}} t : A$, $t \sqsubseteq t'$, and $\Gamma \sqsubseteq \Gamma'$ then $\Gamma' \vdash_{\text{SG}} t' : B$ and $A \sqsubseteq B$.*

Proof. This is a proof by induction on $\Gamma \vdash_{\text{SG}} t : A$; see Appendix B.8 for the complete proof. □

The remaining parts of the gradual guarantee follow from the next result.

Lemma 7.3 (Simulation of More Precise Programs)**.** *Suppose $\Gamma \vdash_{\text{CG}} t_1 : A$, $\Gamma \vdash t_1 \sqsubseteq t_1'$, $\Gamma \vdash_{\text{CG}} t_1' : A'$, and $t_1 \rightsquigarrow t_2$. Then $t_1' \rightsquigarrow^* t_2'$ and $\Gamma \vdash t_2 \sqsubseteq t_2'$ for some t_2'.*

Proof. This proof holds by induction on $\Gamma \vdash_{\text{CG}} t_1 : A_1$. See Appendix B.9 for the complete proof. □

This result simply states that programs may become less precise by adding or removing casts, but they will behave in an expected manner.

The proofs of the previous results require a number of auxiliary lemmas that are too numerous to include in this section, but they can all be found in Appendix A, but we omit most of their proofs in the interest of space. Each of the proofs of the previous results take great care in pointing out where these auxiliary results are used.

8 Explicit Casts in Gradual Type Systems

In this section we introduce something that is seemingly small, but very useful when programming in gradual type systems. In addition, the authors are not aware of this being pointed out in the literature.

First, the untyped Y combinator can be defined in Surface Grady as follows:

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

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

The previous definition is a lot cleaner without the explicit casts, and in the style of programming in the untyped λ-calculus.

Now suppose we added polymorphism – in Haskell style – to the Grady languages, then we might want to define a typed version of fix like the following:

```
fixT : (X → X) → X
fixT = \(f:X → X) → fix (\(y:?) → (f y)));
```

However, fixT does not type check. Notice that we must produce something of type X, but fix (\(y:?)–> (f y)) has type ? and it will not be implicitly cast.

We can better understand the issue by examining the function application typing rule:

$$\frac{\Gamma \vdash_{\mathsf{SG}} t_1 : C \qquad A_2 \sim A_1 \qquad \Gamma \vdash_{\mathsf{SG}} t_2 : A_2 \qquad \mathsf{fun}(C) = A_1 \to B_1}{\Gamma \vdash_{\mathsf{SG}} t_1\ t_2 : B_1}\ \to_e$$

As we can see the only implicit casting that occurs is in the case of the function, t_1, and the argument, t_2, but not the actual result of the application. Thus, in order to fix fixT we must insert an explicit cast, but we have removed the explicit casts from Surface Grady.

All is not lost, because as it turns out, explicit casts can be defined in Surface Grady:

$$\begin{array}{lcl} \mathsf{box}_A\ t & = & (\lambda(x : A).x)\ t \\ \mathsf{unbox}_A\ t & = & (\lambda(x : ?).x)\ t \end{array}$$

Their typing rules are also derivable (proof omitted).

Lemma 8.1 (Box and Unbox in Surface Grady)**.** *The following typing rules are derivable:*

$$\frac{\Gamma \vdash_{\mathsf{SG}} t : A}{\Gamma \vdash_{\mathsf{SG}} \mathsf{box}_A\ t : ?}\ \mathsf{box} \qquad \frac{\Gamma \vdash_{\mathsf{SG}} t : A}{\Gamma \vdash_{\mathsf{SG}} \mathsf{unbox}_A\ t : A}\ \mathsf{unbox}$$

Lastly, if we run the cast insertion algorithm (Figure 6) on $\mathsf{box}_A\ t$, then it will produce the Core Grady program $(\lambda(x : A).x)\ (\mathsf{box}_A\ t)$ and if we run the algorithm on $\mathsf{unbox}_A\ t$, then it will produce $(\lambda(x : ?).x)\ (\mathsf{unbox}_A\ t)$. This shows that the Surface Grady explicit casts correspond to the Core Grady explicit casts. We can now use Surface Grady's explicit casts to properly define fixT:

```
fixT : (X → X) → X
fixT = \(f:X → X) → unbox X (fix (\(y:?) → (f y))));
```

The problem we outline here is not specific to Surface Grady, but to all gradual type systems. If we program only using implicit casts, then we are not taking advantage of the full power of our type system, but if we combine them with explicit casts, then more programs become typable.

9 Related Work

Abadi et al. [1] combine dynamic and static typing by adding a new type called Dynamic along with a new case construct for pattern matching on types. We do not add such a case construct, and as a result, show that we can obtain a surprising amount of expressivity without it. They also provide denotational models.

Henglein [10] defines the dynamic λ-calculus by adding a new type Dyn to the simply typed λ-calculus and then adding primitive casting operations called tagging and check-and-untag. These new operations tag type constructors with their types. Then untagging checks to make sure the target tag matches the source tag, and if not, returns a dynamic type error. These operations can be used to build casting coercions which are very similar to our casting morphisms. We can also define split, squash, box, and unbox in terms of Henglein's casting coercions. We consider this paper as a clarification of Henglein's system. His core casting calculus can be interpreted into our setting where we require retracts instead of full isomorphisms.

10 Future Work

The categorical model and corresponding type system presented here sets the stage for a number of interesting lines of research.

Linear typing has a number of applications in functional programming [25]. For example, allowing values of linear type to change the world. Thus, it is worthwhile combining gradual typing with linear typing. Benton [2] showed that the statically typed λ-calculus can be mixed with the statically typed linear λ-calculus by taking a symmetric monoidal adjunction between a cartesian closed category and a symmetric monoidal closed category called a mixed linear/non-linear model or LNL model. Gradual λ-models defined here make up half of a LNL model mixing gradual typing with linear typing. We are actively working on this new combination.

Gradual λ-models correspond to the core language of a gradual type system. In the future we will investigate the categorical model for the surface language and a functorial relationship between this new model and gradual λ-models. The most interesting aspect of this model will be how to handle implicit casting. Then we will be able to state and prove a model theoretic version of the gradual guarantee. Another interesting aspect will be in how to deal with type and term precision in the model.

11 Acknowledgments

The authors thank Jeremy Siek for his feedback on previous drafts of this paper. The first author thanks Ronald Garcia for his wonderful invited talk at Trends in Functional Programming 2016 which introduced the first author to gradual typing and its open problems. This paper was typeset with the help of the amazing Ott tool [21].

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] P. N. Benton. 1995. A Mixed Linear and Non-Linear Logic: Proofs, Terms and Models (Extended Abstract). In *Selected Papers from the 8th International Workshop on Computer Science Logic (CSL '94)*. Springer-Verlag, London, UK, UK, 121–135.

[3] Y. Bres, B. P. Serpette, and M. Serrano. 2004. Compiling scheme programs to .NET common intermediate language. *2nd International Workshop on .NET Technologies, Pilzen, Czech Republic* (May 2004).

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

[5] C. Chambers and the Cecil Group. 2004. *The Cecil language: Specification and rationale.* Technical report. Department of Computer Science and Engineering, University of Washington, Seattle, Washington,.

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

[7] R. B. de Oliveira. 2005. The Boo programming language. (2005). http://boo.codehaus.org.

[8] 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.

[9] Kathryn E. Gray, Robert Bruce Findler, and Matthew Flatt. 2005. Fine-grained Interoperability Through Mirrors and Contracts. *SIGPLAN Not.* 40, 10 (Oct. 2005), 231–245. DOI: http://dx.doi.org/10.1145/1103845.1094830

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

[11] 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.

[12] Khurram A. Jafery and Joshua Dunfield. 2017. Sums of Uncertainty: Refinements Go Gradual. *SIGPLAN Not.* 52, 1 (Jan. 2017), 804–817. DOI: http://dx.doi.org/10.1145/3093333.3009865

[13] 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.

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

[15] Nico Lehmann and Éric Tanter. 2017. Gradual Refinement Types. In *Proceedings of the 44th ACM SIGPLAN Symposium on Principles of Programming Languages (POPL 2017)*. ACM, New York, NY, USA, 775–788. DOI: http://dx.doi.org/10.1145/3009837.3009856

[16] E. Meijer and P. Drayton. 2004. Static typing where possible, dynamic typing when needed: The end of the cold war between programming languages. *OOPSLA'04 Workshop on Revival of Dynamic Languages* (2004).

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

[18] John C. Reynolds. 1995. Using Functor Categories to Generate Intermediate Code. In *Proceedings of the 22Nd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL '95)*. ACM, New York, NY, USA, 25–36. DOI: http://dx.doi.org/10.1145/199448.199452

[19] 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.

[20] M. Serrano. 2002. *Bigloo: a practical Scheme compiler.* Inria-Rocquencourt.

[21] 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.

[22] Jeremy Siek and Walid Taha. 2007. Gradual Typing for Objects. In *Proceedings of the 21st European Conference on ECOOP 2007: Object-Oriented Programming (ECOOP '07)*. Springer-Verlag, Berlin, Heidelberg, 2–27. DOI: http://dx.doi.org/10.1007/978-3-540-73589-2_2

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

[24] 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.

[25] Philip Wadler. 1990. Linear types can change the world. In *IFIP TC*, Vol. 2. 347–359.

[26] Philip Wadler. 1995. *Monads for functional programming.* Springer Berlin Heidelberg, Berlin, Heidelberg, 24–52. DOI: http://dx.doi.org/10.1007/3-540-59451-5_2

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

A Auxiliary Metatheoretic Results

Lemma A.1 (Type Preservation). *If* $\Gamma \vdash_{\mathsf{CG}} t_1 : A$ *and* $t_1 \rightsquigarrow t_2$, *then* $\Gamma \vdash_{\mathsf{CG}} t_2 : A$.

Proof. This proof holds by induction on $\Gamma \vdash_{\mathsf{CG}} t_1 : A$ with further case analysis on the structure the derivation $t_1 \rightsquigarrow t_2$. □

Lemma A.2 (Inversion for Type Precision). *Suppose* $A \sqsubseteq B$. *Then:*

i. if $A = A_1 \to B_1$, *then* $B = ?$, *or* $B = A_2 \to B_2$, $A_1 \sqsubseteq A_2$, *and* $B_1 \sqsubseteq B_2$.
ii. if $A = A_1 \times B_1$, *then* $B = ?$, *or* $B = A_2 \times B_2$, $A_1 \sqsubseteq A_2$, *and* $B_1 \sqsubseteq B_2$.

Proof. This proof holds by induction on the form of $A \sqsubseteq B$. □

Lemma A.3 (Surface Grady Inversion for Term Precision). *Suppose* $t \sqsubseteq t'$. *Then:*

i. if $t = \mathsf{succ}\ t_1$, *then* $t' = \mathsf{succ}\ t_2$ *and* $t_1 \sqsubseteq t_2$.
ii. if $t = (\mathsf{case}\ t_1\ \mathsf{of}\ 0 \to t_2, (\mathsf{succ}\ x) \to t_3)$, *then* $t' = (\mathsf{case}\ t_1'\ \mathsf{of}\ 0 \to t_2', (\mathsf{succ}\ x) \to t_3')$, $t_1 \sqsubseteq t_1'$, $t_2 \sqsubseteq t_2'$, *and* $t_3 \sqsubseteq t_3'$.
iii. if $t = (t_1, t_2)$, *then* $t' = (t_1', t_2')$, $t_1 \sqsubseteq t_1'$, *and* $t_2 \sqsubseteq t_2'$.
iv. if $t = \mathsf{fst}\ t_1$, *then* $t' = \mathsf{fst}\ t_1'$ *and* $t_1 \sqsubseteq t_1'$.
v. if $t = \mathsf{snd}\ t_1$, *then* $t' = \mathsf{snd}\ t_1'$ *and* $t_1 \sqsubseteq t_1'$.
vi. if $t = \lambda(x : A_1).t_1$, *then* $t' = \lambda(x : A_1).t_1'$ *and* $t_1 \sqsubseteq t_1'$.
vii. if $t = (t_1\ t_2)$, *then* $t' = (t_1'\ t_2')$, $t_1 \sqsubseteq t_1'$, *and* $t_2 \sqsubseteq t_2'$.

Proof. This proof holds by induction on the form of $t \sqsubseteq t'$. □

Lemma A.4 (Inversion for Type Consistency). *Suppose* $A \sim B$. *Then:*

i. if $A = A_1 \to B_1$, *then* $B = ?$, *or* $B = A_2 \to B_2$, $A_2 \sim A_1$, *and* $B_1 \sim B_2$.
ii. if $A = A_1 \to B_1$, *then* $B = ?$, *or* $B = A_2 \to B_2$, $A_2 \sim A_1$, *and* $B_1 \sim B_2$.
iii. if $A = A_1 \times B_1$, *then* $B = ?$, *or* $B = A_2 \times B_2$, $A_1 \sim A_2$, *and* $B_1 \sim B_2$.

Proof. This proof holds by induction on the the form of $A \sim B$. □

Lemma A.5 (Symmetry for Type Consistency). *If* $A \sim B$, *then* $B \sim A$.

Proof. This holds by induction on the form of $A \sim B$. □

Lemma A.6 (Type Precision and Consistency). *If* $A \sqsubseteq B$, *then* $A \sim B$.

Proof. This proof holds by induction on $A \sqsubseteq B$. □

Lemma A.7. *If* $A \sqsubseteq B$ *and* $A \sqsubseteq C$, *then* $B \sim C$.

Proof. It must be the case that either $B \sqsubseteq C$ or $C \sqsubseteq B$, but in both cases we know $B \sim C$ by Lemma A.6. □

Lemma A.8 (Transitivity for Type Precision). *If* $A \sqsubseteq B$ *and* $B \sqsubseteq C$, *then* $A \sqsubseteq C$.

Proof. This proof holds by induction on $A \sqsubseteq B$ with a case analysis over $B \sqsubseteq C$. □

Lemma A.9. *Suppose $A \sqsubseteq B$. Then*

i. If $\mathsf{nat}(A) = \mathsf{Nat}$, then $\mathsf{nat}(B) = \mathsf{Nat}$.
ii. If $\mathsf{list}(A) = \mathsf{List}\ C$, then $\mathsf{list}(B) = \mathsf{List}\ C'$ and $C \sqsubseteq C'$.
iii. If $\mathsf{fun}(A) = A_1 \to A_2$, then $\mathsf{fun}(B) = A_1' \to A_2'$, $A_1 \sqsubseteq A_1'$, and $A_2 \sqsubseteq A_2'$.

Proof. This proof holds by induction on $A \sqsubseteq B$. □

Lemma A.10.

i. If $A_1 \sqsubseteq A_1'$ and $A_1 \sim A_2$ then $A_1' \sim A_2$.
ii. If $A_2 \sqsubseteq A_2'$ and $A_1 \sim A_2$ then $A_1 \sim A_2'$.

Proof. Both parts hold by induction on the assumed type consistency judgment. See Appendix B.7 for the complete proof. □

Corollary A.11. *If $A_1 \sqsubseteq A_1'$, $A_2 \sqsubseteq A_2'$, and $A_1 \sim A_2$ then $A_1' \sim A_2'$.*

Lemma A.12 (Typing for Type Precision)**.** *If $\Gamma \vdash_{\mathsf{SG}} t_1 : A$, $t_1 \sqsubseteq t_2$, and $\Gamma \sqsubseteq \Gamma'$, then $\Gamma' \vdash_{\mathsf{SG}} t_2 : B$ and $A \sqsubseteq B$.*

Proof. This proof holds by induction on $\Gamma \vdash_{\mathsf{SG}} t_1 : A$ with a case analysis over $t_1 \sqsubseteq t_2$. □

Lemma A.13 (Substitution for Term Precision)**.** *If $\Gamma, x : A \vdash t_1 \sqsubseteq t_2$ and $\Gamma \vdash t_1' \sqsubseteq t_2'$, then $\Gamma \vdash [t_1'/x]t_1 \sqsubseteq [t_2'/x]t_2$.*

Proof. This proof holds by induction on $\Gamma, x : A \vdash t_1 \sqsubseteq t_2$. □

Lemma A.14 (Typeability Inversion)**.**

i. If $\Gamma \vdash_{\mathsf{CG}} \mathsf{succ}\ t : A$, then $\Gamma \vdash_{\mathsf{CG}} t : A'$ for some A'.
ii. If $\Gamma \vdash_{\mathsf{CG}} \mathsf{case}\ t : \mathsf{Nat}\ \mathsf{of}\ 0 \to t_1, (\mathsf{succ}\ x) \to t_2 : A$, then $\Gamma \vdash_{\mathsf{CG}} t : A_1$, $\Gamma \vdash_{\mathsf{CG}} t_1 : A_2$, and $\Gamma, x : \mathsf{Nat} \vdash_{\mathsf{CG}} t_2 : A_3$ for types A_1, A_2, A_3.
iii. If $\Gamma \vdash_{\mathsf{CG}} (t_1, t_2) : A$, then $\Gamma \vdash_{\mathsf{CG}} t_1 : A_1$ and $\Gamma \vdash_{\mathsf{CG}} t_2 : A_2$ for types A_1 and A_2.
iv. If $\Gamma \vdash_{\mathsf{CG}} \mathsf{fst}\ t : A$, then $\Gamma \vdash_{\mathsf{CG}} t : A_1$ for some type A_1.
v. If $\Gamma \vdash_{\mathsf{CG}} \mathsf{snd}\ t : A$, then $\Gamma \vdash_{\mathsf{CG}} t : A_1$ for some type A_1.
vi. If $\Gamma \vdash_{\mathsf{CG}} \lambda(x : B).t : A$, then $\Gamma, x : B \vdash_{\mathsf{CG}} t : A_1$ for some type A_1.
vii. If $\Gamma \vdash_{\mathsf{CG}} t_1\ t_2 : A$, then $\Gamma \vdash_{\mathsf{CG}} t_1 : A_1$ and $\Gamma \vdash_{\mathsf{CG}} t_2 : A_2$ for types A_1 and A_2.

Lemma A.15 (Inversion for Term Precision for Core Grady)**.** *Suppose $\Gamma \vdash t_1 \sqsubseteq t_2$. Then either $t_2 = \mathsf{box}_A\ t_1$ and $\Gamma \vdash_{\mathsf{CG}} t_1 : A$, or one of the following holds:*

i. If $t_1 = x$, then $t_2 = x$ and $x : A \in \Gamma$.
ii. If $t_1 = \mathsf{box}$, then $t_2 = \mathsf{box}$.
iii. If $t_1 = \mathsf{unbox}$, then $t_2 = \mathsf{unbox}$.
iv. If $t_1 = 0$, then $t_2 = 0$.
v. If $t_1 = \mathsf{triv}$, then $t_2 = \mathsf{triv}$.
vi. If $t_1 = \mathsf{succ}\ t_1'$, then $t_2 = \mathsf{succ}\ t_2'$ and $\Gamma \vdash t_1' \sqsubseteq t_2'$.
vii. If $t_1 = \mathsf{case}\ t_1' : \mathsf{Nat}\ \mathsf{of}\ 0 \to t_2', (\mathsf{succ}\ x) \to t_3'$, then $t_2 = \mathsf{case}\ t_4' : \mathsf{Nat}\ \mathsf{of}\ 0 \to t_5', (\mathsf{succ}\ x) \to t_6', \Gamma \vdash t_1' \sqsubseteq t_4', \Gamma \vdash t_2' \sqsubseteq t_5'$, and $\Gamma, x : \mathsf{Nat} \vdash t_3' \sqsubseteq t_6'$.
viii. If $t_1 = (t_1', t_2')$, then $t_2 = (t_3', t_4')$, $\Gamma \vdash t_1' \sqsubseteq t_3'$, and $\Gamma \vdash t_2' \sqsubseteq t_4'$.
ix. If $t_1 = \mathsf{fst}\ t_1'$, then $t_2 = \mathsf{fst}\ t_2'$ and $\Gamma \vdash t_1' \sqsubseteq t_2'$.
x. If $t_1 = \mathsf{snd}\ t_1'$, then $t_2 = \mathsf{snd}\ t_2'$ and $\Gamma \vdash t_1' \sqsubseteq t_2'$.
xi. If $t_1 = \lambda(x : A_1).t_1$, then $t_2 = \lambda(x : A_2).t_2$ and $\Gamma, x : A_2 \vdash t_1 \sqsubseteq t_2$ and $A_1 \sqsubseteq A_2$.
xii. If $t_1 = t_1'\ t_2'$, then one of the following is true:
  a. $t_2 = t_3'\ t_4'$, $\Gamma \vdash t_3 \sqsubseteq t_3'$, and $\Gamma \vdash t_4 \sqsubseteq t_4'$
  b. $t_1' = \mathsf{unbox}_A$ and $t_2 = t_2'$
xiii. If $t_1 = \mathsf{unbox}_A\ t_1'$, then $t_2 = t_1'$ and $\Gamma \vdash_{\mathsf{CG}} t_1' : ?$.
xiv. If $t_1 = \mathsf{error}_{A_1}$, then $\Gamma \vdash_{\mathsf{CG}} t_2 : A_2$ and $A_1 \sqsubseteq A_2$.

Proof. The proof of this result holds by induction on $\Gamma \vdash t_1 \sqsubseteq t_2$. □

B Proofs

B.1 Proof of Lifted Retract (Lemma 2.10)

We only show that $\widehat{\mathsf{box}_A}; \widehat{\mathsf{unbox}_A} = \mathsf{id}_A$, because the case when a dynamic type error is raised is similar using the fact that A and B must have the same skeleton or one could not compose $\widehat{\mathsf{box}_A}$ and $\widehat{\mathsf{unbox}_B}$. This implies that A and B only differ at an atomic type.

This is a proof by induction on the form of A.

Case. Suppose A is atomic. Then:

$$\widehat{\mathsf{box}_A}; \widehat{\mathsf{unbox}_A} = \mathsf{box}_A; \mathsf{unbox}_A = \mathsf{id}_A$$

Case. Suppose A is ?. Then:

$$\begin{array}{rcl} \widehat{\mathsf{box}_A}; \widehat{\mathsf{unbox}_A} & = & \widehat{\mathsf{box}_?}; \widehat{\mathsf{unbox}_?} \\ & = & \mathsf{id}_?; \mathsf{id}_? \\ & = & \mathsf{id}_? \\ & = & \mathsf{id}_A \end{array}$$

Case. Suppose $A = A_1 \to A_2$. Then:

$$\begin{array}{rcl} \widehat{\mathsf{box}_A}; \widehat{\mathsf{unbox}_A} & = & \widehat{\mathsf{box}_{(A_1 \to A_2)}}; \widehat{\mathsf{unbox}_{(A_1 \to A_2)}} \\ & = & (\widehat{\mathsf{unbox}_{A_1}} \to \widehat{\mathsf{box}_{A_2}}); (\widehat{\mathsf{box}_{A_1}} \to \widehat{\mathsf{box}_{A_2}}) \\ & = & (\widehat{\mathsf{box}_{A_1}}; \widehat{\mathsf{unbox}_{A_1}}) \to (\widehat{\mathsf{box}_{A_2}}; \widehat{\mathsf{unbox}_{A_2}}) \end{array}$$

By two applications of the induction hypothesis we know the following:

$$\widehat{\mathsf{box}_{A_1}}; \widehat{\mathsf{unbox}_{A_1}} = \mathsf{id}_{A_1} \quad \text{and} \quad \widehat{\mathsf{box}_{A_2}}; \widehat{\mathsf{unbox}_{A_2}} = \mathsf{id}_{A_2}$$

Thus, we know the following:

$$\begin{array}{rcl} (\widehat{\mathsf{box}_{A_1}}; \widehat{\mathsf{unbox}_{A_1}}) \to (\widehat{\mathsf{box}_{A_2}}; \widehat{\mathsf{unbox}_{A_2}}) & = & \mathsf{id}_{A_1} \to \mathsf{id}_{A_2} \\ & = & \mathsf{id}_{A_1 \to A_2} \\ & = & \mathsf{id}_A \end{array}$$

Case. Suppose $A = A_1 \times A_2$. Then:

$$\begin{array}{rcl} \widehat{\mathsf{box}_A}; \widehat{\mathsf{unbox}_A} & = & \widehat{\mathsf{box}_{(A_1 \times A_2)}}; \widehat{\mathsf{unbox}_{(A_1 \times A_2)}} \\ & = & (\widehat{\mathsf{box}_{A_1}} \times \widehat{\mathsf{box}_{A_2}}); (\widehat{\mathsf{unbox}_{A_1}} \times \widehat{\mathsf{box}_{A_2}}) \\ & = & (\widehat{\mathsf{box}_{A_1}}; \widehat{\mathsf{unbox}_{A_1}}) \times (\widehat{\mathsf{box}_{A_2}}; \widehat{\mathsf{unbox}_{A_2}}) \end{array}$$

By two applications of the induction hypothesis we know the following:

$$\widehat{\mathsf{box}_{A_1}}; \widehat{\mathsf{unbox}_{A_1}} = \mathsf{id}_{A_1} \quad \text{and} \quad \widehat{\mathsf{box}_{A_2}}; \widehat{\mathsf{unbox}_{A_2}} = \mathsf{id}_{A_2}$$

Thus, we know the following:

$$\begin{array}{rcl} (\widehat{\mathsf{box}_{A_1}}; \widehat{\mathsf{unbox}_{A_1}}) \times (\widehat{\mathsf{box}_{A_2}}; \widehat{\mathsf{unbox}_{A_2}}) & = & \mathsf{id}_{A_1} \times \mathsf{id}_{A_2} \\ & = & \mathsf{id}_{A_1 \times A_2} \\ & = & \mathsf{id}_A \end{array}$$

B.2 Proof of Lemma 2.11

We must show that the function

$$\mathsf{S}_{A,B} : \mathsf{Hom}_C(A,B) \longrightarrow \mathsf{Hom}_{\mathcal{S}}(\mathsf{S}A, \mathsf{S}B)$$

is injective.

So suppose $f \in \mathsf{Hom}_C(A,B)$ and $g \in \mathsf{Hom}_C(A,B)$ such that $\mathsf{S}f = \mathsf{S}g : \mathsf{S}A \longrightarrow \mathsf{S}B$. Then we can easily see that:

$$\begin{array}{rcl} \mathsf{S}f & = & \widehat{\mathsf{unbox}_A}; f; \widehat{\mathsf{box}_B} \\ & = & \widehat{\mathsf{unbox}_A}; g; \widehat{\mathsf{box}_B} \\ & = & \mathsf{S}g \end{array}$$

But, we have the following equalities:

$$\begin{array}{c} \widehat{\mathsf{unbox}_A}; f; \widehat{\mathsf{box}_B} = \widehat{\mathsf{unbox}_A}; g; \widehat{\mathsf{box}_B} \\ \widehat{\mathsf{box}_A}; \widehat{\mathsf{unbox}_A}; f; \widehat{\mathsf{box}_B}; \widehat{\mathsf{unbox}_B} = \widehat{\mathsf{box}_A}; \widehat{\mathsf{unbox}_A}; g; \widehat{\mathsf{box}_B}; \widehat{\mathsf{unbox}_B} \\ \mathsf{id}_A; f; \widehat{\mathsf{box}_B}; \widehat{\mathsf{unbox}_B} = \mathsf{id}_A; g; \widehat{\mathsf{box}_B}; \widehat{\mathsf{unbox}_B} \\ \mathsf{id}_A; f; \mathsf{id}_B = \mathsf{id}_A; g; \mathsf{id}_B \\ f = g \end{array}$$

The previous equalities hold due to Lemma 2.10.

B.3 Proof of Type Consistency in the Model (Lemma 5.2)

This is a proof by induction on the form of $A \sim B$.

Case: $\dfrac{}{A \sim A}$ refl

Choose $c_1 = c_2 = \mathsf{id}_A : A \longrightarrow A$.

Case: $\dfrac{}{A \sim ?}$ box

Choose $c_1 = \mathsf{Box}_A : A \longrightarrow ?$ and $c_2 = \mathsf{Unbox}_A : ? \to A$.

Case: $\dfrac{}{? \sim A}$ unbox

Choose $c_1 = \mathsf{Unbox}_A : ? \longrightarrow A$ and $c_2 = \mathsf{Box}_A : A \to ?$.

Case: $\dfrac{A_2 \sim A_1 \quad B_1 \sim B_2}{(A_1 \to B_1) \sim (A_2 \to B_2)} \to$

By the induction hypothesis there exists four casting morphisms $c_1' : A_1 \longrightarrow A_2$, $c_2' : A_2 \longrightarrow A_1$, $c_3' : B_1 \longrightarrow B_2$, and $c_4' : B_2 \longrightarrow B_1$. Choose $c_1 = c_2' \to c_3' : (A_1 \to B_1) \longrightarrow (A_2 \to B_2)$ and $c_2 = c_1' \to c_4' : (A_2 \to B_2) \longrightarrow (A_1 \to B_1)$.

Case: $\dfrac{A_1 \sim A_2 \quad B_1 \sim B_2}{(A_1 \times B_1) \sim (A_2 \times B_2)} \times$

By the induction hypothesis there exists four casting morphisms $c_1' : A_1 \longrightarrow A_2$, $c_2' : A_2 \longrightarrow A_1$, $c_3' : B_1 \longrightarrow B_2$, and $c_4' : B_2 \longrightarrow B_1$. Choose $c_1 = c_1' \times c_3' : A_1 \times B_1 \longrightarrow A_2 \times B_2$ and $c_2 = c_2' \times c_4' : A_2 \times B_2 \longrightarrow A_1 \times B_1$.

B.4 Proof of Interpretation of Types Theorem 5.3

First, we show how to interpret the rules of Surface Grady and then Core Grady. This is a proof by induction on $\Gamma \vdash_{\mathsf{SG}} t : A$.

Case: $\dfrac{x : A \in \Gamma}{\Gamma \vdash_{\mathsf{SG}} x : A}$ var

Suppose without loss of generality that $[\![\Gamma]\!] = A_1 \times \cdots \times A_i \times \cdots \times A_j$ where $A_i = A$. We know that $j > 0$ or the assumed typing derivation would not hold. Then take $[\![x]\!] = \pi_i : [\![\Gamma]\!] \longrightarrow A$.

Case: $\dfrac{}{\Gamma \vdash_{\mathsf{SG}} \mathsf{triv} : \mathsf{Unit}}$ Unit

Take $[\![\mathsf{triv}]\!] = \mathsf{triv}_{[\![\Gamma]\!]} : [\![\Gamma]\!] \longrightarrow \mathsf{Unit}$ where $\mathsf{triv}_{[\![\Gamma]\!]}$ is the unique terminal arrow that exists because C is cartesian closed.

Case: $\dfrac{}{\Gamma \vdash_{\mathsf{SG}} 0 : \mathsf{Nat}}$ zero

Take $[\![0]\!] = \mathsf{triv}_{[\![\Gamma]\!]}; \mathsf{z} : [\![\Gamma]\!] \longrightarrow \mathsf{Nat}$ where $\mathsf{z} : \mathsf{Unit} \longrightarrow \mathsf{Nat}$ exists because C has an NRNO.

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

First, by the induction hypothesis there is a morphism $[\![t]\!] : [\![\Gamma]\!] \longrightarrow A$. Now we have two cases to consider, one when $A = ?$ and one when $A = \mathsf{Nat}$. Consider the former. Then interpret $[\![\mathsf{succ}\ t]\!] = [\![t]\!]; \mathsf{unbox}_{\mathsf{Nat}}; \mathsf{succ} : [\![\Gamma]\!] \longrightarrow \mathsf{Nat}$ where $\mathsf{succ} : \mathsf{Nat} \longrightarrow \mathsf{Nat}$ exists because C has an NRNO. Similarly, when $A = \mathsf{Nat}$, $[\![\mathsf{succ}\ t]\!] = [\![t]\!]; \mathsf{succ} : [\![\Gamma]\!] \longrightarrow \mathsf{Nat}$.

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

By three applications of the induction hypothesis we have the following morphisms:

$$\begin{array}{l} [\![t]\!] : [\![\Gamma]\!] \longrightarrow C \\ [\![t_1]\!] : [\![\Gamma]\!] \longrightarrow A_1 \\ [\![t_2]\!] : [\![\Gamma]\!] \times \mathsf{Nat} \longrightarrow A_2 \end{array}$$

In addition, we know $A_1 \sim A$ and $A_2 \sim A$ by assumption, and hence, by type consistency in the model (Lemma 5.2) we know there are casting morphisms $c_1 : A_1 \longrightarrow A$ and $c_2 : A_2 \longrightarrow A$. Now every gradual λ-model has an NRNO (Definition 2.4, Definition 2.5), and so, there is a unique morphism:

$$\mathsf{case}_{[\![\Gamma]\!],A}\langle [\![t_1]\!]; c_1, [\![t_2]\!]; c_2\rangle : [\![\Gamma]\!] \times \mathsf{Nat} \longrightarrow A$$

At this point we have two cases to consider: one when $C = ?$ and one when $C = \mathsf{Nat}$. Consider the former. Then we have the following:

$$\begin{array}{l} [\![\mathsf{case}\ t\ \mathsf{of}\ 0 \to t_1, (\mathsf{succ}\ x) \to t_2]\!] \\ = \langle \mathsf{id}_{[\![\Gamma]\!]}, [\![t]\!]; \mathsf{unbox}_{\mathsf{Nat}}\rangle; \mathsf{case}_{[\![\Gamma]\!],A}\langle [\![t_1]\!]; c_1, [\![t_2]\!]; c_2\rangle \\ : [\![\Gamma]\!] \longrightarrow A \end{array}$$

In the second case we have the following:

$$\begin{array}{l} [\![\mathsf{case}\ t\ \mathsf{of}\ 0 \to t_1, (\mathsf{succ}\ x) \to t_2]\!] \\ = \langle \mathsf{id}_{[\![\Gamma]\!]}, [\![t]\!]\rangle; \mathsf{case}_{[\![\Gamma]\!],A}\langle [\![t_1]\!]; c_1, [\![t_2]\!]; c_2\rangle \\ : [\![\Gamma]\!] \longrightarrow A \end{array}$$

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

By two applications of the induction hypothesis there are two morphisms $[\![t_1]\!] : [\![\Gamma]\!] \longrightarrow A$ and $[\![t_2]\!] : [\![\Gamma]\!] \longrightarrow B$. Then using the fact that $\mathcal{C}$ is cartesian we take $[\![(t_1, t_2)]\!] = \langle [\![t_1]\!], [\![t_2]\!] \rangle : [\![\Gamma]\!] \longrightarrow A \times B$.

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

First, by the induction hypothesis there is a morphism $[\![t]\!] : [\![\Gamma]\!] \longrightarrow B$. Now we have two cases to consider, one when $B = ?$ and one when $B = A_1 \times A_2$ for some types A_1 and A_2. Consider the former. We then know that it must be the case that $A_1 \times A_2 = ? \times ?$. Thus, we obtain the following interpretation $[\![\mathsf{fst}\ t]\!] = [\![t]\!]; \mathsf{unbox}_{(?\times?)}; \pi_1 : [\![\Gamma]\!] \longrightarrow ?$. Similarly, when $B = A_1 \times A_2$, then $[\![\mathsf{fst}\ t]\!] = [\![t]\!]; \pi_1 : [\![\Gamma]\!] \longrightarrow A_1$.

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

First, by the induction hypothesis there is a morphism $[\![t]\!] : [\![\Gamma]\!] \longrightarrow B$. Now we have two cases to consider, one when $B = ?$ and one when $B = A_1 \times A_2$ for some types A_1 and A_2. Consider the former. We then know that it must be the case that $A_1 \times A_2 = ? \times ?$. Thus, we obtain the following interpretation $[\![\mathsf{snd}\ t]\!] = [\![t]\!]; \mathsf{unbox}_{(?\times?)}; \pi_2 : [\![\Gamma]\!] \longrightarrow ?$. Similarly, when $B = A_1 \times A_2$, then $[\![\mathsf{snd}\ t]\!] = [\![t]\!]; \pi_2 : [\![\Gamma]\!] \longrightarrow A_2$.

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

By the induction hypothesis there is a morphism $[\![t]\!] : [\![\Gamma]\!] \times A \longrightarrow B$. Then take $[\![\lambda(x : A).t]\!] = \mathsf{curry}([\![t]\!]) : [\![\Gamma]\!] \longrightarrow (A \to B)$, where

$$\mathsf{curry} : \mathsf{Hom}_{\mathcal{C}}(X \times Y, Z) \longrightarrow \mathsf{Hom}_{\mathcal{C}}(X, Y \to Z)$$

exists because $\mathcal{C}$ is closed.

Case: $\dfrac{\Gamma \vdash_{\mathsf{SG}} t_1 : C \qquad A_2 \sim A_1 \qquad \Gamma \vdash_{\mathsf{SG}} t_2 : A_2 \qquad \mathsf{fun}(C) = A_1 \to B_1}{\Gamma \vdash_{\mathsf{SG}} t_1\ t_2 : B_1} \to_e$

By the induction hypothesis there are two morphisms $[\![t_1]\!] : [\![\Gamma]\!] \longrightarrow C$ and $[\![t_2]\!] : [\![\Gamma]\!] \longrightarrow A_2$. In addition, by assumption we know that $A_2 \sim A_1$, and hence, by type consistency in the model (Lemma 5.2) there are casting morphisms $c_1 : A_2 \longrightarrow A_1$ and $c_2 : A_1 \longrightarrow A_2$. We have two cases to consider, one when $C = ?$ and one when $C = A_1 \to B_1$. Consider the former. Then we have the interpretation:

$$[\![t_1\ t_2]\!] = \langle [\![t_1]\!]; \mathsf{unbox}_{(?\to?)}, [\![t_2]\!]; c_1 \rangle; \mathsf{app}_{A_1, B_1} : [\![\Gamma]\!] \longrightarrow B_1$$

Similarly, for the case when $C = A_1 \to B_1$ we have the interpretation:

$$[\![t_1\ t_2]\!] = \langle [\![t_1]\!], [\![t_2]\!]; c_1 \rangle; \mathsf{app}_{A_1, B_1} : [\![\Gamma]\!] \longrightarrow B_1$$

Note that $\mathsf{app}_{A_1, B_1} : (A_1 \to B_1) \times A_1 \longrightarrow B_1$ exists because the model is cartesian closed.

Next we turn to Core Grady, but we do not show every rule, because it is similar to what we have already shown above except without casting morphism, and so we only show the case for the box and unbox rules, and the error rule.

The first two cases use the well-known bijection:

$$\begin{array}{rcl} \mathsf{Hom}_{\mathcal{C}}(A, B) & \cong & \mathsf{Hom}_{\mathcal{C}}(\mathsf{Unit} \times A, B) \\ & \cong & \mathsf{Hom}_{\mathcal{C}}(\mathsf{Unit}, A \to B) \end{array}$$

When $f \in \mathsf{Hom}_{\mathcal{C}}(A, B)$, then we denote by $\mathsf{curry}(f)$, the morphism $\mathsf{curry}(f) \in \mathsf{Hom}_{\mathcal{C}}(\mathsf{Unit}, A \to B)$.

Case: $\dfrac{}{\Gamma \vdash_{\mathsf{CG}} \mathsf{box}_A : A \to ?}\mathsf{box}$

We have the following interpretation:

$$[\![\mathsf{box}_A]\!] = \mathsf{triv}_{[\![\Gamma]\!]}; \mathsf{curry}(\mathsf{Box}_A) : [\![\Gamma]\!] \longrightarrow (A \to ?)$$

Case: $\dfrac{}{\Gamma \vdash_{\mathsf{CG}} \mathsf{unbox}_A : ? \to A}\mathsf{unbox}$

We have the following interpretation:

$$\begin{array}{l} [\![\mathsf{unbox}_A]\!] \\ = \mathsf{triv}_{[\![\Gamma]\!]}; \mathsf{curry}(\mathsf{Unbox}_A) \\ : [\![\Gamma]\!] \longrightarrow (? \to A) \end{array}$$

Case: $\dfrac{}{\Gamma \vdash_{\mathsf{CG}} \mathsf{error}_A : A}\mathsf{error}$

We have the following interpretation:

$$[\![\mathsf{error}_A]\!] = \mathsf{error}_{[\![\Gamma]\!], A} : [\![\Gamma]\!] \longrightarrow A$$

B.5 Proof of Interpretation of Evaluation (Theorem 5.4)

This proof requires the following corollary to Lemma 5.2, and the following lemma called inversion for typing.

Corollary B.1. *Suppose* $(\mathcal{T}, \mathcal{C}, ?, \mathsf{T}, \mathsf{split}, \mathsf{squash}, \mathsf{box}, \mathsf{unbox})$ *is a gradual* λ*-model. Then we know the following:*

i. If $A \sim A$, *then* $c_1 = c_2 = \mathsf{id}_A : A \longrightarrow A$.

ii. If $A \sim ?$, *then there are casting morphisms:*

$$\begin{array}{rcl} c_1 & = & \mathsf{Box}\, A : A \longrightarrow ? \\ c_2 & = & \mathsf{Unbox}\, A : ? \longrightarrow A \end{array}$$

iii. If $? \sim A$, *then there are casting morphisms:*

$$\begin{array}{rcl} c_1 & = & \mathsf{Unbox}\, A : ? \longrightarrow A \\ c_2 & = & \mathsf{Box}\, A : A \longrightarrow ? \end{array}$$

iv. If $A_1 \to B_1 \sim A_2 \to B_2$, *then there are casting morphisms:*

$$\begin{array}{rcl} c & = & c_1 \to c_2 : (A_1 \to B_1) \longrightarrow (A_2 \to B_2) \\ c' & = & c_3 \to c_4 : (A_2 \to B_2) \longrightarrow (A_1 \to B_1) \end{array}$$

where $c_1 : A_2 \longrightarrow A_1$ *and* $c_2 : B_1 \longrightarrow B_2$, *and* $c_3 : A_1 \longrightarrow A_2$ *and* $c_4 : B_2 \longrightarrow B_1$.

v. If $A_1 \times B_1 \sim A_2 \times B_2$, then there are casting morphisms:

$$\begin{array}{lcl} c & = & c_1 \times c_2 : (A_1 \times B_1) \longrightarrow (A_2 \times B_2) \\ c' & = & c_3 \times c_4 : (A_2 \times B_2) \longrightarrow (A_1 \times B_1) \end{array}$$

where $c_1 : A_1 \longrightarrow A_2$ and $c_2 : B_1 \longrightarrow B_2$, and $c_3 : A_2 \longrightarrow A_1$ and $c_4 : B_2 \longrightarrow B_1$.

Proof. This proof holds by the construction of the casting morphisms from the proof of the previous result, and the fact that the type consistency rules are unique for each type. □

Lemma B.2 (Inversion for Typing)**.**

i. If $\Gamma \vdash_{\mathsf{CG}} \mathsf{succ}\, t : A$, then $A = \mathsf{Nat}$ and $\Gamma \vdash_{\mathsf{CG}} t : \mathsf{Nat}$.
ii. If $\Gamma \vdash_{\mathsf{CG}} \mathsf{case}\, t\, \mathsf{Nat}\, \mathsf{of}\, x \to t_1, (\mathsf{succ}\, x) \to t_2 : A$, then $\Gamma \vdash_{\mathsf{CG}} t : \mathsf{Nat}$, $\Gamma \vdash_{\mathsf{CG}} t_1 : A$, and $\Gamma, x : \mathsf{Nat} \vdash_{\mathsf{CG}} t_2 : A$.
iii. If $\Gamma \vdash_{\mathsf{CG}} (t_1, t_2) : A$, then there are types B and C, such that, $A = B \times C$, $\Gamma \vdash_{\mathsf{CG}} t_1 : B$, and $\Gamma \vdash_{\mathsf{CG}} t_2 : C$.
iv. If $\Gamma \vdash_{\mathsf{CG}} \mathsf{fst}\, t : A$, then there is a type B, such that, $\Gamma \vdash_{\mathsf{CG}} t : A \times B$.
v. If $\Gamma \vdash_{\mathsf{CG}} \mathsf{snd}\, t : A$, then there is a type B, such that, $\Gamma \vdash_{\mathsf{CG}} t : B \times A$.
vi. If $\Gamma \vdash_{\mathsf{CG}} \lambda(x : A).t : A$, then there are types B and C, such that, $A = B \to C$ and $\Gamma, x : B \vdash_{\mathsf{CG}} t : C$.
vii. If $\Gamma \vdash_{\mathsf{CG}} t_1\, t_2 : A$, then there is a type B, such that, $\Gamma \vdash_{\mathsf{CG}} t_1 : B \to A$ and $\Gamma \vdash_{\mathsf{CG}} t_2 : B$.

Proof. Each case of this proof holds trivially by induction on the assumed typing derivation, because there is only one typing rule per term constructor. □

This proof holds by induction on the form of $t_1 \rightsquigarrow t_2$ with an appeal to inversion for typing on $\Gamma \vdash_{\mathsf{SG}} t_1 : A$ and $\Gamma \vdash_{\mathsf{SG}} t_2 : A$. We only show the cases for the retract rules, and the error rule, because the others are well-known to hold within any cartesian closed category; see [13] or [6]. We will routinely use the interpretation given in the proof of Theorem 5.3 and summarized in Figure 7 throughout this proof without mention.

The cases to follow will make use of the following result, essentially the semantic equivalent to an instance of the β-rule, that holds in any cartesian closed category:

$$\begin{array}{l} \langle \mathsf{triv}_C; \mathsf{curry}(g), f \rangle; \mathsf{app}_{A,B} \\ = \langle \mathsf{triv}_C, f \rangle; (\mathsf{curry}(g) \times \mathsf{id}_A); \mathsf{app}_{A,B} \\ = \langle \mathsf{triv}_C, f \rangle; \mathsf{snd}; g \\ = f; g \end{array}$$

where $g : A \longrightarrow B$ and $f : C \longrightarrow A$. Note that $\mathsf{app}_{A,B} : (A \to B) \times A \longrightarrow B$ exists, because C is a cartesian closed category.

Case: $\dfrac{}{\mathsf{unbox}_A\, (\mathsf{box}_A\, t) \rightsquigarrow t}$ retract

We know by assumption that $\Gamma \vdash_{\mathsf{CG}} \mathsf{unbox}_A\, (\mathsf{box}_A\, t) : A$ and $\Gamma \vdash_{\mathsf{CG}} t : A$. By interpretation for typing (Theorem 5.3) and using the above equation we obtain the following morphisms:

$$\begin{array}{l} [\![\mathsf{box}_A\, t]\!] \\ = \langle \mathsf{triv}_{[\![\Gamma]\!]}; \mathsf{curry}(\mathsf{Box}_A), [\![t]\!] \rangle; \mathsf{app}_{A,?} \\ = [\![t]\!]; \mathsf{Box}_A \\ : [\![\Gamma]\!] \longrightarrow ? \end{array}$$

$$\begin{array}{l} [\![\mathsf{unbox}_A\, (\mathsf{box}_A\, t)]\!] \\ = \langle \mathsf{triv}_{[\![\Gamma]\!]}; \mathsf{curry}(\mathsf{Unbox}_A), [\![\mathsf{box}_A\, t]\!] \rangle; \mathsf{app}_{?,A} \\ = [\![\mathsf{box}_A\, t]\!]; \mathsf{Unbox}_A \\ = [\![t]\!]; \mathsf{Box}_A; \mathsf{Unbox}_A \\ : [\![\Gamma]\!] \longrightarrow A \end{array}$$

where $[\![t]\!] : [\![\Gamma]\!] \longrightarrow A$. At this point it is easy to see that $[\![t]\!]; \mathsf{Box}_A; \mathsf{Unbox}_A = [\![t]\!]; \mathsf{id}_A = [\![t]\!]$. Thus, we obtain our result.

Case: $\dfrac{A \neq B}{\mathsf{unbox}_A\, (\mathsf{box}_B\, t) \rightsquigarrow \mathsf{error}_A}$ raise

This case follows similarly to the previous case. Using the semantic β-equation given above, then we will obtain $[\![t]\!]; \mathsf{Box}_B; \mathsf{Unbox}_A = \mathsf{error}_{[\![\Gamma]\!], A}$ using the error axioms from the definition of the gradual λ-model (Definition 2.5).

Case: $\dfrac{x : B \vdash_{\mathsf{CG}} \mathcal{E}[x] : A}{\mathcal{E}[\mathsf{error}_B] \rightsquigarrow \mathsf{error}_A}$ error

This case follows from a case analysis over the structure of $\mathcal{E}$, and then using the error axioms from the definition of the gradual λ-model (Definition 2.5).

B.6 Proof of Interpretation of Evaluation for $\lambda^{\rightrightarrows}_{\to}$ (Theorem 6.2)

This proof holds by induction on the form of $t_1 \rightsquigarrow t_2$, but must appeal to inversion for typing. We only show the cases for the casting rules, because the others are well-known to hold within any cartesian closed category; see [13] or [6]. We will routinely use Theorem 5.3 throughout this proof without mention.

Case.

$$\frac{}{v : T \Rightarrow T \rightsquigarrow v}\ \textsc{id-atom}$$

We know by assumption that $\Gamma \vdash v : T$, in addition, we always know that $T \sim T$. Thus, We have a morphism $[\![v : T \Rightarrow T]\!] = [\![v]\!]; c_1 : [\![\Gamma]\!] \longrightarrow T$ based on the interpretation of typing. It must be the case that $c_1 = \mathsf{id}_T : T \longrightarrow T$ by Corollary B.1. Therefore, $[\![v : T \Rightarrow T]\!] = [\![v]\!]; c_1 = [\![v]\!] : [\![\Gamma]\!] \longrightarrow T$.

Case.

$$\frac{}{v : ? \Rightarrow ? \rightsquigarrow v}\ \textsc{id-u}$$

Similar to the previous case.

Case: $\dfrac{}{v : R \Rightarrow ? \Rightarrow R \rightsquigarrow v}$ succeed

By inversion for typing the typing derivation for $v : R \Rightarrow ? \Rightarrow R$ is as follows:

$$\dfrac{\dfrac{\Gamma \vdash v : R \quad R \sim ?}{\Gamma \vdash v : R \Rightarrow ? : ?} \qquad ? \sim R}{\Gamma \vdash v : R \Rightarrow ? \Rightarrow R : R}$$

By the induction hypothesis we have the morphism $[\![v]\!] : [\![\Gamma]\!] \longrightarrow R$. As we can see we will first use Box_R and then Unbox_R based on Corollary B.1. Thus, $[\![v : R \Rightarrow ? \Rightarrow R]\!] = [\![v]\!]; \mathsf{Box}_R; \mathsf{Unbox}_R = [\![v]\!]$, because Box_R and Unbox_R form a retract (Lemma 2.15).

Case: $\dfrac{A = (A_1 \to B_1) \qquad B = (A_2 \to B_2)}{(v_1 : A \Rightarrow B)\ v_2 \rightsquigarrow v_1\ (v_2 : A_2 \Rightarrow A_1) : B_1 \Rightarrow B_2}\ {\to}_{\Rightarrow}$

First, by inversion for typing we know $\Gamma \vdash v_1 : A_1 \to B_1$, $\Gamma \vdash v_2 : A_2$, and $(A_1 \to B_1) \sim (A_2 \to B_2)$. Then by the induction hypothesis and Corollary B.1 we have the following morphisms:

$$\begin{array}{l} [\![v_1]\!] : [\![\Gamma]\!] \longrightarrow (A_1 \to B_1) \\ [\![v_2]\!] : [\![\Gamma]\!] \longrightarrow A_2 \\ c_1 : A_2 \longrightarrow A_1 \\ c_2 : B_1 \longrightarrow B_2 \end{array}$$

We must show the following:

$$\begin{array}{l} [\![(v_1 : (A_1 \to B_1) \Rightarrow (A_2 \to B_2))\ v_2]\!] \\ = \langle [\![v_1]\!], (c_1 \to c_2), [\![v_2]\!] \rangle; \mathsf{app}_{A_2,B_2} \\ = \langle [\![v_1]\!], [\![v_2]\!]; c_1 \rangle; \mathsf{app}_{A_1,B_1}; c_2 \end{array}$$

The previous equation holds as follows where we give properties in between the equations for the reason why they hold:

$$\begin{array}{l} \langle [\![v_1]\!], [\![v_2]\!]; c_1 \rangle; \mathsf{app}_{A_1,B_1}; c_2 \\ \text{(Cartesian Products)} \\ = \langle [\![v_1]\!], [\![v_2]\!] \rangle; (\mathsf{id}_{A_1 \to B_1} \times c_1); \mathsf{app}_{A_1,B_1}; c_2 \\ \text{(Naturality)} \\ = \langle [\![v_1]\!], [\![v_2]\!] \rangle; (\mathsf{id}_{A_1 \to B_1} \times c_1); \\ \quad ((\mathsf{id}_{A_1} \to c_2) \times \mathsf{id}_{A_1}); \mathsf{app}_{A_1,B_2} \\ \text{(Closure)} \\ = \langle [\![v_1]\!], [\![v_2]\!] \rangle; (\mathsf{curry}((\mathsf{id}_{A_1 \to B_1} \times c_1); \\ \quad ((\mathsf{id}_{A_1} \to c_2) \times \mathsf{id}_{A_1}); \mathsf{app}_{A_1,B_2}) \times \mathsf{id}_{A_2}); \mathsf{app}_{A_2,B_2} \\ \text{(Closure)} \\ = \langle [\![v_1]\!], [\![v_2]\!] \rangle; ((c_1 \to c_2) \times \mathsf{id}_{A_2}); \mathsf{app}_{A_2,B_2} \\ \text{(Cartesian Products)} \\ = \langle [\![v_1]\!]; (c_1 \to c_2), [\![v_2]\!]; \mathsf{id}_{A_2} \rangle; \mathsf{app}_{A_2,B_2} \\ = \langle [\![v_1]\!]; (c_1 \to c_2), [\![v_2]\!] \rangle; \mathsf{app}_{A_2,B_2} \end{array}$$

Case.

$$\dfrac{A \sim R \quad A \neq R \quad A \neq ?}{v : A \Rightarrow ? \rightsquigarrow v : A \Rightarrow R \Rightarrow ?}\ \textsc{expand}_1$$

We know by assumption that $\Gamma \vdash v : A$. By the induction hypothesis and Lemma 5.2 we have the following morphisms:

$$\begin{array}{l} [\![v]\!] : [\![\Gamma]\!] \longrightarrow A \\ c_1 : A \longrightarrow R \end{array}$$

We must show the following:

$$\begin{array}{rcl} [\![v : A \Rightarrow ?]\!] & = & [\![v]\!]; \mathsf{Box}_A \\ & = & [\![v]\!]; c_1; \mathsf{Box}_R \end{array}$$

However, notice that given the constraints above, it must be the case that $R = T$ or $R = ? \to ?$. If the former is true, then $A = T$ by the definition of consistency and the constraints above, but this implies that $c_1 = \mathsf{id}_T$ by Corollary B.1, and the result follows. However, consider the case when $R = ? \to ?$. Then given the constraints above $A = A_1 \to A_2$. Thus, $c_1 = (\mathsf{unbox}_{A_1} \to \mathsf{box}_{A_2})$ by Corollary B.1. In addition, it must be the case that $\mathsf{Box}_R = \mathsf{squash}_{(? \to ?)}$ by the definition of Box_R, but by inspection of the definition of Box_A we have the following:

$$\begin{array}{rcl} \mathsf{Box}_A & = & \mathsf{Box}_{(A_1 \to A_2)} \\ & = & (\mathsf{unbox}_{A_1} \to \mathsf{box}_{A_2}); \mathsf{squash}_{(? \to ?)} \\ & = & c_1; \mathsf{squash}_{(? \to ?)} \\ & = & c_1; \mathsf{Box}_R \end{array}$$

Thus, we obtain our result.

Case.

$$\dfrac{A \sim R \quad A \neq R \quad A \neq ?}{v : ? \Rightarrow A \rightsquigarrow v : ? \Rightarrow R \Rightarrow A}\ \textsc{expand}_2$$

This case is similar to the previous case, except that the interpretation uses Unbox_A and Unbox_R instead of Box_A and Box_R.

B.7 Proof of Congruence of Type Consistency Along Type Precision (Lemma A.10)

Proof. The proofs of both parts are similar, and so we only show a few cases of the first part, but the omitted cases follow similarly.

Proof of part one. This is a proof by induction on the form of $A_1 \sqsubseteq A_1'$.

Case: $\dfrac{}{A_1 \sqsubseteq ?}\ ?$

In this case $A_1' = ?$. Suppose $A_1 \sim A_2$. Then it suffices to show that $? \sim A_2$, but this easily follows.

Case: $\dfrac{A \sqsubseteq C \quad B \sqsubseteq D}{(A \to B) \sqsubseteq (C \to D)}\ \to$

In this case $A_1 = A \to B$ and $A_1' = C \to D$. Suppose $A_1 \sim A_2$. Then by inversion for type consistency it must be the case that either $A_2 = ?$, or $A_2 = A' \to B'$, $A \sim A'$, and $B \sim B'$.

Consider the former. Then it suffices to show that $A_1' \sim ?$, but this easily follows.

Consider the case when $A_2 = A' \to B'$, $A \sim A'$, and $B \sim B'$. It suffices to show that $(C \to D) \sim (A' \to B')$ which follows from $A' \sim C$ and $D \sim B'$. Thus, it suffices to show that latter. By assumption we know the following:

$$\begin{array}{l} A \sqsubseteq C \text{ and } A \sim A' \\ B \sqsubseteq D \text{ and } B \sim B' \end{array}$$

Now by two applications of the induction hypothesis we obtain $C \sim A'$ and $D \sim B'$. By symmetry the former implies $A' \sim C$ and we obtain our result.

□

B.8 Proof of Gradual Guarantee Part One (Lemma 7.2)

This is a proof by induction on $\Gamma \vdash_{\mathsf{SG}} t : A$. We only show the most interesting cases, because the others follow similarly.

Case: $\dfrac{x : A \in \Gamma}{\Gamma \vdash_{\mathsf{SG}} x : A}\ \textsc{var}$

In this case $t = x$. Suppose $t \sqsubseteq t'$. Then it must be the case that $t' = x$. If $x : A \in \Gamma$, then there is a type A' such that $x : A' \in \Gamma'$ and $A \sqsubseteq A'$. Thus, choose $B = A'$ and the result follows.

Case: $\dfrac{\Gamma \vdash_{\mathsf{SG}} t_1 : A' \quad \mathsf{nat}(A') = \mathsf{Nat}}{\Gamma \vdash_{\mathsf{SG}} \mathsf{succ}\, t_1 : \mathsf{Nat}}\ \textsc{succ}$

In this case $A = \mathsf{Nat}$ and $t = \mathsf{succ}\, t_1$. Suppose $t \sqsubseteq t'$ and $\Gamma \sqsubseteq \Gamma'$. Then by definition it must be the case that $t' = \mathsf{succ}\, t_2$ where $t_1 \sqsubseteq t_2$. By the induction hypothesis $\Gamma' \vdash_{\mathsf{SG}} t_2 : B'$ where $A' \sqsubseteq B'$. Since $\mathsf{nat}(A') = \mathsf{Nat}$ and $A' \sqsubseteq B'$, then it must be the case that $\mathsf{nat}(B') = \mathsf{Nat}$ by Lemma A.9. At this point we obtain our result by choosing $B = \mathsf{Nat}$, and reapplying the rule above.

Case: $\dfrac{\begin{array}{ll}\Gamma \vdash_{\mathsf{SG}} t_1 : C & \mathsf{nat}(C) = \mathsf{Nat} \\ \Gamma \vdash_{\mathsf{SG}} t_2 : A_1 & A_1 \sim A \\ \Gamma, x : \mathsf{Nat} \vdash_{\mathsf{SG}} t_3 : A_2 & A_2 \sim A\end{array}}{\Gamma \vdash_{\mathsf{SG}} \mathsf{case}\, t_1\, \mathsf{of}\, 0 \to t_2, (\mathsf{succ}\, x) \to t_3 : A}\ \mathsf{Nat}_e$

In this case $t = \mathsf{case}\, t_1\, \mathsf{of}\, 0 \to t_2, (\mathsf{succ}\, x) \to t_3$. Suppose $t \sqsubseteq t'$ and $\Gamma \sqsubseteq \Gamma'$. This implies that $t' = \mathsf{case}\, t_1'\, \mathsf{of}\, 0 \to t_2', (\mathsf{succ}\, x) \to t_3'$ such that $t_1 \sqsubseteq t_1'$, $t_2 \sqsubseteq t_2'$, and $t_3 \sqsubseteq t_3'$. Since $\Gamma \sqsubseteq \Gamma'$ then $(\Gamma, x : \mathsf{Nat}) \sqsubseteq (\Gamma', x : \mathsf{Nat})$. By the induction hypothesis we know the following:

$$\begin{array}{l}\Gamma' \vdash_{\mathsf{SG}} t_1' : C' \text{ for } C \sqsubseteq C' \\ \Gamma' \vdash_{\mathsf{SG}} t_2 : A_1' \text{ for } A_1 \sqsubseteq A_1' \\ \Gamma', x : \mathsf{Nat} \vdash_{\mathsf{SG}} t_3 : A_2' \text{ for } A_2 \sqsubseteq A_2'\end{array}$$

By assumption we know that $A_1 \sim A$, $A_2 \sim A$, and $\Gamma \sqsubseteq \Gamma'$. By the induction hypothesis we know that $A_1 \sqsubseteq A_1'$ and $A_2 \sqsubseteq A_2'$, so by using Lemma A.10 we may obtain that $A_1' \sim A$ and $A_2' \sim A$. At this point choose $B = A$ and we obtain our result by reapplying the rule.

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

In this case $A = A_1 \times A_2$ and $t = (t_1, t_2)$. Suppose $t \sqsubseteq t'$ and $\Gamma \sqsubseteq \Gamma'$. This implies that $t' = (t_1', t_2')$ where $t_1 \sqsubseteq t_1'$ and $t_2 \sqsubseteq t_2'$.

By the induction hypothesis we know:

$$\begin{array}{l}\Gamma' \vdash_{\mathsf{SG}} t_1' : A_1' \text{ and } A_1 \sqsubseteq A_1' \\ \Gamma' \vdash_{\mathsf{SG}} t_2' : A_2' \text{ and } A_2 \sqsubseteq A_2'\end{array}$$

Then choose $B = A_1' \times A_2'$ and the result follows by reapplying the rule above and the fact that $(A_1 \times A_2) \sqsubseteq (A_1' \times A_2')$.

Case: $\dfrac{\Gamma, x : A_1 \vdash_{\mathsf{SG}} t_1 : B_1}{\Gamma \vdash_{\mathsf{SG}} \lambda(x : A_1).t_1 : A_1 \to B_1} \to_i$ In this case $A_1 \to B_2$ and $t = \lambda(x : A_1).t_1$. Suppose $t \sqsubseteq t'$ and $\Gamma \sqsubseteq \Gamma'$. Then it must be the case that $t' = \lambda(x : A_2).t_2$, $t_1 \sqsubseteq t_2$, and $A_1 \sqsubseteq A_2$. Since $\Gamma \sqsubseteq \Gamma'$ and $A_1 \sqsubseteq A_2$, then $(\Gamma, x : A_1) \sqsubseteq (\Gamma', x : A_2)$ by definition. Thus, by the induction hypothesis we know the following:

$$\Gamma', x : A_2 \vdash_{\mathsf{SG}} t_1' : B_2 \text{ and } B_1 \sqsubseteq B_2$$

Choose $B = A_2 \to B_2$ and the result follows by reapplying the rule above and the fact that $(A_1 \to B_1) \sqsubseteq (A_2 \to B_2)$.

Case: $\dfrac{\begin{array}{l}\Gamma \vdash_{\mathsf{SG}} t_1 : C \quad \mathsf{fun}(C) = A_1 \to B_1 \\ \Gamma \vdash_{\mathsf{SG}} t_2 : A_2 \quad A_2 \sim A_1\end{array}}{\Gamma \vdash_{\mathsf{SG}} t_1\, t_2 : B_1} \to_e$

In this case $A = B_1$ and $t = t_1\, t_2$. Suppose $t \sqsubseteq t'$ and $\Gamma \sqsubseteq \Gamma'$. The former implies that $t' = t_1'\, t_2'$ such that $t_1 \sqsubseteq t_1'$ and $t_2 \sqsubseteq t_2'$. By the induction hypothesis we know the following:

$$\begin{array}{l}\Gamma' \vdash_{\mathsf{SG}} t_1' : C' \text{ for } C \sqsubseteq C' \\ \Gamma' \vdash_{\mathsf{SG}} t_2' : A_2' \text{ for } A_2 \sqsubseteq A_2'\end{array}$$

We know by assumption that $A_2 \sim A_1$. Since $C \sqsubseteq C'$ and $\mathsf{fun}(C) = A_1 \to B_1$, then $\mathsf{fun}(C') = A_1' \to B_1'$ where $A_1 \sqsubseteq A_1'$ and $B_1 \sqsubseteq B_1'$ by Lemma A.9. Furthermore, we know $A_2 \sim A_1$ and $A_2 \sqsubseteq A_2'$ and $A_1 \sqsubseteq A_1'$, then we know $A_2' \sim A_1'$ by Corollary A.11. So choose $B = B_1'$. Then reapply the rule above and the result follows, because $B_1 \sqsubseteq B_1'$.

B.9 Proof of Simulation of More Precise Programs (Lemma 7.3)

This is a proof by induction on $\Gamma \vdash_{\mathsf{CG}} t_1 : A_1$. We only give the most interesting cases. All others follow similarly. Throughout the proof we implicitly make use of typability inversion (Lemma A.14) when applying the induction hypothesis.

Case: $\dfrac{\Gamma \vdash_{\mathsf{CG}} t : \mathsf{Nat}}{\Gamma \vdash_{\mathsf{CG}} \mathsf{succ}\, t : \mathsf{Nat}}\ \textsc{succ}$

In this case $t_1 = \mathsf{succ}\, t$ and $A = \mathsf{Nat}$. Suppose $\Gamma \vdash_{\mathsf{CG}} t_1' : A'$. By inversion for term precision we must consider the following cases:

i. $t_1' = \mathsf{succ}\, t'$ and $\Gamma \vdash t \sqsubseteq t'$
ii. $t_1' = \mathsf{box}_{\mathsf{Nat}}\, t_1$ and $\Gamma \vdash_{\mathsf{CG}} t_1 : \mathsf{Nat}$

Proof of part i. Suppose $t_1' = \mathsf{succ}\, t'$, $\Gamma \vdash t \sqsubseteq t'$, and $t_1 \rightsquigarrow t_2$. Then $t_2 = \mathsf{succ}\, t''$ and $t \rightsquigarrow t''$. Then by the induction hypothesis we know that there is some t''' such that $t' \rightsquigarrow^* t'''$ and $\Gamma \vdash t'' \sqsubseteq t'''$. Choose $t_2' = \mathsf{succ}\, t'''$ and the result follows.

Proof of part ii. Suppose $t_1' = \mathsf{box}_{\mathsf{Nat}}\, t_1$, $\Gamma \vdash_{\mathsf{CG}} t_1 : \mathsf{Nat}$, and $t_1 \rightsquigarrow t_2$. Then choose $t_2' = \mathsf{box}_{\mathsf{Nat}}\, t_2$, and the result follows, because we know by type preservation that $\Gamma \vdash_{\mathsf{CG}} t_2 : \mathsf{Nat}$, and hence, $\Gamma \vdash t_2 \sqsubseteq t_2'$.

Case: $\dfrac{\Gamma \vdash_{CG} t : \mathsf{Nat} \quad \Gamma \vdash_{CG} t_3 : A \quad \Gamma, x : \mathsf{Nat} \vdash_{CG} t_4 : A}{\Gamma \vdash_{CG} \mathsf{case}\ t\colon \mathsf{Nat}\ \mathsf{of}\ 0 \to t_3, (\mathsf{succ}\ x) \to t_4 : A}\ \mathsf{Nat}_e$

In this case $t_1 = \mathsf{case}\ t\colon \mathsf{Nat}\ \mathsf{of}\ 0 \to t_3, (\mathsf{succ}\ x) \to t_4$. Suppose $\Gamma \vdash_{CG} t_1' : A'$. Then inversion of term precision implies that one of the following must hold:

i. $t_1' = \mathsf{case}\ t'\colon \mathsf{Nat}\ \mathsf{of}\ 0 \to t_3', (\mathsf{succ}\ x) \to t_4', \Gamma \vdash t \sqsubseteq t', \Gamma \vdash t_3 \sqsubseteq t_3'$, and $\Gamma, x : \mathsf{Nat} \vdash t_4 \sqsubseteq t_4'$
ii. $t_1' = \mathsf{box}_A\ t_1$ and $\Gamma \vdash_{CG} t_1 : A$

Proof of part i. Suppose $t_1' = \mathsf{case}\ t'\colon \mathsf{Nat}\ \mathsf{of}\ 0 \to t_3', (\mathsf{succ}\ x) \to t_4', \Gamma \vdash t \sqsubseteq t', \Gamma \vdash t_3 \sqsubseteq t_3'$, and $\Gamma, x : \mathsf{Nat} \vdash t_4 \sqsubseteq t_4'$.

We case split over $t_1 \rightsquigarrow t_2$.

Case. Suppose $t = 0$ and $t_2 = t_3$. Since $\Gamma \vdash t_1 \sqsubseteq t_1'$ we know that it must be the case that $t' = 0$ and $t_1' \rightsquigarrow t_3'$ by inversion for term precision or t_1' would not be typable which is a contradiction. Thus, choose $t_2' = t_3'$ and the result follows.

Case. Suppose $t = \mathsf{succ}\ t''$ and $t_2 = [t''/x]t_4$. Since $\Gamma \vdash t_1 \sqsubseteq t_1'$ we know that $t' = \mathsf{succ}\ t'''$, or t_1' would not be typable, and $\Gamma \vdash t'' \sqsubseteq t'''$ by inversion for term precision. In addition, $t_1' \rightsquigarrow [t'''/x]t_4'$. Choose $t_2 = [t'''/x]t_4'$. Then it suffices to show that $\Gamma \vdash [t''/x]t_4 \sqsubseteq [t'''/x]t_4'$ by substitution for term precision (Lemma A.13).

Case. Suppose a congruence rule was used. Then $t_2 = \mathsf{case}\ t''\colon \mathsf{Nat}\ \mathsf{of}\ 0 \to t_3'', (\mathsf{succ}\ x) \to t_4''$. This case will follow straightforwardly by induction and a case split over which congruence rule was used.

Proof of part ii. Suppose $t_1' = \mathsf{box}_A\ t_1, \Gamma \vdash_{CG} t_1 : A$, and $t_1 \rightsquigarrow t_2$. Then choose $t_2' = \mathsf{box}_A\ t_2$, and the result follows, because we know by type preservation that $\Gamma \vdash_{CG} t_2 : A$, and hence, $\Gamma \vdash t_2 \sqsubseteq t_2'$.

Case: $\dfrac{\Gamma \vdash_{CG} t : A \times B}{\Gamma \vdash_{CG} \mathsf{fst}\ t : A}\ \times_{e_1}$

In this case $t_1 = \mathsf{fst}\ t$. Suppose $\Gamma \vdash t_1 \sqsubseteq t_1'$ and $\Gamma \vdash_{CG} t_1' : A'$. Then inversion for term precision implies that one of the following must hold:

i. $t_1' = \mathsf{fst}\ t'$ and $\Gamma \vdash t \sqsubseteq t'$
ii. $t_1' = \mathsf{box}_A\ t_1$ and $\Gamma \vdash_{CG} t_1 : A$

We only consider the proof of part i, because the other follows similarly to the previous case. Case split over $t_1 \rightsquigarrow t_2$.

Case. Suppose $t = (t_3', t_3'')$ and $t_2 = t_3'$. By inversion for term precision it must be the case that $t' = (t_4', t_4'')$ because $\Gamma \vdash t_1 \sqsubseteq t_1'$ or else t_1' would not be typable. In addition, this implies that $\Gamma \vdash t_3' \sqsubseteq t_4'$ and $\Gamma \vdash t_3'' \sqsubseteq t_4''$. Thus, $t_1' \rightsquigarrow t_4'$. Thus, choose $t_2' = t_4'$ and the result follows.

Case. Suppose a congruence rule was used. Then $t_2 = \mathsf{fst}\ t''$. This case will follow straightforwardly by induction and a case split over which congruence rule was used.

Case: $\dfrac{\Gamma, x : A_1 \vdash_{CG} t : A_2}{\Gamma \vdash_{CG} \lambda(x : A_1).t : A_1 \to A_2}\ \to_i$

In this case $t_1 = \lambda(x : A_1).t$ and $A = A_1 \to A_2$. Suppose $\Gamma \vdash t_1 \sqsubseteq t_1'$ and $\Gamma \vdash_{CG} t_1' : A'$. Then inversion of term precision implies that one of the following must hold:

i. $t_1' = \lambda(x : A_1').t'$
ii. $t_1' = \mathsf{box}_A\ t_1$ and $\Gamma \vdash_{CG} t_1 : A$

We only consider the proof of part i. The reduction relation does not reduce under λ-expressions. Hence, $t_2 = t_1$, and thus, choose $t_2' = t_1'$, and the case trivially follows.

Case: $\dfrac{\Gamma \vdash_{CG} t_3 : A_1 \to A_2 \quad \Gamma \vdash_{CG} t_4 : A_1}{\Gamma \vdash_{CG} t_3\ t_4 : A_2}\ \to_e$

In this case $t_1 = t_3\ t_4$. Suppose $\Gamma \vdash t_1 \sqsubseteq t_1'$ and $\Gamma \vdash_{CG} t_1' : A'$. Then by inversion for term prevision we know one of the following is true:

i. $t_1' = t_3'\ t_4', \Gamma \vdash t_3 \sqsubseteq t_3'$, and $\Gamma \vdash t_4 \sqsubseteq t_4'$
ii. $t_1' = \mathsf{box}_{A_2}\ t_1$ and $\Gamma \vdash_{CG} t_1 : A$
iii. $t_3 = \mathsf{unbox}_{A_2}, t_1' = t_4$, and $\Gamma \vdash_{CG} t_4 : ?$

Proof of part i. Suppose $t_1' = t_3'\ t_4', \Gamma \vdash t_3 \sqsubseteq t_3'$, and $\Gamma \vdash t_4 \sqsubseteq t_4'$.

We case split on the from of $t_1 \rightsquigarrow t_2$.

Case. Suppose $t_3 = \lambda(x : A_1).t_5$ and $t_2 = [t_4/x]t_5$. Then by inversion for term precision we know that $t_3' = \lambda(x : A_1').t_5'$ and $\Gamma, x : A_2' \vdash t_5 \sqsubseteq t_5'$, because $\Gamma \vdash t_3 \sqsubseteq t_3'$ and the requirement that t_1' is typable. Choose $t_2' = [t_4'/x]t_5'$ and it is easy to see that $t_1' \rightsquigarrow [t_4'/x]t_4'$. We know that $\Gamma, x : A_2' \vdash t_5 \sqsubseteq t_5'$ and $\Gamma \vdash t_4 \sqsubseteq t_4'$, and hence, by Lemma A.13 we know that $\Gamma \vdash [t_4/x]t_5 \sqsubseteq [t_4'/x]t_5'$, and we obtain our result.

Case. Suppose $t_3 = \mathsf{unbox}_A$, $t_4 = \mathsf{box}_A\ t_5$, and $t_2 = t_5$. Then by inversion for term prevision $t_3' = \mathsf{unbox}_A$, $t_4' = \mathsf{box}_A\ t_5'$, and $\Gamma \vdash t_5 \sqsubseteq t_5'$. Note that $t_4' = \mathsf{box}_A\ t_5'$ and $\Gamma \vdash t_5 \sqsubseteq t_5'$ hold even though there are two potential rules that could have been used to construct $\Gamma \vdash t_4 \sqsubseteq t_4'$. Choose $t_2' = t_5'$ and it is easy to see that $t_1' \rightsquigarrow t_5'$. Thus, we obtain our result.

Case. Suppose $t_3 = \mathsf{unbox}_A$, $t_4 = \mathsf{box}_B\ t_5$, $A \neq B$, and $t_2 = \mathsf{error}_B$. Then $t_3' = \mathsf{unbox}_A$ and $t_4' = \mathsf{box}_B\ t_5'$. Choose $t_2' = \mathsf{error}_B$ and it is easy to see that $t_1' \rightsquigarrow t_5'$. Finally, we can see that $\Gamma \vdash t_2 \sqsubseteq t_2'$ by reflexivity.

Case. Suppose a congruence rule was used. Then $t_2 = t_5'\ t_6'$. This case will follow straightforwardly by induction and a case split over which congruence rule was used.

Proof of part ii. We know that $t_1 = t_3\ t_4$. Suppose

$t_1' = \mathsf{box}_{A_2}\ t_1$ and $\Gamma \vdash_{\mathsf{CG}} t_1 : A$. If $t_1 \rightsquigarrow t_2$, then $t_1' = (\mathsf{box}_{A_2}\ t_1) \rightsquigarrow (\mathsf{box}_{A_2}\ t_2)$. Thus, choose $t_2' = \mathsf{box}_{A_2}\ t_2$.

Proof of part iii. We know that $t_1 = t_3\ t_4$. Suppose $t_3 = \mathsf{unbox}_{A_2}$, $t_1' = t_4$, and $\Gamma \vdash_{\mathsf{CG}} t_4 : ?$. Then $t_1 = \mathsf{unbox}_{A_2}\ t_4$. We case split over $t_1 \rightsquigarrow t_2$. We have three cases to consider.

Suppose $t_4 = \mathsf{box}_{A_2}\ t_5$ and $t_2 = t_5$. Then choose $t_2' = t_4 = t_1'$, and we obtain our result.

Suppose $t_4 = \mathsf{box}_{A_3}\ t_5$, $A_2 \neq A_3$, and $t_2 = \mathsf{error}_{A_2}$. Then choose $t_2' = t_4 = t_1'$, and we obtain our result.

Suppose a congruence rule was used. Then $t_2 = t_3\ t_4'$. This case will follow straightforwardly by induction.